Author Picks

FREE



Exploring .NET Core

with Microservices, ASP.NET Core, and Entity Framework Core

Chapters selected by Dustin Metzgar

manning



Exploring .NET Core with Microservices, ASP.NET Core, and Entity Framework Core

Selected by Dustin Metzgar

Manning Author Picks

Copyright 2017 Manning Publications
To pre-order or learn more about these books go to www.manning.com

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2017 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.



Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617295089
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 21 20 19 18 17 16

contents

Introduction iv

REFACTORING 1

Refactoring
Chapter 4 *from Re-Engineering Legacy Software by Chris Birchall. 2*

IDENTIFYING AND SCOPING MICROSERVICES 39

Identifying and scoping microservices
Chapter 3 *from Microservices in .NET Core by Christian Horsdal Gammelgaard. 40*

CREATING AND COMMUNICATING WITH WEB SERVICES 63

Creating a Microservice
Chapter 7 *from .NET Core in Action by Dustin Metzgar. 64*

CREATING WEB PAGES WITH MVC CONTROLLERS 85

Creating web pages with MVC Controllers
Chapter 4 *from ASP.NET Core in Action by Andrew Lock. 86*

QUERYING THE DATABASE 111

Querying the database
Chapter 2 *from Entity Framework Core in Action by Jon Smith. 112*

index 141

introduction

Developers love to write code. We follow the latest trends, learn new techniques, and try to implement what we learn in our daily work. Most developers, however, are not writing projects in completely green fields. Most often we write software to maintain, augment, replace, or interact with older applications. .NET Core integrates modern development practices into your daily work with legacy applications, and may be just the option you've been looking for.

If you're not developing with the .NET Framework, you've likely heard of .NET before and decided not to use it for various reasons. If you are one of the many users who are not using .NET because it is only available on Windows, doesn't have a strong open source community, and isn't keeping up with high performance web frameworks, you should take another look at .NET Core. .NET Core is now available in Linux with open-source source code. You'll also find that current .NET Core frameworks are lightweight, modular, and focused on delivering high-performance results. Perhaps you are already a .NET Framework developer. If so, you may want to take advantage of the latest advances like containers, ASP.NET Core, or Entity Framework Core. Either way, there has never been a better time to use .NET.

This collection of chapters from several Manning books offers guidance on refactoring legacy code, carving monolithic n-tier architectures into microservices, and using ASP.NET Core and Entity Framework Core to build microservices and web applications. Learn how to apply modern development practices to your daily work and feel confident that the code you're writing is making your applications better.

We hope you'll enjoy this selection and the code you'll write because of it.

Dustin Metzgar
Author of *.NET Core in Action*

Refactoring

Legacy code is code that you have no confidence in your ability to change. Chris Birchall's book, *Re-Engineering Legacy Software*, gives you the tools and techniques to tackle legacy projects with confidence. What better opportunity is there to tackle those projects than to move to .NET Core? In this chapter, you'll learn how to understand, test, and refactor legacy code, which will make moving to a modern development framework much easier.

Chapter 4 from *Re-Engineering Legacy Software* by Chris Birchall.

Refactoring

This chapter covers

- Methods for maintaining discipline when refactoring
- Common smells found in legacy code, and refactoring techniques for removing them
- Using automated tests to support refactoring

In this chapter we'll look at your most important weapon in the fight against legacy, namely *refactoring*. I'll introduce some general tips for effective refactoring, as well as a few specific refactorings that I often use in real-world code. We'll also look at techniques for writing tests for legacy code, which is vital if you want assurance that your refactoring work hasn't broken anything.

4.1 Disciplined refactoring

Before we start looking at specific refactoring techniques, I want to talk about the importance of discipline when refactoring code. When performed correctly, refactoring should be perfectly safe. You can refactor all day long without fear of introducing any bugs. But if you're not careful, what started out as a simple refactoring can rapidly spiral out of control, and before you know it you've edited half the files

in the project, and you're staring at an IDE full of red crosses. At this point you have to make a heart-wrenching decision: should you give up and revert half a day's work, or keep going and try to get the project back into a compiling state? Even if you can reach the other side, you have no guarantee that the software still works as it's supposed to.

