Microsoft

Modern Web Development

Understanding domains, technologies, and user experience

Professional

Dino Esposito

Microsoft

Modern Web Development: Understanding domains, technologies, and user experience

Dino Esposito

PUBLISHED BY
Microsoft Press
A division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2015934865
ISBN: 978-1-5093-0001-3

Printed and bound in the United States of America.

First Printing

Acquisitions and Developmental Editor: Devon Musgrave
Project Editor: Steve Sagman
Editorial Production: Waypoint Press
Technical Reviewer: Marc Young
Copyeditor: Roger LeBlanc
Indexer: Toni Culley
Cover: Twist Creative • Seattle and Joel Panchot

To my wife Silvia.

You make me feel sandy like a clepsydra. I get empty and filled all the time; but it's such a thin kind of sand that even when I'm full, without you, I just feel empty.

—Dino

Contents at a glance

PART I UNDERSTANDING THE DOMAIN

CHAPTER 1 Conducting a thorough domain analysis 3
CHAPTER 2 Selecting the supporting architecture 19
CHAPTER 3 UX-driven design 47
CHAPTER 4 Architectural options for a web solution 63
CHAPTER 5 The layered architecture 87

PART II DEVELOPMENT

CHAPTER 6 ASP.NET state of the art 103
CHAPTER 7 Whys, wherefores, and technical aspects of ASP.NET Core 1.0 109
CHAPTER 8 Core of ASP.NET MVC 133
CHAPTER 9 Core of Bootstrap 171
CHAPTER 10 Organizing the ASP.NET MVC project 217
CHAPTER 11 Presenting data 247
CHAPTER 12 Editing data 279
CHAPTER 13 Persistence and modeling 313

PART III USER EXPERIENCE

CHAPTER 14 Creating more interactive views 335
CHAPTER 15 Pros and cons of responsive design 365
CHAPTER 16 Making websites mobile-friendly 381

Contents

Introduction . . . . *xvii*

PART I UNDERSTANDING THE DOMAIN

Chapter 1 Conducting a thorough domain analysis 3

Domain-driven design to the rescue. 4

Introducing design driven by the domain. 4

Clearing up common misconceptions about DDD 4

Introducing the ubiquitous language 6

Creating a vocabulary of domain-specific terms 6

Keeping business and code in sync 8

Introducing the bounded context 10

Discovering bounded contexts 10

Implementing bounded contexts 12

Introducing context mapping. 15

Examining relationships between bounded contexts 15

Introducing event storming 17

Having unlimited modeling space 17

Finding events in the domain 17

Leading the discussion 18

Summary. 18

Chapter 2 Selecting the supporting architecture 19

It’s all about business logic 19
Examining the application logic 20
Examining the domain logic 21
Exploring patterns for the business logic 22

Using a single model 24
Exploring the elements of an object-oriented domain model 24
Putting business rules inside 26
Discovering aggregates 28
Exploring the role of domain services 30

Implementing command and query separation 32
Working with the Command and Query Separation principle 33
Implementing CQRS 36

Introducing message-based formulation 39
Ad-hoc infrastructure 39
Introducing event sourcing 43

Summary 46

Chapter 3 UX-driven design 47

Why a top-down approach is better than a bottom-up one 48
Foundation of the bottom-up approach 48
Planning with a top-down approach 50

Looking at user experience from an architectural perspective 52
UX is not UI 52
Explaining UXDD in three steps 55

Why UXDD is beneficial to nearly everybody 59

Summary 62

Chapter 4 Architectural options for a web solution 63

Assessing the available web solutions 64
Deciding on the best framework 64
Laying out a solution 65

Examining the role of ASP.NET Core 1.0 66
Considering ASP.NET as the starting point 66
Examining the architectural dependencies in ASP.NET Core 1.0 68
Exploring the reasons to choose ASP.NET Core 1.0 69
Determining if you should use ASP.NET Web Forms 72
Examining a common scenario 72
ASP.NET Web Forms at a glance 73
What's still good with Web Forms 73
Why you should move away from Web Forms 74
Determining if you should use ASP.NET MVC 76
ASP.NET MVC at a glance 76
What's good with ASP.NET MVC 77
Weak points of ASP.NET MVC 78
Examining the role of ASP.NET Web API 79
Moving from WCF to Web API 79
Comparing ASP.NET Web API and ASP.NET MVC 80
Talking about REST 82
Using Web API in ASP.NET Core 1.0 82
Single-page applications 83
Setting up a SPA 84
Hybrid SPA 84
Weak points of a SPA 85
Summary 85

Chapter 5 The layered architecture 87

Beyond classic three-tier systems 87
Working with a three-tier architecture today 88
Fifty shades of gray areas 89
The presentation layer 90
The user experience 90
The input model 91
The view model 92

The application layer 92
Entry point in the system's back end 93
Orchestration of business processes. 94

The domain layer 95
The mythical domain model 95
The equally mythical concept of domain services 96
A more pragmatic view of domain modeling. 96

The infrastructure layer 97
Current state storage. 97
Event stores. 98
Caching layers. 99
External services. 99

Summary. 100

PART II DEVELOPMENT

Chapter 6 ASP.NET state of the art 103

Web flavors 103
The web could have been different 104
Classic web is the winner 104

ASP.NET is feature-complete 105
No more to add is no more to add. 105
Is it full potential or software obsolescence? 106

ASP.NET Core 1.0 has no new functions 106
It's about the new runtime 106
It's about the business model 107
It's about the development model 107

What is the state of ASP.NET? 108

Chapter 7 Whys, wherefores, and technical aspects of ASP.NET Core 1.0 109

The background of ASP.NET Core 110

The cost of a large memory footprint 110

Reconsidering the cloud as the silver bullet 111

Making the case for the necessity of a different programming model. 111

The impact on everyday work 112

The ASP.NET Core runtime at a glance 113

The DNX host 113

Hosting web applications in DNX 115

ASP.NET Core HTTP pipeline 116

ASP.NET Core for ASP.NET developers 118

Creating a new project 118

Application startup 122

Application settings. 126

Authentication 128

Other aspects of web programming 129

Summary. 132

Chapter 8 Core of ASP.NET MVC 133

Routing incoming requests. 134

Simulating the ASP.NET MVC runtime 134

Exploring the URL routing HTTP module 137

Using application routes 138

Exploring the controller class 145

Looking at aspects of a controller. 145

Writing controller classes 147

Processing input data. 152

Manual parameter binding 152

Model binding 153

Producing action results 158
Wrapping results 158
Returning HTML markup 160
Returning JSON content 167
Summary. 169

Chapter 9 Core of Bootstrap 171

Bootstrap at a glance 171
LESS and the foundation of Bootstrap 172
Setting up Bootstrap 174
Putting Bootstrap into perspective. 175
Responsive layouts 176
The grid system 176
Screen-based rendering 178
Taxonomy of today's web elements 183
Restyling basic HTML elements 183
Restyling list HTML elements. 193
A look at more advanced components 196
Bootstrap extensions 202
Autocompletion 202
Date picking 209
Custom components 211
Summary. 215

Chapter 10 Organizing the ASP.NET MVC project 217

Planning the project solution 217
Mapping projects to the Layered Architecture pattern 218
Application startup 220
Examining application services 225
Adding in other assets. 227
Creating presentation layouts 228
Serving resources more effectively 231
Working with Bundling 231
Using minification 236

Examining other aspects 238
Exploring error handling. 238
Configuring user authentication 241
Summary. 246

Chapter 11 Presenting data 247

Structuring an HTML view. 247
Exploring the view model. 247
Examining the page layout. 250
Presenting the elements of a view 254
Displaying a list of data items. 258
Creating a grid view. 258
Adding paging capabilities. 263
Adding scrolling capabilities to page elements 268
Adding a detail view. 271
Popover views. 271
Drill-down views. 274
Summary. 277

Chapter 12 Editing data 279

A common form for the login page 279
Presenting the form. 279
Processing posted data. 281
Input forms. 284
The Post-Redirect-Get pattern 284
Form validation. 286
Modal input forms 296
Quick tips for improving the user experience .304
Using date pickers is great, but... 305
Using autocompletion instead of long drop-down lists 307
Miscellaneous tips for large input forms 310
Summary. 311

Chapter 13 Persistence and modeling 313

Examining the different flavors of a model 313
- The persistence model 314
- The domain model 315
- The input model 317
- The view model 318

Designing a persistence layer 319
- Using an implicit and legacy data model 320
- Using Entity Framework 322
- The Repository pattern 326

Polyglot persistence 328
- Polyglot persistence by example 329
- Costs of polyglot persistence 330

Summary 332

PART III USER EXPERIENCE

Chapter 14 Creating more interactive views 335

Exposing JSON content 336
- Creating JSON endpoints 336
- Negotiating content 339
- Solving the cross-origin puzzle 340

Designing a Web API 344
- Purpose of the ASP.NET Web API 344
- Web API in the context of ASP.NET MVC 345
- Securing a standalone Web API 346

Pulling content 349
- The Ajax core 349
- The jQuery tools 351
- Binding data to the current DOM 354

Pushing content to the client 357
- ASP.NET SignalR at a glance 358
- Monitoring remote tasks 359
- Other scenarios for ASP.NET SignalR 362

Summary 363

Chapter 15 Pros and cons of responsive design 365

Foundation of Responsive Web Design 365

A brief history of RWD 366

CSS media queries 366

RWD and device independence 371

Adapting RWD to non-desktop devices 373

Dealing with images 374

Dealing with fonts 376

Dealing with orientation 378

Summary 379

Chapter 16 Making websites mobile-friendly 381

Adapting views to the actual device 381

The best of HTML5 for mobile scenarios 382

Feature detection 384

Client-side device detection 386

A look into the future 389

Device-friendly images 390

The ImageEngine platform 390

Resizing images automatically 391

Serving device-friendly views 392

What's the best way to offer mobile content? 392

Server-side detection 394

Summary 399

Index 401

Introduction

No later than the summer of 2008, I gave a few public talks about the future of the web. Customers who hired me at the time heard from this expert voice that the web of the (near) future would be significantly different than what it was looking like in 2008. At the time, the brilliant future of the web seemed to be in the hands of compiled code run from within the browser.

JavaScript? It's dead, at last! ASP.NET? It's gone, thankfully!

The future as I saw it back then (along with many other experts) had only rich-client technologies in store for millions of us. And Microsoft Silverlight stood at the center of the new web universe.

If you started hibernating in 2008 and woke up any time in the past three or even four years, you found a different world than I, or possibly you, had imagined. It was solidly server-side-based and different from what the expectations were. Today, you find a web world in which JavaScript reigns and, with it, a ton of ad hoc tools and frameworks.

Customers who paid good money to hear my expert voice back in 2008 tell them to invest in Silverlight are now paying good money to switch back more or less to where they were in 2008.

Well, not exactly.

This book comes at a weird time, but it's not a weird book. Two decades of web experience taught us that real revolutions happen when, mostly due to rare astral alignments, a bunch of people happen to have the same programming needs. So it was for Ajax, and so it is today for responsive and interactive front ends. JavaScript has been revived because it is the simplest way for programmers to achieve goals. And because it is still effective enough to make solutions easy to sell.

Planning a web solution today means having a solid server-side environment to serve rich and interactive HTML pages, styled with CSS and actioned by JavaScript. Even though a lot of new ad hoc technologies have been developed, the real sticking points with modern applications (which are for the most part web applications) are domain analysis and the supporting architecture. Everything else revolves around the implementation of a few common practices for a few common tasks, some of which are relatively new requirements—for example, push notifications from the server.

In this book, you will find a summary of practices and techniques that guarantee effective solutions for your customers. The point today is no longer to use the latest release of the latest platform or framework. The point is just to give customers what they really want. Tools to build software exist; ideas and plans make the difference.

Who should read this book

This book exists to help web developers improve their skills. The inspiring principle for the book is that today we mostly write software to mirror a piece of the real world, rather than to bend the real world to a piece of technology.

If you just want to do your day-to-day job better, learning from the mistakes that others made and looking at the same mistakes you made with a more thoughtful perspective, then you should definitely read this book.

Assumptions

This book assumes you are familiar with the Microsoft web stack. This experience can range from having done years of Web Forms development to being a JavaScript angel. The main focus is ASP.NET MVC, because that will be the standard with ASP.NET Core and remain so for the future of the ASP.NET platform. Here are some key goals for readers of the book: learning a method general enough so that you can start development projects with a deep understanding of the domain of the problem, select the right approach, and go forward with reliable coding practices.

This book might not be for you if…

If you're looking for a step-by-step guide to some ASP.NET MVC or perhaps Bootstrap, this book is probably not the best option you have. It does cover basic aspects of both technologies, but it hardly does that with the necessary slow pace of a beginner book.

Organization of this book

The book is divided in three parts: understanding the business domain, implementing common features, and analyzing the user experience.

Part I offers a summary of modern software architecture, with a brief overview of domain-driven design concepts and architectural patterns. The focus is on the real meaning of the expression *domain model* and examining how it differs from other flavors of models you might work with. Key to effective design today—an approach that weds domain analysis and user experience—is the separation of commands and queries into distinct stacks. This simple strategy has a number of repercussions in terms of persistence model, scalability, and actual implementation.