The life of a software developer is stressful enough without having to make decisions like that, so let's look at some ways to avoid getting into that situation.

4.1.1 Avoiding the Macbeth Syndrome

> *I am in blood*
> *Stepped in so far that, should I wade no more,*
> *Returning were as tedious as go o'er.*
>
> —Macbeth, act 3, scene 4

When Macbeth uttered these words, he'd already done away with his beloved King Duncan, two innocent guards, and his best friend Banquo, and he was debating with himself whether to stop the carnage or to carry on slaughtering people until he had no enemies left. (In the end he chooses the latter path, and, suffice it to say, it doesn't work out.)

Now, if I'm interpreting the text correctly, and this may come as quite a shock to the Shakespeare scholars among you, the Scottish play was actually intended to be an allegory about undisciplined refactoring. Macbeth, spurred on by a feature request from his wife and Project Manager, Lady Macbeth, starts out with the simple aim of removing a piece of global mutable state called King Duncan. He succeeds in this, but it turns out that there were some implicit dependencies that also needed to be refactored. Macbeth tries to tackle all of these at once, leading to excessive amounts of change, and the refactoring rapidly becomes unsustainable. In the end, our hero is attacked by trees and decapitated, which is pretty much the worst result I can imagine for a failed refactoring.

So what can we do to avoid ending up like poor Macbeth? Let's look at a few simple techniques.

4.1.2 Separate refactoring from other work

Often you realize that a piece of code is ripe for refactoring when you're actually trying to do something else. You may have opened the file in your editor in order to fix a bug or add a new feature, but then decide that you may as well refactor the code at the same time.

For example, imagine you've been asked to make a change to the following Java class.

```
/**
 * Note: This class should NOT be extended -- Dave, 4/5/2009
 */
public class Widget extends Entity {
    int id;
    boolean isStable;
```

```
    public String getWidgetId() {
        return "Widget_" + id;
    }

    @Override
    public String getEntityId() {
        return "Widget_" + id;
    }

    @Override
    public String getCacheKey() {
        return "Widget_" + id;
    }

    @Override
    public int getCacheExpirySeconds() {
        return 60; // cache for one minute
    }

    @Override
    public boolean equals(Object obj) {
        ...
    }
}
```

The specification for widgets has changed so that the cache expiry of a widget should depend on the value of its `isStable` flag. You've been asked to update the logic in the `#getCacheExpirySeconds()` method accordingly. But as soon as you glance at the code, you notice a number of things you'd like to refactor.

- There's a comment from some guy called Dave saying the class shouldn't be extended, so why not mark it as `final`?
- The fields are mutable and they have package-private visibility. This is a dangerous combination, as it means other objects might change their values. Maybe they should be private and/or final?
- There is redundancy among the various ID/key-generating methods. The ID generation logic could be factored out into one place.
- The class overrides `#equals(Object)` but not `#hashCode()`. This is bad form and can lead to some nasty bugs.
- There's no comment explaining the meaning of the `isStable` flag. It might be nice to add one.

The actual change to the cache expiry logic is quite simple, so it's tempting to combine it with a spot of refactoring. But be careful! Some of the refactorings in the preceding list are more complex than they first appear.

- Even though Dave says we shouldn't extend `Widget`, maybe somebody already did. If there are any subclasses of `Widget` anywhere in the project, then marking the class as `final` will cause compilation errors. You'll have to go through the subclasses one by one and decide what should be done about them. Is it OK to

extend `Widget` after all? Or should those subclasses be fixed to remove the inheritance?

- What about those mutable fields? Is anybody actually mutating them? If so, should they be? If you want to make them immutable, you'll have to change any code that is currently relying on their mutability.
- Although the lack of a `#hashCode()` method is generally a Bad Thing in Java code, there might be some code somewhere that actually relies on this behavior. (I've seen code like this in the wild.) You'll have to check all code that deals with `Widget`s. Also, if the person who wrote `Widget` (Dave?) also wrote any other entity classes, the chances are that they're all missing this method. If you decide to fix all of them, it might become a much larger job.

If you try to take on all this refactoring at the same time as the change you were originally planning, that's a lot of stuff to fit in your head. Maybe you're a refactoring wizard, in which case go for it, but I wouldn't trust myself to handle that kind of cognitive load without making any mistakes.

It's much safer to split the work into stages: either make the change, then refactor, or vice versa. Don't try to do both at the same time. Splitting refactoring work and other changes into separate commits in your version control system also makes it easier for you and other developers both to review the changes and to make sense of what you were doing when you revisit the code later.

4.1.3 Lean on the IDE

Modern IDEs provide good support for the most popular refactoring patterns. They can perform refactorings automatically, which obviously has a number of benefits over performing them by hand.

- *Faster*—The IDE can update hundreds of classes in milliseconds. Doing this by hand would take a prohibitively long time.
- *Safer*—They're not infallible, but IDEs make far fewer mistakes than humans when updating code.
- *More thorough*—The IDE often takes care of things that I hadn't even thought of, such as updating references inside code comments and renaming test classes to match the corresponding production classes.
- *More efficient*—A lot of refactoring is boring drudge work, which is what computers were designed to do. Leave it to the IDE while you, rock star programmer that you are, get on with more important things!

To give you a taste of what the IDE is capable of, figure 4.1 shows a screenshot of IntelliJ IDEA's Refactor menu for a Java project. (Your menu might look slightly different, depending on what IDE you use and what plugins you have installed.) It's definitely worth the time to go through each of your IDE's refactoring options, try them out on some real code, and see what they do.

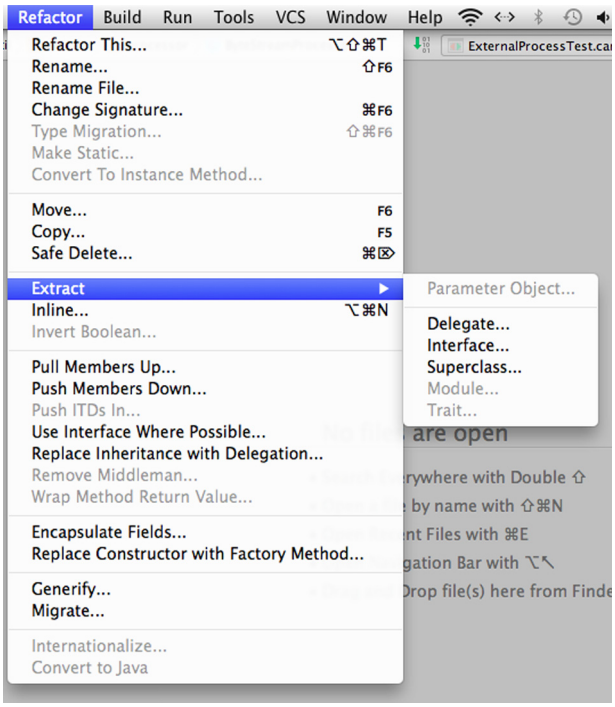


Figure 4.1 IntelliJ's Refactor menu

As an example of what your IDE can do for you, let's use IntelliJ IDEA to help us replace a large and unwieldy constructor using the Builder pattern. Figure 4.2 shows a Java class that represents a tweet. As you can see, the large number of fields results in a constructor that's frankly ridiculous.