Part II begins with a summary of the ASP.NET MVC programming model—the way to go for web developers, especially in light of the new ASP.NET Core platform. Next, it covers Bootstrap for styling and structuring the client side of the views and looks at techniques for posting and presenting data.

Part III is all about user experience in the context of web applications. Web content is consumed through various devices and in a number of situations. This creates a need for having adaptive front ends that "respond" intelligently to the requesting devices. In this book, you'll find two perspectives regarding client responsiveness: a common responsive web design perspective and the server-side device perspective.

So, in the end, what's this book about?

It's about what you need to do and know to serve your customers in the best possible way as far as the ASP.NET platform is concerned. At the same time, the practices and the techniques discussed in the book position you well for participating in the bright future of ASP.NET Core.

Finding your best starting point in this book

Overall, I see two main ways to approach the book. One is reading it from cover to cover, paying special attention to software design and architecture first and then taking note of how those principles get applied in the context of common but isolated programming tasks. The other approach consists of treating Part I—the part on software design and architecture—as a separate book and reading it when you feel it is necessary.

If you are	Follow these steps
Relatively new to ASP.NET development but not to web development	Ideally, you should read the book cover to cover, and be sure not to skip Chapter 4, "Architectural options for a web solution."
Familiar with ASP.NET MVC or Bootstrap	Briefly skim Chapter 8, "Core of ASP.NET MVC," and Chapter 9, "Core of Bootstrap." Also, depending on your personal feelings, you might want to also skim Chapter 6, "ASP.NET state of the art," and Chapter 10, "Organizing the ASP.NET MVC project." Note that the book provides one chapter about ASP.NET Core, but that is mostly to help you form an idea about it.
Interested in practical solutions	Read Part II and Part III.

Most of the book's chapters include hands-on samples you can use to try out the concepts you just learned. No matter which sections you choose to focus on, be sure to download and install the sample applications on your system.

System requirements

To open and run provided examples, you just need a working edition of Microsoft Visual Studio.

Downloads

All sample projects can be downloaded from the following page:

http://aka.ms/ModernWebDev/downloads

Acknowledgments

A bunch of great people made this book happen: Devon Musgrave, Roger LeBlanc, Steve Sagman, and Marc Young. It's a battle-tested team that works smoothly and effectively to turn draft text into readable and, hopefully, pleasantly readable text.

When we started this book project, we expected to cover a new product named ASP.NET vNext, but the new product, now known as ASP.NET Core, is still barely in sight. In light of this, we moved the target along the way, and Devon was smart enough and flexible enough to accept my suggestions on variations to the original plan.

Although you'll find some information about ASP.NET Core in the book, a new ASP.NET Core book is on its way. Ideally, it will be from the same team!

Errata, updates, & book support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

http://aka.ms/ModernWebDev

If you discover an error that is not already listed, please submit it to us at the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *http://support.microsoft.com*.

Free ebooks from Microsoft Press

From technical overviews to in-depth information on special topics, the free ebooks from Microsoft Press cover a wide range of topics. These ebooks are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at:

http://aka.ms/mspressfree

Check back often to see what is new!

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://aka.ms/tellpress

We know you're busy, so we've kept it short with just a few questions. Your answers go directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

CHAPTER 5

The layered architecture

We shape our buildings; thereafter they shape us.

—Winston Churchill

It has been quite a few years since computer programs have been the result of monolithic software. Monolithic software is an end-to-end sequence of procedural instructions that achieve a goal. While nearly no professional developers or architects would seriously consider writing end-to-end programs today, building monoliths is the most natural way of approaching software development for newbies. Monoliths are not bad per se—it's whether the program achieves its mission or not that really matters—but monoliths become less and less useful as the complexity of the program grows. In real-world software architecture, therefore, monoliths are simply out of place. And they have been out of place for decades now.

In software, a layer hides the implementation details of a given set of functionality behind a known interface. Layers serve the purpose of Separation of Concerns and facilitate interoperability between distinct components of the same system. In an object-oriented system, a layer is essentially a set of classes implementing a given business goal. Different layers may be deployed to physical tiers, sometimes in the form of services or micro-services available over a known network protocol such as HTTP.

A layer is a segment of software that lives in-process with other layers. Layers refer to a logical, rather than physical, separation between components. The *layered architecture* I present in this chapter is probably the most widely accepted way to mix functionality and business to produce a working system.

Beyond classic three-tier systems

You might have grown up with the idea that any software system should be arranged around three segments: the presentation, business, and data layers. The *presentation* segment is made of screens (either desktop, mobile, or web interfaces) used to collect input data and present results. The *data* segment is where you deal with databases and save and read information. The *business* segment is where everything else you need to have fits in. (See Figure 5-1.)

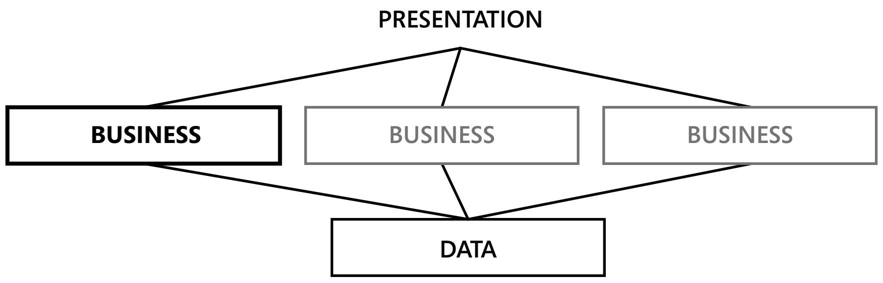

FIGURE 5-1 The classic three-tier segmentation of software architecture

Industry literature mostly refers to the architecture depicted in Figure 5-1 as a *three-tier architecture*. However, you can allocate segments both on physical tiers and logical layers. It just depends on your needs.

Working with a three-tier architecture today

The three-tier architecture has been around for a long time, but it originated at a time when most business work was either database related or restricted to external components such as mainframes. For the most part, the three-tier architecture uses a single data model that travels from the data store up to the presentation and back.

The architecture certainly doesn't prevent you from using other flavors of data-transfer objects (DTOs), but for the most part the three-tier architecture is emblematic of just one data model and is database-centric. The challenge you face these days—foreseen by Domain-Driven Design (DDD)—is matching persistence with presentation needs. Even though the core operations of any system remains Create-Read-Update-Delete (CRUD), the way in which business rules affect these core operations require the business layer to take care of way too many data transformations and way too much process orchestration.

Even though a lot of tutorials insist on describing an e-commerce platform as a plain CRUD regarding customers and orders, the reality is different. You never just add an order record to a database. You never have just a one-to-one match between the user interface of an order and the schema of the Orders table. Most likely, you don't even have the perception of an order at the presentation level. You more likely have something like a shopping cart that, once processed, produces an order record in some database tables.

Business processes are the hardest part to organize in a three-tier mental model. And business processes are not simply the most important thing in software; they're the only thing that really matters and they're the thing for which no compromises are possible. At first glance, business processes are the heart of the business tier. However, more often than not, business processes are too widespread to be easily restricted within the boundaries of an architecture that likes to have layers that can easily turn into physical tiers. Different presentation layers can trigger different business processes, and different business processes can refer to the same core behavior and the implementation of a few rules.

Although the meaning of business logic seems to be quite obvious, the right granularity of the reusable pieces of business logic is not obvious at all.

Fifty shades of gray areas

In a plain three-tier scenario, where would you fit the logic that adapts data to presentation? And where does the logic that optimizes input data for persistence belong? These questions highlight two significant gray areas that some architects still struggle with these days.

A *gray area* is an area of uncertainty or indeterminacy in some business context. In software architecture, the term also refers to a situation in which the solution to apply is not obvious and the uncertainty originates more from the availability of multiple choices than the lack of tools to solve the problem.

To clear the sky of gray areas, a slightly revisited architecture is in order. When Eric Evans first introduced Domain-Driven Design (which I discuss in Chapter 1, "Conducting a thorough domain analysis"), he also introduced the layered architecture, as depicted in Figure 5-2.

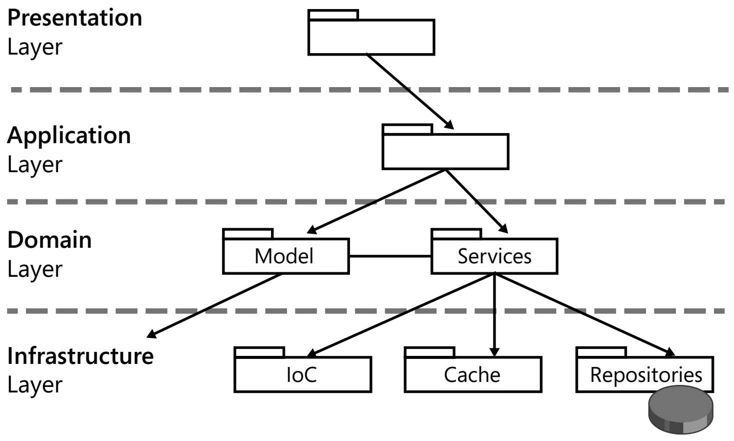

FIGURE 5-2 The layered architecture

The layered architecture has an extra layer and expands the notion of a data-access layer to that of just the provider of any necessary infrastructure, such as data access via object-relational mapping (ORM) tools, implementation of inversion of control (IoC) containers, and many other cross-cutting concerns such as security, logging, and caching.

The business layer exploded into the application and domain layers. This trend is an attempt to clear up the gray areas and make it clear that there are two types of business logic: application and domain. The *application* logic orchestrates any tasks triggered by the presentation. The *domain* logic is any core logic of the specific business that is reusable across multiple presentation layers.

The application layer aligns nearly one-to-one with the presentation layer and is where any UI-specific transformation of data takes place. The domain logic is about business rules and core business tasks using a data model that is strictly business oriented.

The presentation layer

The presentation layer is responsible for providing some user interface to accomplish any necessary tasks. The presentation layer consists of a collection of screens, either HTML forms or anything else. Today, more and more systems have multiple presentation layers. This is an ASP.NET book, so you might think that, at least in the current context, there's just one presentation layer. Not exactly, I'd say.

The mobile web is another presentation layer that must be taken into account for a web application. A mobile web presentation layer, then, can be implemented through responsive HTML templates or a completely distinct set of screens. However, that doesn't change the basic fact that the mobile web is an additional presentation layer for nearly any web application.

The user experience

No matter what kind of smart code you lovingly craft in the middle tier, no applications can be consumed by any users without a presentation front end. Furthermore, no applications can be enjoyable and effective without a well-designed user experience. However, for a long time the presentation layer has been the last concern of developers and architects.

Many architects consider presentation as the less noble part of the system—almost a detail once the business and data-access layers have been fully and successfully completed. The truth is that the presentation, as well as the business logic and data-access code, is equally necessary in a system of any complexity. These days, though, the user experience—the experience the users go through when they interact with the application—is invaluable, as I explained in Chapter 3, "UX-Driven Design."

At any rate, whether you develop the system in a top-down manner (as recommended in Chapter 3) or in a more classic bottom-up fashion, you need to understand the purpose of the presentation. Take a look at Figure 5-3.

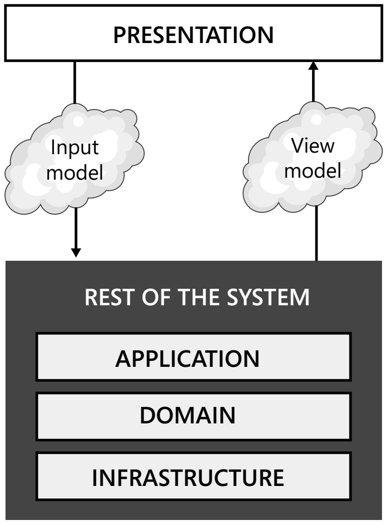

FIGURE 5-3 Describing the data that goes into and out of presentation screens

The presentation layer funnels data to the rest of the system, ideally using its own data model that reflects the structure well and organizes the data in the screens. You should render the user interface as an order entity, and not just because you end up storing date in an Orders table. For example, when you submit an order, you typically collect information like the shipping address that might or might not be related to the customer that is paying for the order. And the shipping address doesn't necessarily get stored with the order. It could be that the shipping address is communicated to the shipping company and the reference number is stored with the order.

Generally speaking, each screen in the presentation that posts a command to the back end of the system groups data into an input model and receives a response using classes in a view model. The input and view models might or might not coincide. At the same time, they might or might not coincide with any data model being used in the back end to perform actual tasks.

The input model

In ASP.NET MVC, any user's clicking originates a request that a controller class will handle. Each request is turned into an action mapped to a public method defined on a controller class. What about input data?

In ASP.NET, any input data is wrapped up in the HTTP request, either in the query string, in any form-posted data, or perhaps in HTTP headers or cookies. Input data represents the data being posted for the system to take an action on. Whatever way you look at it, it is just input parameters. You can treat input data as loose values and variables, or you can group them into a class acting as a container. The collection of input classes form the overall input model for the application.

As you'll see in a lot more detail in upcoming chapters, in ASP.NET MVC a component part of the system infrastructure—the model-binding layer—can automatically map sparse and loose variables in the HTTP request to public properties of input model classes. Here are two examples of a controller method that are equally effective:

```
public ActionResult SignIn(string username, string password, bool rememberme)
{
    ...
}

public ActionResult SignIn(LoginInputModel input)
{
    ...
}
```

In the latter case, the *LoginInputModel* class will have public properties whose names match the names of uploaded parameters:

```
public class LoginInputModel
{
    public string UserName { get; set; }
    public string Password { get; set; }
    public bool RememberMe { get; set; }
}
```

The input model carries data in the core of the system in a way that aligns one-to-one with the expectations of the user interface. Employing an input model makes it easier to design the user interface in a strongly business-oriented way. The application layer (shown in Figure 5-3) then takes care of unpacking any data and consuming it as appropriate.

The view model

Any request gets a response and, more often than not, the response you get from ASP.NET MVC is an HTML view. (Admittedly, this is not the only option, but it’s still quite the most common.) In ASP.NET MVC, the creation of an HTML view is governed by the controller, which invokes the back end of the system and gets back some response. It then selects the HTML template to use and passes the HTML template and data to an ad-hoc system component—the view engine—which will mix the template and data and produce the markup for the browser.

In ASP.NET MVC, there are a few ways to pass data to the view engine that will be incorporated in the resulting view. You can use a public dictionary such as *ViewData*, a dynamic object such as *ViewBag*, or a made-to-measure class that collects all properties to pass. Any class you create to carry data to be incorporated in the response contributes to creating the view model. The application layer is the layer that receives input-model classes and returns view-model classes:

```
public ActionResult Edit(LoginInputModel input)
{
    var model = _applicationLayer.GetSomeDataForNextView(input);
    return View(model);
}
```

More and more, in the future the ideal format for persistence will be different from the ideal format for presentation. The presentation layer is responsible for defining the clear boundaries of acceptable data, and the application layer is responsible for accepting and providing data in just those formats. If you take this approach extensively, you then fall in line with the principles outlined in Chapter 3 regarding UX-driven software design.

Note Putting the presentation layer at the center is an approach that pays off whether you use a server-side approach to the building of the web solution or a client-side solution.

The application layer

To carry on business operations, the presentation layer needs a reference to the back end of the system. The shape and color of the entry point in the business layer of the application depends on how you actually organized that.

In an ASP.NET MVC solution, you can call the infrastructure layer directly from the controller via a few repository classes. Generally, though, you want to have an intermediate layer or two in between controllers (for example, as part of the presentation layer) and repositories (for example, as part of the infrastructure layer). Have a look at Figure 5-4.

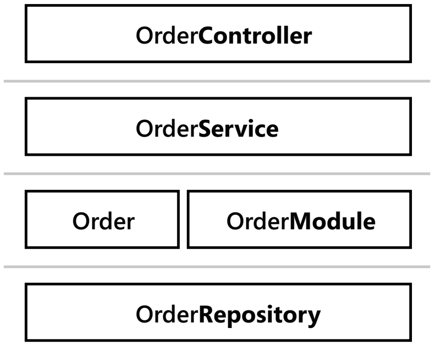

FIGURE 5-4 An aggregate-based section of the layered architecture

As you can see from the picture, you can access repositories from within controllers, but that's just a way to simplify things when the simplification of the design doesn't end up adding more pain than gain. A layered architecture is generally based on four logical layers. Each layer has its own place, and if you don't use any it should be because you have good reasons (mostly because of simplification) to do that.

Entry point in the system's back end

Each interactive element of the user interface (for example, buttons) triggers an action in the back end of the system. In some simple scenarios, the action that follows some user's clicking takes just one step to conclude. More realistically, though, the user's clicking triggers something like a workflow.

The application layer is the entry point in the back end of the system and the point of contact between the presentation and back end. The application layer consists of methods bound in an almost one-to-one fashion to the use-cases of the presentation layer. You can group methods in any way that makes sense to you.

I tend to organize application-layer methods in classes that go hand in hand with controller classes. In this way, the *OrderController* class, for example, has its own private instance of an *OrderService* class. (See Figure 5-4.)

Methods in the *OrderService* class get classes in the input model and return classes from the view model. Internally, this class performs any necessary transformation to make data render nicely on the presentation and be ready for processing in the back end.

Note I suggest you apply the same logic that leads you to split functions on controller classes to create application-layer classes. On the other hand, user requests are mapped to controller actions and controller actions should be mapped to business processes orchestrated in the application layer. However, it is acceptable for you to use a custom mapping of methods onto classes if that helps you achieve a higher level of reusability, especially when multiple presentation front ends are involved.

Orchestration of business processes

The primary purpose of the application layer is abstracting business processes as users perceive them and mapping those processes to the hidden and protected assets of the application's back end. In an e-commerce system, for example, the user expects a shopping cart, but the physical data model might have no entity like the shopping cart. The application layer sits in between the presentation and the back end and performs any necessary transformation.

Accepting an order is typically a multistep workflow and never a single step. If it's a single step, you might not find any benefit in passing through the application layer. Otherwise, the application layer helps immensely to keep workflows distinct from business rules and domain-specific processes. To better understand the difference between application logic and domain logic, consider the following example from a banking scenario.

As a customer, you can talk to a teller and ask to deposit a paper check. Ultimately, some money will be withdrawn from one account and added to another one. But actual processes might be quite different. At a minimum, the teller will go through a process that first places a request to the issuing bank and then adds some money to your account. So you have two operations:

- Cash check from a bank.
- Add money to a bank account.

Both operations are domain-level operations, and both are core tasks of the business domain. The combination of the two, on the other hand, is a workflow that is bound to a specific use-case of the presentation layer—letting users deposit a check. The resulting workflow represents a statement like "Deposit a check" and, depending on the implementation and external services involved, might or might not be a core domain operation. That's an architectural decision after all.

Splitting the business logic into application and domain logic gives you the logical tools to better model the business logic as close as possible to the real world and, more than everything else, close to the user's expectations.

Note These days, the user experience is more important than it once was for the success of any application at any level of complexity. So, to provide an excellent user experience, the golden rule is, "You better have an application layer."

The domain layer

Any software—even the simplest data-entry application—is written against a business domain. Each business domain has its own rules. The number of rules is sometimes close to zero, but as an architect you should always reserve room for a collection of business rules. Finally, each business domain exposes a sort of an application programming interface (API). The way in which the presentation allows end users to interact with such an API—use-cases—determines the application layer.

In a nutshell, the domain layer hosts the entire business logic that is not specific to one or more use-cases. Typically, the domain layer consists of a model (known as the *domain model*) and possibly a family of services (known as *domain services*).

The mythical domain model

Frankly, I find that there's a lot of confusion around the intended role and purpose of the domain model. Abstractly speaking, the domain model is a plain software model that helps render the business domain. The software model can be defined using an object-oriented paradigm (the most common scenario) or any other approach you might find appropriate, such as the functional paradigm. The domain model is the place where you implement the business rules and common, reusable business processes.

Even when you use an object-oriented paradigm, the domain model might or might not be a plain entity-relationship model and might or might not have a one-to-one relationship with the persistence model. The domain model is not strictly related to persistence; the domain model must serve the supreme purpose of implementing business rules. Persistence comes next and is one of the concerns of the infrastructure layer.

In terms of technologies, there's a lot of hype about Entity Framework Code First, which makes it easy to create your classes and then instruct the runtime to create database tables accordingly. This is not domain modeling—it's persistence modeling. As an architect, you should be well aware of logical layers: domain models are a different thing than persistence models. However, the two models can match—and usually match in simpler scenarios—and this brings Entity Framework Code First into play.

In a domain model that focuses on the logic and business rules, inevitably you have classes with factories, methods, and read-only properties. *Factories*, more than constructors, let you express the logic necessary to create new instances. *Methods* are the only way to alter the state of the system according to business tasks and actions. *Properties* are simply a way to read the current state of an instance of the domain model.

An object-oriented domain model is not necessarily an entity relationship model. It can be simply a collection of sparse and loose classes that contain data and store any behavior in separate classes with only methods. A model with entities devoid of any significant behavior—that is, mere data structures—form an *anemic domain model*.

A domain model lives in memory and is, to a great extent, stateless. Yet, some business-relevant actions require database reads and writes. Any interaction with the database—including the canonical example of determining whether a given customer has reached the status of "gold" customer (whatever that means in the business)—should happen outside the domain model. It should occur within the domain layer. This is why, along with a domain model, the domain layer should also feature domain services.

The equally mythical concept of domain services

It's quite simple to explain what a domain model is. Typically, developers nearly instantaneously and completely understand the concept of a software model that renders a business domain. The trouble emerges at a later time when you insist on the ideal separation between domain and persistence.

The point is that a significant part of the business logic is related to the manipulation of information that is persistently saved in some data store or held and controlled by some external web services. In an e-commerce system, to determine whether a customer has reached the status of gold customer, you need to count the amount of orders placed in a given timeframe and compare it to a selected range of products. The output of such a complex calculation is a plain Boolean value that you store in a fresh instance of the Customer domain-model class. Yet you still have a lot of work to do to get that value.

Which module is in charge of that?

Domain service is the umbrella under which a number of helper classes fall. A *domain service* is a class that performs reusable tasks related to the business logic, and it performs them around the classes in the domain model that implement business rules. Classes in the domain services segment have free access to the infrastructure layer, including databases and external services. A domain service, for example, orchestrates repositories—plain classes that perform CRUD operations on entities in the persistence model.

A simple rule for domain services is that you have such a domain service class for any piece of logic you need that requires access to external resources, including databases.

A more pragmatic view of domain modeling

I've probably been too rigorous and abstract in describing the domain layer. Whatever I stated is correct, but in the real world you often apply some degree of simplification. Simplification is never a bad thing, as long as you know exactly which logical layers you are removing for simplicity. If you look at a simplified model, you risk missing some important architectural points that exist even though they might be overkill in that scenario.

There are two terms I need to further explain here in the context of simplifying the architecture of the domain layer: *aggregates* and *repositories*.

Both terms have some DDD heritage. An *aggregate* is a whole formed by combining one or more distinct domain objects that are relevant in the business. It's a logical grouping you apply to simplify the management of the business domain by working with fewer coarse-grained objects. For example, you don't need to have a separate set of functions to deal with the items of an order. Order items make little sense without an order; therefore, orders and order items typically go in the same aggregate. Also, products might be used in the context of an order but, unlike order items, a product might also be acted on outside of orders—for example, when users view the product description before buying.

A *repository* is a component that manages the persistence of a relevant domain object or aggregation of domain objects. You can assign repositories any programming you like, though many developers design these classes around a type *T* being a relevant domain type.

In DDD domain modeling, the concept of an aggregate is a key concept. The vision I'm trying to convey here is more task-oriented and subsequently less centered on entities. The role of an aggregate, therefore, loses importance in the context of the domain layer but remains central in the realm of the infrastructure layer.

In the domain layer, you should focus on classes that express business rules and processes. You should not aim at identifying aggregations of data to persist. Any aggregation you identify should simply descend from your business understanding and modeling. Next, you have the problem of persisting the state of the system.

And when it comes to this, you have at least two options. One option is the classic persistence of the last-known-good-state of the system; the other option is the emerging approach known as *event sourcing*, in which you just save what happened and describe what has happened and any data involved. In the former case, you need aggregates. In the latter case, you might not need aggregates as a way to keep related data together in the description of the event that has happened.

The infrastructure layer

The infrastructure layer is anything related to using concrete technologies, whether it's data persistence (ORM frameworks like Entity Framework), external web services, specific security API, logging, tracing, IoC containers, caching, and more.

The most prominent component of the infrastructure layer is the persistence layer—nothing more than the old-faithful data-access layer, possibly extended to cover a few data sources other than plain relational data stores. The persistence layer knows how to read or save data and is made of repository classes.

Current state storage

If you use the classic approach of storing the current state of the system, you'll need one repository class for each relevant group of entities—this is the aggregate concept. By *group of entities*, I mean entities that always go together like orders and order items.

The structure of a repository can be CRUD-like, meaning you have Save, Delete, and Get methods on a generic type *T* and work with predicates to query ad hoc sections of data. Nothing prevents you from giving your repository a remote procedure call (RPC) style with methods that reflect actions—whether the actions are reads, deletes, or insertions—that serve the business purpose.

I usually summarize this by saying that there's no wrong way to write a repository. Technically, a repository is part of the infrastructure layer. However, from the perspective of simplifying things, a repository can be seen as a domain service and can be exposed up to the application layer so that the application can better orchestrate complex application-level workflows.

Event stores

I would bet that event sourcing will have a dramatic impact on the way we write software. As discussed in Chapter 2, "Selecting the supporting architecture," event sourcing involves using events as the primary data source of the application.

Event sourcing is not necessarily useful for all application. In fact, developers blissfully ignored it for decades. Today, however, more and more domain experts need to track the sequence of events the software can produce. You can't do this with a storage philosophy centered on saving the current state. When events are the primary data source of your application, a few things change and the need for new tools emerges.