Figure 4.3 shows an example of using the class's constructor directly. It's very hard to read, and a lot of the fields are optional, so this is an ideal candidate for using the Builder pattern.

```
public final class Tweet {
    private final Coordinates coordinates;
    private final boolean favorited;
    private final boolean truncated;
    private final Date createdAt;
    private final Entities entities;
    private final Long inReplyToUserId;
    private final List<Contributor> contributors;
    private final String text;
    private final int retweetCount;
    private final Long inReplyToStatusId;
    private final long id;
    private final Geo geo;
    private final boolean retweeted;
    private final boolean possiblySensitive;
    private final String place;
    private final User user;
    private final String inReplyToScreenName;
    private final String source;

    public Tweet(Coordinates coordinates, boolean favorited, boolean truncated, Date createdAt,
                 Entities entities, Long inReplyToUserId, List<Contributor> contributors, String text,
                 int retweetCount, Long inReplyToStatusId, long id, Geo geo,
                 boolean retweeted, boolean possiblySensitive, String place,
                 User user, String inReplyToScreenName, String source) {
        this.coordinates = coordinates;
        this.favorited = favorited;
        this.truncated = truncated;
        this.createdAt = createdAt;
        this.entities = entities;
        this.inReplyToUserId = inReplyToUserId;
        this.contributors = contributors;
        this.text = text;
        this.retweetCount = retweetCount;
        this.inReplyToStatusId = inReplyToStatusId;
        this.id = id;
        this.geo = geo;
        this.retweeted = retweeted;
        this.possiblySensitive = possiblySensitive;
        this.place = place;
        this.user = user;
        this.inReplyToScreenName = inReplyToScreenName;
        this.source = source;
    }

    // getters, other methods ...

}
```

Figure 4.2 A Java class representing a tweet

```
private final Tweet myTweet = new Tweet(
        null, false, false, new Date(), new Entities(),
        null, Collections.<Contributor>emptyList(),
        "hello world", 123, null, 456789, null, false,
        false, null, new User(), null, "twitter.com"
);
```

Figure 4.3 Using the Tweet constructor directly

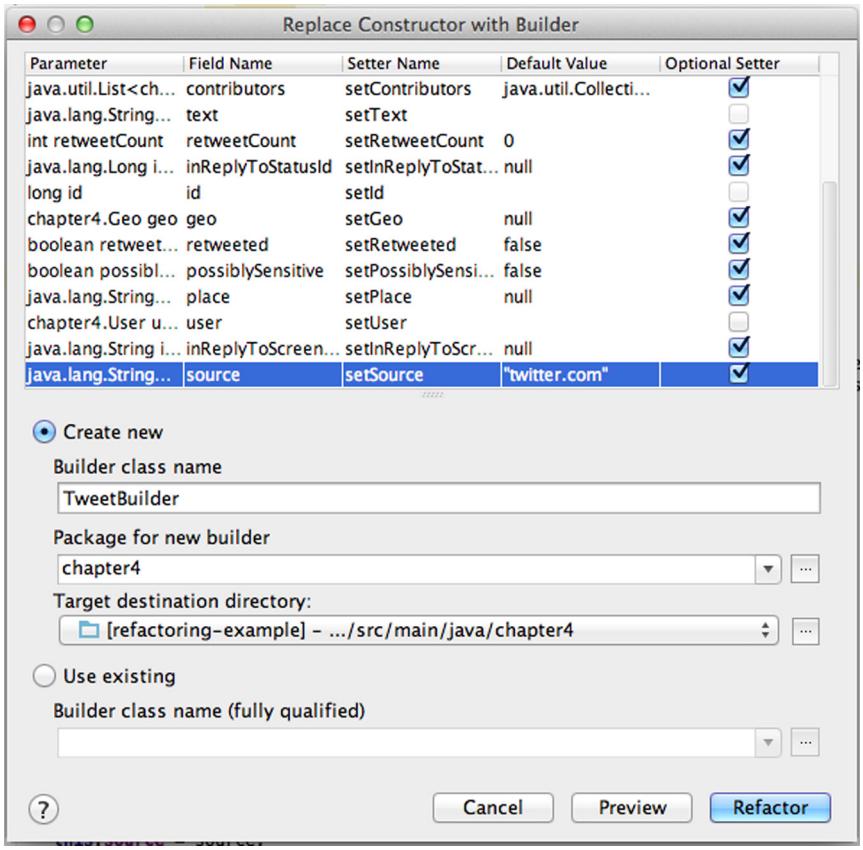


Figure 4.4 The Replace Constructor with Builder wizard

Let's ask IntelliJ to create a Builder to fix this mess. Figure 4.4 shows the Replace Constructor with Builder wizard, where I can set default values for optional fields.