Event sourcing has an impact on two aspects: persistence and queries. *Persistence* is characterized by three core operations: insert, update, and delete. In an event-sourcing scenario, insert is nearly the same as in a classic system that persists the current state of entities. The system receives a request and writes a new event to the store. The event contains its own unique identifier (for example, a GUID), a type name or code that identifies the type of the event, a timestamp, and associated information such as the content that makes up the data entity being created. The update exists in another insert in the same container of data entities. The new entry simply indicates the data—which properties has changed, the new value and, if relevant in the business domain, why and how it changed. Once an update has been performed, the data store evolves as in Figure 5-5.

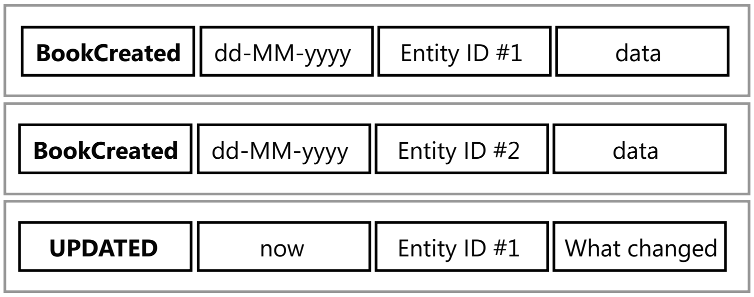

FIGURE 5-5 A new record to indicate the update to the book entity with ID #1

The delete operation works in the same way as an update except that it has different type information.

Making updates in an event-based data store immediately creates a few issues when it comes to queries. How do you get to know if a given record exists or, if it exists, what its current state is? That requires an ad hoc layer for queries that conceptually selects all records with a matching ID and then analyzes the data-set event after the event. For example, it could create a new data entity based on the content of the Created event and then replay all successive steps and return what remains at the end of the stream. This technique is known as *event replay*.

The plain replay of events to rebuild the state might raise some concerns about performance because of the possible huge number of events to process. The problem is easy to understand if you think of the list of events that make up the history of a bank account. As a customer, you probably opened the bank account a few years back and went through hundreds of operations per year. In this context, is it acceptable to process hundreds of events every time you want to see the current balance? The theory of event sourcing has workarounds for this scenario, the most important of which consists of creating snapshots. A *snapshot* is a record that saves the known state of the entity at any given time. In this way, to get the current balance you process only the events recorded since the latest snapshot was taken.

Event sourcing gives architects and domain experts a lot more power to design effective solutions, but for the time being it requires a lot of extra work to create and operate the necessary infrastructure. Event sourcing requires a new family of tools—*event stores*. An event store is another type of database with a public API and a programming language tailor-made for event data items.

Caching layers

Not all data you have in a system changes at the same rate. In light of this, it makes little sense to ask the database server to read unchanged data each and every time a request comes in. At the same time, in a web application requests come in concurrently. Many requests might hit the web server in a second, and many of those concurrent requests might request the same page. Why shouldn't you cache that page or at least the data it consumes?

Very few applications can't survive a second or two of data caching. In a high-traffic site, a second or two can make the difference. So caching, in many situations, has become an additional layer built around ad hoc frameworks (actually, in-memory databases), such as Memcached, ScaleOut, or NCache.

External services

Yet another scenario for the infrastructure layer is when data is accessible only through web services. A good example of this scenario is when the web application lives on top of some customer relationship management (CRM) software or has to consume proprietary company services. The infrastructure layer, in general, is responsible for wrapping external services as appropriate.

In summary, architecturally speaking, these days we really like to think of an infrastructure layer rather than a plain data-access layer that wraps up a relational database. Caching, services, and events are all emerging or consolidated aspects of a system, and they work side by side with plain persistence.

Summary

Software will never be what it was some 10 years ago. Software is destined to be more and more integrated with real life. For this to happen—and it will happen—we have to revisit our architectural principles and change some of them.

In this chapter, I presented a general and generic architecture that can be adapted to any type of software you might write today. It's an evolutionary phase of the classic multitier architecture we grew up with. Although it apparently adds only an extra layer, it has a deep impact on the way we think and do things.

I encourage you to form a clear picture of the purpose of the layered architecture and all its parts before you move further into the book. The issue I see is not that you might miss the point of what a layered architecture represents. That's well-known, at least at a big-picture level. It's the little-known details of the layered architecture—the parts subject to simplification—that represent the sore point and the aspects of the architecture I recommend you spend some time on. Even spending time to decide you don't need those parts is more productive than ignoring them.

Index

Symbols

/ (forward slash), 139, 141

A

abstraction, 226
Accept-* HTTP headers, 389
AcceptVerbs, 148
Access-Control-Allow-Methods, 346
Access-Control-Allow-Origin, 345
access tokens, 349
ACID consistency, 35
Action, 257–258
action filters, 295
 HandleError, 238
 xxxConfig, 222
action methods, 150–151
 producing results, 158–168
ActionName, 148, 163, 250
ActionResult, 80, 151, 158–159, 338, 340
action result classes, 160
actions
 mapping to HTTP verbs, 148–150
 mapping to methods, 147–148
active users, architecture design, 51
ActiveX, 351
adaptiveness, 365
adding services, 125
ad hoc event stores, 44
ad hoc frameworks, 84
ad hoc infrastructure, 39
ad hoc mobile sites, 393
ad hoc model binding, 153–158
AdminController, 288
ADO.NET, 31
 calls, 146
 data access, 14
 data-access layer, 214
 persistence, 222
 repository class, 227
advertised_device_os, 395
advertising, optimizing with WURFL.JS, 388
aggregated objects, 23
aggregate root, 29
aggregates, 18, 28–30, 96–97, 229
 Customer, 31
 distinct stacks, 34
 replaying events, 45
Ajax, 104, 351
 $.ajax function, 353–355
 $.ajax method, 344
 caching, 356
 $.get, 355
 $.getJSON, 355
 JQuery snippets, 167
 load, 355
 paging data, 266–267
 placing calls, 355
 $.post, 355
 shorthand functions, 355
 user experience, 337
$.ajax function, 353–355
$.ajax method, 344
Ajax POST, 301
Ajax requests, CORS, 350–351
alert widgets, 294
all-encompassing context, 13
AllowAnonymous, 242
always, jqXHR, 354
Amazon, device detection, 393–394
analytics, enhancing with WURFL.JS, 388
AND, 367–368
Android
 adapting views for, 381–389
 CSS media queries level 4 standard, 368
 date input field, 383

Android (*continued*)
- PICTURE, 376
- tel input field, 382–383

anemic domain model, 95
AngularJS, 77, 84, 104, 358
anticorruption layer, 16
ApiController, 80, 349
APIs (application programming interfaces), 95
- private setters, 27

append-only data stores, 44
Apple iPad, device detection, 397
Apple iPhone
- media types, 367
- Responsive Web Design (RWD), 366

ApplicationDbContext, 226
application hosts, 114
application/json MIME, 338
application layer, 20, 92–94
- controllers, 147
- data model, 220
- external services, 32
- repository classes, 228
- worker-service class, 225

application-layer methods, 58
application logic, 20–21, 50, 65, 89
- design of, 9
- difference from domain logic, 21
- gray areas, 20
- patterns of, 21

application programming interfaces (APIs), 27, 95
application routes, 138–145
applications, 90, 227
- mapping ASP.NET MVC, 218–219
- multitenant, 223

application services, 225–227
application settings, 126–127
Application_Start, 140
application startup, 220–224
appSettings, 126–127
App_Start, 140
architect role, 51–52
architecture
- application-layer, 58
- ASP.NET Core, 122
- ASP.NET MVC, 77, 80
- ASP.NET Web API, 80
- bounded context, 14
- controller, 58
- data model, 214
- DDD changes to, 50
- dependencies in ASP.NET Core, 68
- gray area, 89
- ImageEngine platform, 390
- input model, 218
- layered, 65–66
- onion diagram 1990s, 48
- onion diagram today, 49
- three-tier, 65, 88
- top-down approach, 48
- user vs. developer perspective, 66
- view model, 219–220

architecture design,
- active users, 51
- choosing Bootstrap, 175
- passive users, 51
- Table Module, 23
- Transaction Script, 22

AreaMasterLocationFormats, 162
AreaPartialViewLocationFormats, 162
areas, 145
- ASP.NET MVC, 220
- configuration, 222
- registering for application startup, 220

AreaViewLocationFormats, 162
art direction, 375, 391
Asimov, Isaac, 171
aspect ratio, 367
ASP.NET
- Application Hosting, 223–224
- binding layer login form, 280
- diagram, 67–68
- feature-complete, 105–106
- Identity, 244
- middleware components, 117
- Responsive Web Design (RWD), 366
- SignalR, 104, 359–361
- state of, 108
- URL routing HTTP module, 143
- virtual path provider, 161–163
- Web Forms, 105

ASP.NET 1.x, 115
ASP.NET Core, 64, 66–71
- impact of, 112
- new functions, 106
- reasons to choose, 69–71
- runtime, 113–118

ASP.NET Core 1.0, 346
ASP.NET MVC, 14, 58, 64, 76–79, 105, 111, 118
- vs. ASP.NET Web API, 80–82
- display modes, 398
- exposing JSON content, 338
- input model, 218

ASP.NET MVC (*continued*)
  login form, 281–282
  paging data, 263–264
  planning projects, 217–230
  registering routes, 221
  URL routing HTTP module, 137–138
  weak points, 78–79
  Web API, 347–348
  vs. Web Forms, 133
  xxxFor, 291
ASP.NET MVC 3, 105
ASP.NET Web API, 64–65
  vs. ASP.NET MVC, 80–82
  purpose of, 346–347
  REST, 82
  role of, 79–83
ASP.NET Web Forms, 64, 72–75, 118, 133, 221, 244. *See also* Web Forms
ASPX, 161
ASPX pages, 133
async, placing scripts for speed, 252
async/await, 129
asynchronous operations, 168
attribute routing, 143–145
attributes
  About, 148
  AcceptVerbs, 148–150
  ActionName, 148, 163
  AllowAnonymous, 242
  Authorize, 242
  Compare, 287
  contentedittable, 306
  CustomValidation, 287
  data annotations, 287–288
  data-dismiss, 297–298
  data-id, 273
  Flags, 148
  float, 261
  HandleError, 238
  HttpGet, 148
  HttpPost, 148
  HttpPut, 148
  media, 367
  NonAction, 147–148
  OutputCache, 280
  Range, 287
  RegularExpression, 287
  Remote, 287
  Required, 287
  route, 140
  Route, 144
  RouteArea, 144
  RoutePrefix, 144
  StringLength, 287
  type, 306
  ValidationAttribute, 287
  ViewResult, 163
audit logs, 44
authentication, 128–129
  OAuth, 349–350
authorization, 128
  by host, 348
Authorize attribute, 128, 242, 348
autocompletion, 307–310
  Bootstrap Typeahead, 307–308
  simulate a drop-down list, 310
Auto-growth, 233
autosave, 311
Axure, 56
Azure SQL Database, 231

B

back ends
  end point, 93–94
  from prototype, 58–59
backoffice, 12
Backup, 232
Balsamiq, 54, 56
Base64 text stream, IMG element, 374
Basic authentication, 348–349
basic query language, 367–368
behavior, 25
  value objects, 26
binding,
  DOM, 356–359
  posting from login form, 281
  primitive types, 154–155
  webHttpBinding, 346
Bin folder, 217
blog engine, multitenant applications, 223
Bloodhound, Typeahead, 308
BODY, Bootstrap, 376
Boolean model binding, 156
Boolean value, 96
Bootstrap
  codes, 177
  data-placement, 272
  date-picker, 305–306
  drill-down view, 276
  fluid grid schema, 176–177
  fonts in RWD, 376–378

Bootstrap (*continued*)
  at a glance, 171–176
  glyph icons, 251
  grid system, 176–178
  hidden-xx, 181–182
  Internet Explorer Compatibility, 175
  line height, 376
  media queries, 369
  modal, 296
  offsetting columns, 183
  orientation, 379
  overall template, 176–177
  popovers in action, 274
  popover view, 271–274
  progress bar, 363
  pulling columns, 183
  putting in perspective, 175–176
  rendering data in logical groups, 261–263
  Responsive HTML templates, 392
  Responsive Web Design (RWD), 366
  screen-based rendering, 178–183
  setting up, 174–175
  tool tips, 272
  visible-xx, 181
  WURFL.JS, 389
Bootstrap 3, 378
Bootstrap 3.x, 376
Bootstrap 4, 376
  changes in, 378
  Internet Explorer 8, 378
  REM option, 378
Bootstrap Datepicker, 305–306
Bootstrap Typeahead, 307–308
bottom-up approach, 48–50
bounded context, 4, 10–14
  architecture, 14
  backoffice, 12
  conformist relationship, 16
  customer/supplier relationship, 16
  DDD, 226
  detecting, 11–12
  discovering, 10
  distinct, 13–14
  event storming, 17–18
  implementing, 12–14
  online store application example, 11–12
  partner relationship, 16
  relationships between, 15–16
  subdomains, 11
  two-pizza rule, 17
bounded text
  anticorruption layer, 16
  shared kernel, 16
Bower file, 119–120
bower.json, 119–120
BPM (Business Process Management), 4
Brandolini, Alberto, 17
breakpoints, 177
  Bootstrap codes, 179–180
  screen size, 179–180
brittle model, 13
brownfield migration, 112
browser personalization, WURFL.JS, 389
browsers
  CORS-aware, 345
  date pickers, 305
  new input types, 382
  responsive images, 374–376
Bundle
  CSS, 233
  ScriptBundle, 236–237
  StyleBundle, 236–237
BundleCollection, 234
BundleConfig, 231
BundleFileSetOrdering, 234
BundleTable, EnableOptimizations, 232–233
bundling
  browser activity, 234
  LESS, 174
  optimization, 232–233
  script files, 233–234
  working with, 231–235
  xxxConfig, 222
buses
  message-based formulation, 39
  saga, 42
  xxxConfig, 222
business domain, 4
  modeling, 223
  repositories, 229
business layer, 87
  controller class, 146
business logic, 19, 88–89
  DDD changes in, 50
  Domain Model, 23
  external services, 32
  legacy code, 32
  Table Module, 23
  Transaction Script, 22
business logic layer, 65
  domain model, 216

business model, ASP.NET Core, 107
business processes, 88
  orchestration of, 94
Business Process Management (BPM), 4
business rules, 26–28
  aggregates, 30
  anemic entity class, 26–27
  behavior-rich entity classes, 27–28
  domain model, 216
  private setters, 27
button, command, 40

C

C#, 9, 24
  input forms, 284
  in Razor, 166
cache
  data model, 220
  debug=true, 388
caching, 8
  Ajax, 356
  jQuery, 299, 356
  paging data, 266
caching layers, 99
calling back, 362
canonical data stores, 50
canonical layered architecture, 33–34
canonical recovery, 241
cascading options, 31
Cascading Style Sheets. *See* CSS
Cassandra, 231
catch, 241
catch-all parameters, 143
CDN (content delivery network), 233
  ImageEngine platform, 390
cell phones, Responsive Web Design (RWD), 366
child objects, repositories, 31
Chrome,
  date pickers, 305
  PICTURE, 376
Churchill, Winston, 87
classes, 24
  clearfix, 261
  entity, 25
  input model class, 288
  naming conventions, 6
  size-specific, 181–182
  worker-service, 225–227
classic web model, 104
class names, 177
clearfix, 181, 261
click, popover, 272
Client Hints, 389
clients
  broadcasting changes, 364
  calling back, 362
  device detection, 388–389
Clients.All, 365
client-side device detection, 386–389
  and responsive pages, 388–389
  user-agent sniffing, 386
  WURFL.JS, 387–388
client-side enhancements, 392–393
client-side solutions, 92
client-side validation,
  HTML5, 382
  input form, 301–303
  jQuery, 303, 382
cloud, hosting with, 111
cloud API, NuGet, WURFL framework, 394
CloudClientManager, 398
cloud-optimized frameworks, 107
cloud storage, 222
CLR (Common Language Runtime), 67
  as DNX layer, 114
  host, 114
code
  Bootstrap, 177
  breakpoints, 179–180
  Razor, 164–167
code assistants, ReSharper, 9
Code First, 215, 226–227
col CSS, 370
collections, model binding, 157–158
columns,
  offsetting in Bootstrap, 183
  pulling in Bootstrap, 183
Command and Query Separation (CQS), 33–39
CommandDatabase object, 39
command processor, 39
Command Query Responsibility Segregation. *See* CQRS (Command Query Responsibility Segregation)
CommandResponse, 293
commands, 40
  button, 40
  message-based formulation, 39
  saga, 42

command stack
  mapping ASP.NET MVC, 219
  Match example, 36–39
  repository classes, 228
common base class, view model, 248
Common Language Runtime (CLR), 67–68
communication channel, sharing, 361
Compare, 287
complete, 385
complete_device_name, 387, 395
conditional style sheets. *See also* CSS
  CSS media queries, 368–369
configuration,
  areas, 222
  blog engine, 223
  repositories, 229
  routes, 222
  understanding, 223–224
Configure, 116, 123, 131, 223–224
ConfigureServices, 123, 224
confirmation messages, 284–285
conformist relationship, 16
  anticorruption layer, 16
Confucius, 133
$.connection, 360
connection strings, 228
  repositories, 229
container class, model binding, 156
container-fluid class, 178
container.less, 173
containers
  Bootstrap, 177–179
  DIV elements, 178
  horizontal scrolling, 268
content
  downloaded to any user agent, 372
  horizontal scrolling, 268
  modal, 296
  negotiating, 341–342
  negotiation, 81
  pushing, 359–365
  Web API, 347
content delivery network (CDN), 233
  ImageEngine platform, 390
contentedittable, 306
Content folder, 228
ContentResult, 159, 168
context mapping, 4, 15
controller classes,
  exploring, 145–151
  statelessness, 146
  writing, 147–151
controller methods
  example, 91–92
  JSON, 338–341
  POST, 295
controllers, 61, 76, 225
  ApiController, 80
  area, 145
  areas of ASP.NET MVC, 220
  ASP.NET Core 1.0, 346
  asynchronous operations, 168
  error handling, 238–241
  granularity, 145–146
  mapping ASP.NET MVC, 219
  partial view, 228–229
  RouteData, 153
  testability of, 147
  URL routing HTTP module, 137–138
  view engine structure, 161
  view model, 92
controller-specific folder, 256
cookie authentication, 128–129
cookies
  input data, 152–153
  TempData, 286
Cordova, modal windows, 304
CoreCLR, 112
CoreCLR.DNX, 113
CORS (Cross-Origin Resource Sharing), 342–346, 350–351
  enabling, 344–346
  status code, 353
CQRS (Command Query Responsibility Segregation), 14, 24, 33, 65, 79, 219, 285
  event persistence, 43
  mapping ASP.NET MVC, 219
  Match example, 36–39
  message-based architecture, 42
  message-based formulation, 39–42
  polyglot persistence, 231
  repository classes, 228
CQS (Command and Query Separation), 33–39
Create, Read, Update, Delete (CRUD), 12, 75, 78
cross-entity business logic, 30–31
Cross-Origin Resource Sharing. *See* CORS (Cross-Origin Resource Sharing)
cross-platform APIs, WURFL framework, 387
cross-property validation, 292–293
  the next view, 293–295

CRUD (Create, Read, Update, Delete), 12, 75, 78
CSS
  orientation, 378
  show/hide, 372
  style sheets, 393
CSS 3
  AND, 367–368
  aspect-ratio, 367
  core properties, 367
  device-aspect-ratio, 367
  device-height, 367
  device-width, 367
  height, 367
  media attribute, 367–368
  Modernizr, 385
  OR, 367–368
  orientation, 367
  width, 367
CSS bundle, 233
CSS classes, 228
CSS files,
  bundling, 231–232
  LESS, 174
  minification, 236–237
CSS float, 181
CSS media queries
  conditional style sheets, 368–369
  fluid grid, 366–367
  media types, 366–367
  Responsive Web Design (RWD), 366
CSS media queries level 4 standard, 368
  Android, 368
  features added, 368
  hover, 368
  pointer, 368
  resolution, 368
  scripting, 368
CssMinify, 236–237
current state storage, 97–98
Customer aggregate, 29, 31
CustomerRepository, 31
customer/supplier relationship, 16
customization, comparison of tools, 233
custom transformers, 237
CustomValidation, 287, 292
custom view engines, 229–230
  VirtualPathProviderViewEngine, 229

D

DAL (data-access layer), 214
data
  annotation attributes for validation, 287
  grid view, 258–259
  paging, 263–268
  passing to partial view, 256
  posting from a modal input form, 299–301
  processing posted, 281–283, 290–291
  rendering in logical groups, 261–263
data access, 131
data-access API, 229
data-access layer (DAL), 214
data-access logic, 228
data annotations
  validation messages, 291
  validation process, 287–288
DataAnnotationsModelValidatorProvider, 287
database-inferred data models, 224
databases
  event store, 99
  impact on model, 5–6
  schema, 126
  segments, 225–226
data-definition language (DDL), 227. *See also* Code First
data-dismiss, 297–298
data-id, 273
data-input type, Modernizr, 385
data items, displaying, 258–271
data-last, on-demand scrolling, 270–271
data layer, 87
data models, 214
  database-inferred, 224
  different flavors, 214–220
  old-fashioned, 221
data-placement, popover, 272
data repository, 387
DataSet, 222
data sources
  events, 43
  event sourcing, 98
data stores, 98
  append-only, 44
DataTable, 222
data-target, 297
data-toggle, 297
data-transfer objects (DTOs), 33
  view model, 247

data-trigger, popover, 272
date pickers, 305–307
  Bootstrap Datepicker, 305–306
dates
  input fields, 305
  new input types, 382
  plain text, 307
DbContext, 38, 225, 227
  repositories, 229
DbSet, 38
DDD (Domain Driven Design), 4, 88
  changes in, 50
  DDD-bounded contexts, 226
  misconceptions, 4–6
DDL (data-definition language), 227. *See also* Code First
debug=true, 388
default.aspx, 143
DefaultBundleOrderer, 234
default location formats, 162
DefaultModelBinder, 153–154, 157
default values, 140–141
defer, 252
DELETE, 138
delete operations, 99
demos, 54
dependency injection (DI), 225
design,
  strategic, 5
  tactical, 5
desktop browsers, 382
desktop view, modal windows, 304
detail view, 271–276
  drill-down views, 274–276
  modal dialog boxes, 276
detecting authenticated users, 129
development models, 107–108
device-aspect-ratios, 367
device detection, 392
  WURFL.JS, 388
  xxxConfig, 222
device-friendly images, 390–392
device-friendly views, 392–399
device-height, 367
device independence, 371–373
devices, adapting views for, 381–389
device-width, 367
DI (dependency injection), 225
direct binding, 356–357
DisableCors, 350
displayKey, 308
display modes,
  ASP.NET MVC, 398
  device detection, 397
distinct databases, 34–35
distinct stacks, 33–34
DIV
  modal-content, 297
  modal-dialog, 297
DLL files, 217
DNVM (.NET Version Manager), 115
DNX (.NET Execution Environment), 113–116
  hosting web applications, 115–116
  Kestrel, 122
  layers of, 114
  understanding configuration, 223–224
dnx.exe utility, 114
DNX runtime environment, 70
document NoSQL data stores, 50
document.write, 252
DOM, binding data, 356–359
domain, 90
  mapping ASP.NET MVC, 218–219
domain class, 288
Domain-Driven Design (DDD), 4, 88–89
  domain model, 216–218
  misconceptions, 4–6
Domain-Driven Design: Tackling Complexity in the Heart of Software (Evans), 4, 24
domain event storming, 17
domain layer, 30, 50, 95–97
  data model, 220
  domain model, 216
domain logic, 50, 65, 89
  difference from application logic, 21
  examining, 21–22
  persistence, 22
domain model, 5, 23–24, 95––97, 214, 216–218, 227–228
  aggregates, 28–30, 96–97
  architecture, 14
  bounded context, 12
  business rules, 26–28
  canonical layered architecture, 33–34
  data model, 220
  distinct stacks, 33–34
  event store, 44
  Mercator, 32
  repositories, 96–97
  simple exercise, 216–218
  single model, 224

domain model (*continued*)
  sports match example, 26–27
  value types, 26
Domain Model pattern, 5
domain services, 30, 95.96
  cross-entity business logic, 30–31
  in a Domain Model, 23
  repositories, 31
  the role of, 30–32
domain-specific terms, 6–8
done, jqXHR, 354
“Don’t Repeat Yourself” (DRY), 172
downloading to different devices, 372
downstream context, 15
drill-down views, 274–276
drop-down lists
  autocompletion, 307–310
  text box as, 309–310
drop-down menus, 171
DRY (“Don’t Repeat Yourself”), 172
DTOs (data-transfer objects), 33
  view model, 247
dynamic, 165

E

EF Core (Entity Framework Core), 112
Eiffel programming language, 33
Einstein, Albert, 109
ELMAH (Error Logging Modules and Handlers), 118, 241
email input field, 382–383
EmptyResult, 159
EM units, font size, 377
EnableCors, 350
EnableOptimizations, BundleTable, 232–233
endpoints, 93–94
  JSON, 338–341
Enterprise Library Data Access Application Block, 222
entities, 25
  aggregate roots, 29
  identity, 25
  methods, 25
  persistence, 25
  properties, 25
Entity Framework, 5, 31, 37, 97, 131, 223–228
  Code First, 226–227
  database segments, 225
  persistence model, 215
  polyglot persistence, 231
  read model, 38–39
  repository class, 227
Entity Framework 6, 131
Entity Framework Code First, 95, 226–227
Entity Framework Core (EF Core), 112, 131
EnumDataType, 287
errorContainer, 302
error handling, 238–241
errorLabelContainer, 302
Error Logging Modules and Handlers (ELMAH), 118, 241
error messages,
  displaying login, 281–282
  input form, 292
  status code, 352–353
ES (event sourcing), 24, 43–45
Evans, Eric, 3–4, 24, 89
  Domain-Driven Design: Tackling Complexity in the Heart of Software, 4
event-based data stores, 44–45, 50, 98–99, 223
event handlers
  Application_Start, 140
  saga, 42
event projection, 43
event replay, 99
events, 41, 43
  immutable, 43
  orientation change, 379
  replaying, 45
  replaying, 45
  resize, 379
  saga, 42
  sequence diagram, 44
  trend in software, 50
event sourcing (ES), 24, 43–45, 97–99
Event Store project, 45
event storming, 17–18
  modeling space, 17
exception handling tool, 118
Exceptionless, 241
exceptions
  collecting, 241
  NotSupported, 143
  route, 239–240
  route not found, 238
ExecuteResult, 159
extension methods,
  in languages, 9
  Map, 117
  MapWhen, 117

extension methods *(continued)*
  UseExceptionHandler, 117
  UseStaticFiles, 117
external services, 32
  storing data, 99

F

F#, 24
F5 effect, 290
F5 key, 284–285
Facebook,
  OAuth, 349–350
  polyglot persistence, 230
  social authentication, 244–245
  WURFL framework, 387
FacebookAuthenticated, 245
factories, 95
fail, jqXHR, 354
fat-free controllers, 225
favicon, 252
feature detection, 392
  Modernizr, 384–386
  RWD, 384–386
FileContentResult, 159
FileExtensions, 163
FilePathResult, 159
FileSetOrderList, 234
FileStreamResult, 159
Finish, 37
Firefox,
  date pickers, 305
  device independence, 371
  PICTURE, 376
Flags, 148
flexible rendering, 182
float, 261
flow of data, 20
fluid containers, 178
fluid grid, 366–367
folders
  Application, 227
  Content, 228
FontAwesome, 251
fonts
  RWD, 376–378
  in a web application, 228
font size
  EM units, 377
  REM units, 377
  viewport units, 377
formatters, Web API, 347
formatting layer, 81
form_factor, 387
forms, 152–153
  input, 284
  validation, 286–295
Forms authentication, 241
forward slash (/), 141
  URL parameter, 139
Fowler, Martin, 5
  Repository pattern, 227
frameworks
  deciding the best, 64–65
  Web Forms, 64
functional features, 24
functional languages, 24
functional paradigm, 95
f_XXX, 391

G

GAC (Global Assembly Cache), 115
GalaxyBase, 32
Galileo, 18
Gandhi, Mahatma, 63
gated check-ins, 9
geolocation, 384
$.get,
  Ajax, 355
GetHttpHandler, 143
$.getJSON, 344
  Ajax, 355
GetOrderById, 144
GETs, 138, 144, 285
  caching, 356
  login form, 280
global.asax, 139, 397
  application startup, 220–224
  bundling, 231–232
  RegisterAllAreas, 221
  registering routes, 221
Global Assembly Cache (GAC), 115
globally accessible data, 222
glossary, 7
  choosing terms, 7–8
  and international teams, 9
  sharing, 8
glyph icons, 251
God antipattern, 32
gold customer, 30

Google, WURFL framework, 387
Google Maps, 32
graceful degradation, 373
granularity,
  controller, 145–146
  user interface, 146
graph database, polyglot persistence, 231
graphical user interface (GUI), 52–53
gray area, 89
grid system, 371
  Bootstrap, 176–178
  example, 369–370
  media queries, 369–371
grid view, 258–259
GridView, 263–264
group of entities, 97
Grunt, 120
GUI (graphical user interface), 52–53
GUID, saga, 42
Gulp, 174
Gulp file, 120–121
Guriddo jqGrid, 263–264
GZIP, 231

H

handheld media, 366–367. *See also* media types
HandleError, 238
handlers, 41–42
  application startup, 220
  command processor, 40
Hawking, Stephen, 217
headers, Accept-* HTTP, 389
HEAD section, 251, 254
height,
  CSS 3 media queries, 367
  IMG element, 374
helper methods, 338–339
  ASP.NET MVC xxxFor, 291
hidden-xx, 181–182
high scalability, 43
h_NNN, 391
home page, 125
home/test/*, HTTP handler, 136
hosting layer, ASP.NET, 115–116
host pages, client-side validation, 302
hover
  CSS media queries level 4 standard, 368
  popover, 272
HTML, 171
  markup, 160–161
  TextBoxFor, 289
  Web API, 346–350
HTML5, 75
  client-side validation, 382
  date pickers, 305–307
  IMG element, 374
  INPUT, 382
  for mobile devices, 382–384
  Modernizr, 385
  support for browsers, 305–307
Html.CheckBox, 280
HTML FORM, 310
HTML forms, view model, 219–220
HTML helpers, 254
  Html.Raw helper method, 166
  input form, 289
Html.PagedListPager, 265
Html.ValidationMessageFor, 291
Html.ValidationSummary, 291
HTML view, 247–258
HTTP, 346
HTTP/2, 109
HTTP 200, 352–353
HTTP 404 error, 137–138, 143, 239, 352–353
HTTP 500, 352–353
HttpApplication, 222
  application startup, 220
HttpContext, 398
HTTP endpoints, 65
HTTP errors, 238–241
HttpGet, 148
HTTP GET, 80–81
HTTP handlers, 130
  defining behavior, 135–136
  exposing JSON content, 338
  home/test/*, 136
  invoking, 136–137
  MvcEmule.HomeController, 136
HTTP modules, 130
  URL routing, 137–138
HttpNotFoundResult, 159
HTTP pipeline, 116–118
  configuring, 123–124
HttpPost, 148
HttpPut, 148
HTTP requests, 133
  open, 351–352
  placing, 351–352
  send, 351–352
HTTP responses processing, 352
HttpStatusCodeResult, 159

HttpUnauthorizedResult, 159
HTTP verbs, 138
  mapping actions to, 148–150
Hub class, 365
hubs
  internals, 362–363
  ProgressHub, 362
hybrid SPA, 84, 104

I

IActionResult, 17
IApplicationBuilder, 117, 124, 223–224
IBundleOrderer, 234
IBundleTransform, 236–237
icon, website, 252
ID, saga, 42
id attribute, 281
identity, 25
  value objects, 26
ID.Next, 36
IFRAME, 228–229
ignore lists, 235
  BundleCollection, 234–235
IgnoreRoute, 143
IHostingEnvironment, 124
IHTTPHandler, 135
IIS (Internet Information Services), 67–68, 347
IIS 7, 106, 110, 345
IISPlatformHandler, 122
ILoggerFactory, 124
ImageEngine, 390, 393
  account with, 391
ImageEngine Lite, 376
image-resizing tool, 390
images
  client device detection, 388–389
  device-friendly, 390–392
  resizing automatically, 391–392
  responsive, 374–376
  in a web application, 228
IMG elements, 374
  JSONP, 342
immutable events, 43
immutable types, 25
imports, LESS, 173
Include, 232
IncludeDirectory, 232
incoming requests, 134–145
Index defragmentation, 232
index.smartphone.cshtml, 398
Indigo Studio, 57
infrastructure, 90
  mapping ASP.NET MVC, 218–219
infrastructure layer, 30, 97–100
  data model, 220
  domain model, 216
@inject, 127
inline editing, 311
INPUT
  HTML5, 382
  login form, 280
  test input type, 385
input data
  action methods, 150–151
  new types for mobile devices, 382–383
  processing, 152–158
  route, 153
input forms, 284–304
  error messages, 292
  modal, 296–304
  presenting, 289–290
  tips for large, 310–311
input model class, 153
  decorating, 288
input models, 90–92, 218, 227–228
  data model, 220
Instagram, 244
instances, 248
Int32 types, 25
integrated pipeline, 110
interactivity, 365
internal state, Match example, 37
Internet Explorer
  ActiveX, 351
  Bootstrap, 175
  date pickers, 305
  device independence, 371
  input types, 383
  site icon, 252
Internet Explorer 7, 351
Internet Explorer 8
  Bootstrap, 175
  Bootstrap 4, 378
Internet Information Services. *See under* IIS (Internet Information Services)
inversion of control (IoC)
  data model, 220
  xxxConfig, 222
iPad, Responsive Web Design (RWD), 366

iPhone. *See also* Apple iPhone
- adapting views for, 381–389
- Responsive Web Design (RWD), 366

IQueryable objects, 39
IRouteHandler, 142
IsAjaxRequest, 301
IsDesktop, 398
IsGold method, 31
IsInProgress, 27
IsKnownOrigin, 345
is_mobile, 387
IsSmartphone, 398
is_smartphone, 395
is_tablet, 395
IsTeamLeading, 27
is_wireless_device, 395
Item indexer, 152
IViewEngine, 161

J

Java, 24
JavaScript, 68, 109
- action methods with, 151
- for adapting views, 381–389
- ASP.NET MVC, 77
- in Bootstrap, 171
- minification, 236–237
- Paginator, 267
- paging data, 263–264
- placing scripts, 252
- posting from a modal input form, 299–301
- Save button, 310
- template libraries, 357–359

JavaScriptResult, 159
JavaScriptSerializer, 338, 340
jQuery, 77, 167
- $.ajax function, 353–355
- ASP.NET SignalR, 360–361
- caching, 299, 356
- client-side validation, 301, 303, 382
- drill-down view, 276
- file bundling, 234–235
- jqXHR, 353
- library, 344
- Mobile websites, 304
- plugins in Bootstrap, 171
- tools, 353–356
- validation, 299–302

jQuery UI, 305
- date-picker plugin, 385

JsMinify, 236–237
JSON, 81
- action methods with, 151
- endpoints, 338–341
- exposing content, 338–346
- hijacking, 339–340
- negotiating content, 341–342
- response data, 339
- returning, 167

JSON files, 69–70, 126
JSON.NET, 338
JSONP (JSON with Padding), 342–344
jsonpCallback, 344
JsonpResult, 343
JsonRequestBehavior, 339–340
JsonResult, 159, 338, 340–341, 343
JustInMind, 57

K

Kennedy, John F., 247
Kestrel, 115
Kestrel, 122
King, Martin Luther, 103
Knockout, 77, 85
- template libraries, 357

Kotlin, 9

L

lambda code, 117
landscape, 379
languages, extension methods, 9
layered architecture, 65–66, 87, 89, 93
- mapping, 218–219

layouts,
- creating presentation, 228–230
- responsive, 176–183

lead, fonts in RWD, 376
leader of discussion, 18
legacy code, 32, 106
legacy data model, 221–223
LESS, Bootstrap, 172–174
LessTransform, 237
line height, Bootstrap, 376
LINK, 367
LinkedIn,
- on-demand scrolling, 269
- social authentication, 244

LINQ IQueryable, 226
LINQ-to-DataSet, 222–223

LinQ-to-Entities, 38
Linux, 68
  DNX runtime environment, 70
listeners, 41–42
lists, drill-down views, 274–275
ListView, paging data, 263–264
load, Ajax, 355
locale, 142
localization, xxxConfig, 222
logging, 8
logical groups, 261–263
Login, 243
  Facebook authentication, 245
login forms, 280
  common forms, 279–283
  displaying errors, 281–282
  posting from, 281
  processing posted data, 281–283
  system feedback, 283
LoginInputModel, 91, 129, 219
  posting from login form, 281
LoginViewModel, 219
LoveGermanShepherds, 148

M

Mac, 68
  DNX runtime environment, 70
Main, 122–123
maintenance operations, tools, 232
Malcolm X, 381
managed entry point, 114
Mandela, Nelson, 337
Map, 117
mapping
  actions to HTTP verbs, 148–150
  actions to methods, 147–148
  layers to models, 228
  of methods, 94
  model-binding, 153
MapRoute, 140, 221
MapWhen, 117
MapXxx, 117
Marcotte, Ethan, Responsive Web Design (RWD), 366
MasterLocationFormats, 162, 229
master pages, layout, 250
Match class, 26, 37–39
matchMedia, 379
MatchState property, 27
matrix
  Bootstrap, 176–177
  container in Bootstrap, 176–177
MaxJsonLength, 340
media, 366
  LINK, 367
  print, 366
  screen, 366
media queries, 393
  Bootstrap, 369
  grid system, 369–371
  orientation, 378
media types
  CSS media queries, 366–367
  handheld, 366–367
Memcashed, 99
memory footprint,
  ASP.NET Core, 106
  cost of, 110
Menu, common base class, 248
Mercator map, 32
message-based CQRS architecture, 42
message-based formulation, 39–42
methods, 25, 95
  application-layer, 58
  Configure, 123–124
  ConfigureServices, 123
  Get, 144
  GetHttpHandler, 143
  Include, 232
  IncludeDirectory, 232
  Login, 243
  LoveGermanShepherds, 148
  Main, 122–123
  mapping actions to, 147–148
  MapRoute, 140
  MapXxx, 117
  naming conventions, 6
  OnConnected, 365
  OnDisconnected, 365
  OnReconnected, 365
  Partial, 256
  private setters, 27
  RegisterRoutes, 140
  Render, 233
  RenderBody, 253–254
  RenderPartial, 256
  RouteConfig, 140
  Run, 117
  saga, 42
  ToString, 168

methods (*continued*)
  UseXxx, 117
  value objects, 26
  worker-service class, 225
Meyer, Bertrand, 33
Microsoft ActiveX. *See* ActiveX
Microsoft Ajax. *See* Ajax
Microsoft Application Insights, 241
Microsoft ASP.NET Identity, 242–244
Microsoft ASP.NET Web Optimization, 231
  WebGrease, 236–237
Microsoft Azure SQL Database. *See* Azure SQL Database
Microsoft C#, 73
Microsoft C# compiler, 150
Microsoft Cordova, 304
Microsoft Edge, 305
Microsoft Entity Framework. *See* Entity Framework
Microsoft Internet Explorer. *See* Internet Explorer
Microsoft Internet Information Services (IIS). *See under* IIS (Internet Information Services)
Microsoft Json. *See* JSON
Microsoft Message Queuing (MSMQ), 346
Microsoft .NET Framework, 110
  ASP.NET Core, 66
  data-access layer, 214
  JSON data, 338
Microsoft NuGet. *See* NuGet
Microsoft Office Excel, 8
Microsoft OneDrive, 8
Microsoft ReSharper, 9
Microsoft Silverlight, 75, 104
Microsoft SQL Server, 222
  persistence model, 215
Microsoft stack,
  ASP.NET Core, 66
  ASP.NET MVC, 76–79
  web solutions, 64–66
Microsoft Visual Basic 6, 221
Microsoft Visual Basic user interface, 49
Microsoft Visual Studio 2015. *See* Visual Studio 2015
Microsoft Web Essentials, 174
Microsoft Web Forms. *See* Web Forms
middleware, 130–131
  ASP.NET, 117
minification, 174, 236–237
mirroring, 18
mixins, LESS, 172
mobile applications, 346–350
mobile content, 392–394
mobile-first, 373
mobile-last, 373
mobile-only-use-case, 384
mobile sites
  client-side device detection, 386
  redirecting to, 393
mobile web, 90
mockups, 53
modal, 297–298
modal-content, 297
modal dialog boxes, 171, 276, 297
modal forms, 296–304
  configuring, 296–297
  initializing, 298–299
  posting, 299–301
  pros and cons, 303–304
  tablets, 393
modal updates, 364
modal windows,
  cache setting, 356
  desktop view, 304
  improving user experience, 304–311
@model, 164
model binders, 152, 225
model binding, 91, 153–158
  binding complex types, 156
  Boolean, 156
  collections, 157–158
  default, 153–154
  optional values, 155–156
modeling, 18
modeling space, event storming, 17
models, 61, 225
  adding in ASP.NET MVC, 227–228
  areas of ASP.NET MVC, 220
  for the business domain, 50
  classic web, 104
  different programming, 111
  persistence, 5–6
ModelState dictionary, 290
Model-View-Controller (MVC), 61–62, 134
Model-View-Presenter (MVP), 61–62
Model-View-ViewModel (MVVM), 61–62
@Model.XXX, 164
Modernizr
  bundling files, 234–235
  data-input type, 385
  date pickers, 305
  feature detection, 384–386
  polyfill, 376
  what it can do, 384–386
modern webpages, 171

Mono, 71
monoliths, 87
mouse pointer, 393
Moustache, 85
m-site, 393
MSMQ (Microsoft Message Queuing), 346
multiresolution views, 180–181
multitenant applications, 223
mustache.js, 357
  sample template, 359
MVC (Model-View-Controller), 61–62, 67–68, 134
MvcEmuleHandler, 134
MvcEmule.HomeController, 136
MVC triad, 61
MVP (Model-View-Presenter), 61–62
MVVM (Model-View-ViewModel), 61–62
m_XXX, 391

N

name, route, 140
name attributes, 281
named pipes, 346
naming conventions,
  event, 41
  ubiquitous language, 6
native process, 114
navigator, user-agent sniffing, 386
NCache, 99
Neo4j, 231
.NET CLR, 70
.NET Core, 67–68, 71, 107, 111
.NET Execution Environment, 113, 223–224
.NET Framework, 30, 67–68, 110
  ADO.NET., 222
  ASP.NET Core, 66, 70
  data-access layer, 214
  definition, 112
  JSON data, 338
.NET Version Manager (DNVM), 115
.NET X-platform, 67–68
New, 288
NHibernate, 225, 227
NonAction, 148
non-desktop devices, 373–379
NoSQL, 31
NoSQL, 223
NotSupported exception, 143
NuGet, 115, 119, 121, 231
  ASP.NET SignalR, 360–361
  Bootstrap, 174
  cloud API, 394
  ELMAH, 241
  http://github.com/eternicode/bootstrap-datepicker, 305–306
  jQuery Validation, 302
  JSON.NET, 338
  LessTransform, 237
  PagedList, 264
  paging data, 263–268
  Typeahead, 307–308
  WURLF framework, 394
null, 156
numbers, input types, 382

O

OAuth, 244, 349–350
object models, distinct stacks, 34
object-oriented domain model, 24–26
object-oriented paradigm, 95
Object/Relational Modeling (O/RM), 31
obsolescence, 106
OnActionExecuted, 295
OnActionExecuting, 295
OnConnected, 365
OnDisconnected, 365
OnException, 238
online users, counting, 365
onreadystatechange, 352
OnReconnected, 365
Open Web Interface for .NET (OWIN), 67, 112
Opera
  date pickers, 305
  PICTURE, 376
optimization, 109
  bundling, 232–233
optional values, model binding, 155–156
OR, CSS 3, 367–368
OrderController, 93
OrderCreated, 17, 41
OrderDetails, 28–29
orderers, BundleCollection, 234–235
orderId, 144
OrderItem, 29
OrderModule, 93
OrderRepository, 31, 93
Orders, 28
OrderService, 93
orientation, 368–369, 378
  Bootstrap, 379
  changes, 379

origin, 345
O/RM (Object/Relational Modeling), 31, 37, 223
O/RM helper object, 224
output caching, 280
OWIN (Open Web Interface for .NET), 67–68, 78, 112, 128
  ASP.NET SignalR, 360–361

P

package diagnostics, 123
package.json, 121
packages, 124
packages.config, 119
page-agnostic URL, 136
PagedList, 264–265
page layout
  in Bootstrap, 179–180
  examining, 250–253
page loading, placing scripts for speed, 252
Paginator, 267
paging data, 263–268
  Ajax, 266–267
  caching, 266
  Guriddo jqGrid, 263–264
  Html.PagedListPager, 265
  selecting a helper package, 263–264
  Telerik, 264
  URL-based, 264–266
parameter binding, manual, 152–153
parameters, catch-all, 143
Partial, 167, 256
partial view, 167, 228–229, 255–256
  grid view, 258–259
  passing data to, 256
  Razor, 229
PartialViewLocationFormats, 162, 229
PartialViewResult, 159
partner relationship, 16
Pascal (Wirth, Niklaus), 48
passive users, 51
password hashing service, 243
pathinfo, 143
patterns
  business logic, 22–24
  Domain Model, 22–24
  Model-View-Controller (MVC), 61–62, 134
  Model-View-Presenter (MVP), 61–62
  Model-View-ViewModel (MVVM), 61–62
  Post-Redirect-Get pattern (PRG), 284–286
  Repository pattern, 227–229
  Table Module, 22–24
  Transaction Script, 22–24
PaymentCompleted, 41
PayPal, WURFL framework, 387
pc_NN, 391
P.CSS, 376
performance, event replay, 45
persistence, 5, 92
  ADO.NET., 222
  aggregates, 30
  database segments, 225–226
  domain logic, 22
  domain model, 95, 97
  Domain Model, 23
  entities, 25
  Entity Framework Code First, 95
  events, 43–44
  event sourcing, 98
  event store, 44
  infrastructure layer, 97
  mapping ASP.NET MVC, 219
  Match example, 37
  repositories, 31
  Table Module, 23
persistence layer, 220, 224, 228
  designing, 220–229
  domain model, 216
  modeling the business domain, 223
  Repository pattern, 227–229
persistence model, 214–215, 220, 227–228
persistent store, 126
personas, 52
per-status code views, 240
physical files, requests, 142–143
PICTURE, 375, 390
  Android, 376
  Chrome, 376
  Firefox, 376
  hidden costs, 376
  Opera, 376
pilot, 54
pixels
  Bootstrap, 378
  font size, 376–378
placeholders, URL parameter, 139
plain ASP.NET, 105
plain stored procedures, 31
plain text dates, 307
planning projects, 217–230

plugins,
  date-picker, 385
  Modernizr, 305
PoC (proof of concept), 53
pointer, CSS media queries level 4 standard, 368
polyfill, Modernizr, 376
polyglot persistence, 222, 230–233
  canonical example, 231
  costs, 231
  customer support, 231–232
  tools comparison, 232
  variegated information, 230
popovers, 274
  enabling, 272
  linking in Bootstrap, 175
  setting content, 272–273
popover view, 271–274
popups, tablets, 393
portrait orientation, 379
position CSS, 276
$.post, Ajax, 355
Post-Redirect-Get pattern (PRG), 284–286
PostResolveRequestCache, 138
POSTs, 138, 285
  controller methods, 295
  HTTP requests, 351–352
  login form, 280
  modal input form, 301
presentation, 48, 218–219
  layouts, 228–230
presentation layer, 30, 87, 90–92
  controller class, 146
  data model, 220
Presentation Model, 61
presentation screens, 90
  input model, 218
PRG (Post-Redirect-Get pattern), 284–286
primitive types, 26
  binding, 154–155
principal, 349
print, 366
privileged access, 30
productid, 142
program exceptions, 238–239
programming interface, task-oriented, 229
ProgressHub, 362
progressive enhancement, 373
projections, 43
  event store, 44
project.json, 121–122
proof of concept (PoC), 389
proof of concept (PoC), 53
properties, 25, 95
  naming conventions, 6
prototypes, 53–54
  turning views into, 57
pulling content, 351–359
pull-right class, 181
push notifications, 359
PUTs, 138

Q

queries, 30
  event sourcing, 98
  event store, 44
%QUERY, 308
query stack , 219
QueryString, 152–153, 284

R

Range, 287
RavenDB, 231–232
Raygun, 241
Razor, 74, 118, 125, 161
  ASP.NET virtual path provider, 161–163
  C# instructions in, 166
  code snippets, 164–167
  default location formats, 162
  horizontal scrolling, 268
  ImageEngine, 391
  login form, 280
  partial views, 255–256
  referencing CSS bundle, 233
  sections, 254
  template libraries, 357–359
  view engine, 161–163, 228–229
  view model, 247–250
  VirtualPathProviderViewEngine, 229
  Visual Basic instructions in, 166
RazorViewEngine, 229
RDBMS (relational database management system), 214
React, 77, 104
ReadDatabase root object, 39
read model, sample, 38–39
read stack,
  Match example, 37–39
  message-based formulation, 39–42
  repository classes, 228

ready handler, client-side validation, 302
readyState, 352
Recordset, 222
Recovery model, simple, full, 233
RecusionLimit, 340
RedirectResult, 159
RedirectToAction, 282
RedirectToRouteResult, 159
refactoring tools, 9
Refresh, 284
RegisterDisplayModes, 398
RegisterRoutes, 140
RegularExpression, 287
relational database management system (RDBMS), 214
relational databases, 223
  persistence model, 215
  polyglot persistence, 230
relational data stores, 50
REM, Bootstrap 4, 378
Remote, 287
remote procedure calls (RPCs), 78–79, 98
remote tasks, monitoring, 361–364
REM units, font size, 377
Render, 233
render actions, 257–258
RenderBody, page layout, 253
@RenderBody, 164
rendering,
  flexible, 182
  screen-based, 178–183
RenderPartial, 256
RenderSection, 254
RepeatText, 156
replay of events, 45
repositories, 25, 31, 96–97, 227–229
  data model, 220
  distinct stacks, 34
  implementing, 229
repository classes, 228
Repository pattern, 227–229
Representational State Transfer (REST), 138
RequestContext, 142
Request.Params, 152
requests, 152–153
  mapping to a controller method, 153–154
  physical files, 142
  routing incoming, 134–145
Required, 287
RequireJS, 253
ResetAll, 235
ReSharper, 9
resize, events, 379
resolution, CSS media queries level 4 standard, 368
resources,
  serving more effectively, 231–237
  using in web applications, 228
responseText, 352
responsiveness, 365
  responsive HTML templates, 392
  responsive images, 374–376
  responsive layouts, 176–183
  responsive pages, 388–389
Responsive Web Design (RWD), 176
  adapting to non-desktop devices, 373–379
  adaptive, 365
  Bootstrap, 366
  dealing with images, 374–376
  device independence, 371–373
  feature detection, 384–386
  fonts, 376–378
  foundation, 365–373
  history, 366
  Marcotte, Ethan, 366
  mobile-first, 373
  mobile-last, 373
  orientation, 378
  paradox, 372
  responsiveness, 365
  screen size, 369
REST (Representational State Transfer), 78, 138
  starter kit, 346
RESTful, 346
return on investment (ROI), 61
roles, 8
Route, 83, 144, 221
RouteArea, 144
RouteCollection, 142–143
RouteConfig, 140, 221
RouteData, 153
RouteData.Values, 153
route exceptions, 239–240
  search engine optimization (SEO), 240
RouteExistingFiles, 142–143
route handlers
  introducing, 142
  IRouteHandler, 142
route not found, 238
RoutePrefix, 144
routes
  application, 138–145
  configuration, 222

routes (*continued*)
  defining application, 139–141
  global.asax file, 139, 141
  input data, 153
  input forms, 284
  locale, 142
  matching, 141
  processing, 141
  productid, 142
  registering in ASP.NET MVC, 221
  URL routing HTTP module, 137–138
RouteTable, 221
RouteTable.Routes, 140
routing, 137–138
  attribute-based, 143
  enabling attribute, 144
  MVC (Model-View-Controller), 134
  preventing, 143
Rowling, J. K., 279
rows, Bootstrap, 177–178
RPCs (remote procedure calls), 98
Run, 117
runtime
  ASP.NET Core, 106–107, 113–118
  ASP.NET Core 1.0, 346
  ASP.NET MVC (Model-View-Controller), 134–137
  MVC (Model-View-Controller), 134–137
  Web API, 80, 347
runtime pipeline, 67

S

Safari
  date pickers, 305
  device independence, 371
  input types, 383
sagas, 41–42
Same-Origin Policy, 342
Save button, 310
scalability, 60
ScaleOut, 99
schemas, 126
  fluid grid in Bootstrap, 176–177
  login forms, 281
screen-based rendering, 179–180
screen orientation, 368–369
screens, 55–57, 60, 90, 366
screen size,
  breakpoints, 179–180
  conditional style sheets, 368–369
  RWD, 369
  screen-based rendering, 179–180
script-based custom polyfills, 383
ScriptBundle, 236–237
SCRIPT element, 342
script files
  bundling, 233–234
  in a web application, 228
scripting
  CSS media queries level 4 standard, 368
  placing, 252–253
scrolling
  adding to webpage, 268–271
  horizontal, 268–269
  on-demand, 269–271
search engine optimization (SEO)
  per-status code views, 240
  route exceptions, 240
sections, page layout, 254–255
security,
  HTTP 404 error, 239
  Web API, 348
segmented option buttons, 171
send, HTTP requests, 351–352
SEO (search engine optimization), 240
separation of concerns (SoC), 87, 146, 225
serialization, 300, 347
server controls, 73
server-side approaches, 92
server-side detection, 394–399
server-side preprocessing, 174
ServerVariables, 152–153
service buses,
  xxxConfig, 222
services,
  adding, 125
  data model, 220
Session, NHibernate, 225
SetMatchState method, 27
Shakespeare, William, 365
Shared folder,
  partial view, 228–229, 256
  Views, 250
shared kernel, 13, 16
shell script, 114
show events, 298–299
show/hide, 372
Shrink database, 232
Shrink transaction log, 232
SignalR, 118, 364
  ASP.NET, 104

SignalR (*continued*)
  connection ID, 361
  refreshing view, 364
significant entities, 229
Silverlight, 75, 104
Simple Object Access Protocol (SOAP), 346
simple query language, 367–368
single-page applications (SPAs), 64, 75, 83–85, 104
  popovers, 273
  weak points, 85
singleton services, 125
site icons, 252
sketches, 53
small fonts in RWD, 376
smartphones,
  creating mobile sites, 393
  HTML5, 382
  Responsive Web Design (RWD), 366
snapshots, 99
snippets, 167
SOAP (Simple Object Access Protocol), 346
SoC (separation of concerns), 146, 225
social authentication, 244–245
social networks, polyglot persistence, 230
software architects, 3, 51–52
software layers, 87
software models
  for business domain, 50
  object-oriented domain model, 24–26
Solution Explorer, 118–119
sources, 375
span4_*, 370
spans, 370
SPAs (single-page applications), 64, 75, 83–85, 104
  popovers, 273
  weak points, 85
sqlcmd, 232
SQL data stores, 231
SQL queries, 44
SQL Server, 215, 222
  comparison of tools, 232
SQL Server Management Studio, 232
SQL Server Profiler, 232
srcset, 375
StackOverflow, 286
Start method, 37
Start method, 27–28
startup, 116, 131, 223–224
  Code First, 226–227
  Main, 122–123
startup.cs, 360–361
statelessness, 146
status codes, 352–353
StopRoutingHandler, 143
storage, planning architecture, 65
storyboards, 55–57
strategic design, 5, 18
StringLength, 287
String/Object dictionary, RouteData.Values, 153
String types, 25
StyleBundle, 236–237
(RPC)-style HTTP services, Web API, 346
subdomains, 10–11
submitHandler, 302
supporting architecture, 24
swipe, 268
synchronization, 53
syntax, 134
system.web, 67–68, 110
System.Web.Extensions.dll, 338
System.Web.Routing, 142

T

Table Module, 23
tablets,
  HTML5, 382
  modal forms, 393
  popups, 393
  Responsive Web Design (RWD), 366
tab strips, 171
tactical design, 5, 18
tag helpers, 254
task-oriented programming interface, 229
tasks,
  monitoring remote, 361–364
  perform with action methods, 151
  from prototype, 57–58
TCP, 346
Team Foundation Server, 9
tel, input field, 382–383
Telerik, 264
TempData,
  login errors, 282
  Post-Redirect-Get pattern (PRG), 285–286
TempData dictionary, 295
template libraries,
  JavaScript, 357–359
  Knockout, 357
  mustache.js, 357
testable code, 225
text boxes, 309–310

three-tier architecture, 65, 87–89
tilde conversion, 251
tiles, rendering data, 260–261
time, new input types, 382
toggle facility, popover, 272
tokens, access, 349
tooltips, 175
top-down approach, 48–52
  UX, 51
Torvalds, Linus, 47
ToString method, 168
touch-based devices, 393
Transaction Script, 22
transformers, custom, 237
triggers, 296
true, cross-property validation, 292
T-SQL statements, 146
Twain, Mark, 214
Twitter
  Bootstrap, 77, 175
  OAuth, 349–350
  social authentication, 244
two-pizza rule, 17
two-tier schema, 75
Typeahead, 307–308
  Bloodhound, 308
types, 306

U

ubiquitous language, 4, 218
  acronyms, 10
  bounded context, 10–14
  choosing terms, 7–8
  constituent elements of, 7
  extension methods, 9
  and international teams, 9
  mockup, 53
  naming conventions, 6
  reflecting in code, 8–9
  sketch, 53
  wireframe, 53
UI (user interface), 49
Unit-of-Work pattern, 225
Universal Windows applications, 62
updateProgressBar, 363
upstream context, 15
url, input field, 382–383
@Url.Content, 164
URL parameters, 138–139
  ImageEngine, 391
URL patterns
  route, 140
  and routes, 138–139
URL routing HTTP module, 137–138
URLs
  defining syntax of, 134
  IMG element, 374
usability review, 52, 54
UseExceptionHandler, 117
user-agent sniffing, 386
user authentication,
  comparing ways, 242–244
  configuring, 241
  Microsoft ASP.NET Identity, 242–244
user experience (UX), 49, 90–91, 94, 337
  architecture, 52–59
  demo, 54
  improving, 304–311
  pilot, 54
  prioritization of features, 55
  prototype, 53–54
  synchronization, 53
  top-down approach, 51
  UX-Driven Design, 47
user interface (UI), 49
  granularity, 146
  presentation layer, 90
  updating in real time, 363–364
  Visual Basic for, 49
user permission, geolocation, 384
user preferences, 231
users, 349
  detecting authenticated, 129
UseStaticFiles, 117
UseXxx, 117
UXDD (UX-Driven Design), 47
  benefits, 59–62
  diagram, 60–61
  diagram of three steps, 55
  layout, 59
  list of tools, 56
UXPin, 56

V

validate step, 9
ValidationAttribute, 287
validation messages, 291–292
validation process, 286–295
validationSummary, 302
value types, 25–26, 29

variables, LESS, 171
VH, viewport units, 377
ViewBag, 92, 165
ViewData, 92
view engines, 92, 228–229
  custom, 229–230
  invoking, 163
  structure, 161
viewers, 275–276
ViewLocationFormats, 163, 229
View method, 163
View-Model, 225
ViewModelBase, 253
view models, 90, 92, 219–220, 227–228
  data model, 220
  exploring, 247–250
  instances, 249
  worker-service method, 249
viewports
  Bootstrap, 174
  Bootstrap grid system, 176–178
  HEAD section, 251
viewport units, 377
  VH, 377
  VMAX, 377
  VMIN, 377
  VW, 377
ViewResult, 159, 163
ViewResultBase, 159
views, 61, 125, 225
  adapting to devices, 381–389
  adding master/detail, 271–276
  areas, 145
  areas of ASP.NET MVC, 220
  ASP.NET MVC, 217
  ASP.NET virtual path provider, 161–163
  drill-down, 274–276
  error handling, 238–241
  HTML helpers, 254
  image-resizing tool, 390
  page layout, 250
  partial views., 167
  Post-Redirect-Get pattern (PRG), 285–286
  presenting elements, 254–258
  refreshing after modal update, 364
  routing based on detected device, 397–398
  tag helpers, 254
  URL routing HTTP module, 137–138
Views folder, 230
viewstate, 76
virtual measurements, screen-based rendering, 179–180
VirtualPathProviderViewEngine, 229
visible-xx, 181
Visual Basic, 73, 221
  in Razor, 166
  user interface, 49
Visual Studio, 64, 118
  areas of ASP.NET MVC, 221
  error handling, 238
  Gulp file, 120–121
  LESS, 174
  page layout, 251
  partial views, 255–256
Visual Studio 2015, 64, 118
  ASP.NET Core template, 128
VMAX, viewport units, 377
VMIN, viewport units, 377
VW, viewport units, 377

W

W3C (Worldwide Web Consortium), 366
WCF (Windows Communication Foundation), 79–80
Web APIs, 21–22, 67–68, 118
  ASP.NET Core, 82
  ASP.NET MVC, 347–348
  designing, 346–350
  security, 348–351
  standalone, 348
  xxxConfig, 222
WebApplication, 123
web.config file
  defining syntax of URLs, 134
  login page, 279–281
Web Essentials, 174
web farms, Post-Redirect-Get pattern (PRG), 286
web flavors, 103–105
Web Forms, 64, 67–68, 72–75, 105, 118, 130
  controller, 146
  differences from ASP.NET MVC, 133
  good points, 73–74
  login forms, 281–282
  page-agnostic URLs, 136
  page layouts, 253
  paging data, 263–264
  partial views, 167, 255–256
  rendering data as tiles, 260–261
  view engines, 161
Web Forms ASPX, exposing JSON content, 338
WebGrease, 236–237

webHttpBinding, 346
Web Pages, 104, 118
web programming, 129–132
webroot folder, 119
web services
  caching, 99
  front ends, 79
  hosting, 82
website icons, 252
website resources
  http://caniuse.com website, 305–307, 384–386
  http://detectmobilebrowsers.com, 386
  http://github.com/eternicode/bootstrap-datepicker, 305
  http://haacked.com/ archive/2009/06/25/json-hijacking.aspx, 339
  http://igrigorik.github.io/http-client-hints, 389
  http://knockoutjs.com, 357
  http://modernizr.com, 385
  http://scottjehl.github.io/picturefill, 376
  https://developer.mozilla.org/en-US/ docs/Web/HTTP/Access_control_CORS, 346
  https://github.com/ janl/mustache.js, 357
  http://web.wurfl.io, 376, 387–388, 392
  http://wurfl.sourceforge.net, 395
  http://www.expoware.org/wit.html, 375
  http://www.expoware.org/wjs.htm, 388
  http://www.json.org, 338
  http://www.newtonsoft.com/json, 338
  http://www.quirksmode.org/js/detect.html, 386
  http://www.scientiamobile.com, 388, 395
  http://www.w3.org/TR/css3-mediaqueries, 368
web solutions, 64–66
WFC (Windows Communication Foundation), 79–80, 346
what-if scenarios, 45
widgets, reusable alerts, 294
width,
  CSS 3 media queries, 367
  IMG element, 374
wiki, 8
windows, 379
  Web API, 347
Windows CE devices, Responsive Web Design (RWD), 366
Windows Communication Foundation (WCF), 79–80, 346
wireframes, 53–54, 56, 60
Wirify, 57
Wirth, Niklaus, 48
w_NNN, 391
worker-service classes, 225–227
  placement, 227
worker-service methods, 249
workflow
  from prototype, 57–58
  saga, 42
Worldwide Web Consortium (W3C), 366
wrappers, 302
wrapping results, action method, 158–160
WS-* protocols, 346
WURFL, 394–395
  capabilities, 395
  cloud API, 395–397
  cloud user dashboard, 396
  framework, 387
  NuGet, 394
  on-premises API, 394
  server-side detection, 394
WURFL.JS, 393
  client-side detection, 387–388
  properties, 387
  test page, 388
WURFL server, 387
WurlfService class, 396
wwwroot folder, 119, 121

X

XAML-based applications, 62
XML, 126
XML .browser files, 392
XML formatter, 341
XMLHttpRequests, 345, 351–352
  pulling content, 351
XML serializers, 342
XS devices, Bootstrap columns and rows, 178
xxxConfig, 222
xxxFor, 291

Y

Young, Greg, 19
YourApp.MvcApplication, 220

About the author

DINO ESPOSITO is CTO and co-founder of Crionet, a startup providing software and IT services to professional tennis and sports companies. Dino still does a lot of training and consulting and is the author of several other books on web and mobile development. His most recent book is *Architecting Applications for the Enterprise, Second Edition*, written along with Andrea Saltarello. A Pluralsight author, Dino speaks regularly at industry conferences and community events. You can follow Dino on Twitter at @despos and through his blog at *http://software2cents.wordpress.com*.