When I click the Refactor button, the IDE generates a new class called `TweetBuilder`. It also automatically rewrites any code that's calling the `Tweet` constructor, updating it to use `TweetBuilder` instead. After a little manual reformatting, the code to create a new `Tweet` now looks like figure 4.5. Much better!

```
private final Tweet myTweet = new TweetBuilder()
        .setId(456789)
        .setText("hello world")
        .setRetweetCount(123)
        .setUser(new User())
        .createTweet();
```

Figure 4.5 Creating a `Tweet` using `TweetBuilder`

IDEs make mistakes too

Occasionally the IDE can get it wrong. Sometimes it's overzealous, trying to update files that are completely unrelated to the change you want to make. If you rename a method called `execute()`, the IDE might try to update an unrelated code comment such as "I will execute anybody who touches this code." Sometimes it doesn't update a file that it should. For example the IDE's dependency analysis is often foiled by the use of Java reflection.

With this in mind, it's best not to trust the IDE unconditionally:

- If the IDE offers a preview of a refactoring, inspect it carefully.
- Check that the project still compiles afterward, and run automated tests if you have them.
- Use tools like grep to get a second opinion.

4.1.4 Lean on the VCS

I assume that you're using a version control system (VCS) such as Git, Mercurial, or SVN to manage your source code. (If not, put this book down and go and fix that situation right now!) This can be really useful when refactoring. If you get into the habit of committing regularly, you can treat the VCS as a giant Undo button. Any time you feel like your refactoring is getting out of control, you have the freedom to back out by hitting the Undo button (reverting to the previous commit).

Refactoring is often an exploratory experience in which you don't know whether a particular solution will work out until you try it. The safety net of the VCS gives you the freedom to experiment with a number of solutions, safe in the knowledge that you can back out whenever you want. In fact, effective use of branches means that you can have a few different solutions on the go at the same time, while you explore the pros and cons of each approach.

Figure 4.6 shows an example of the Git commits and branches that might be left after a session of experimental refactoring. The newest commits are at the top. You can see that there were a couple of experimental branches that didn't end up getting merged into the master branch.

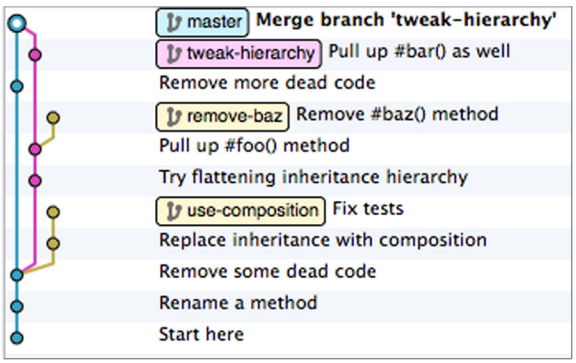


Figure 4.6 An example of using Git branches to aid refactoring

4.1.5 The Mikado Method

One method that I have recently been using with great success to implement large changes, including refactoring, is called the Mikado Method. It's very simple but effective. Basically it involves building up a dependency graph of all the tasks you need to perform, so that you can then execute those tasks more safely and in an optimal order. The dependency graph is constructed in an exploratory manner, with plenty of backtracking and leaning on the VCS.

For more details on the method itself and the motivations behind it, I highly recommend Ola Ellnestam and Daniel Brolund's book *The Mikado Method* (Manning, 2014).

Figure 4.7 shows an actual Mikado graph that I drew recently when I was porting the UI layer of a large application from one web framework to another.

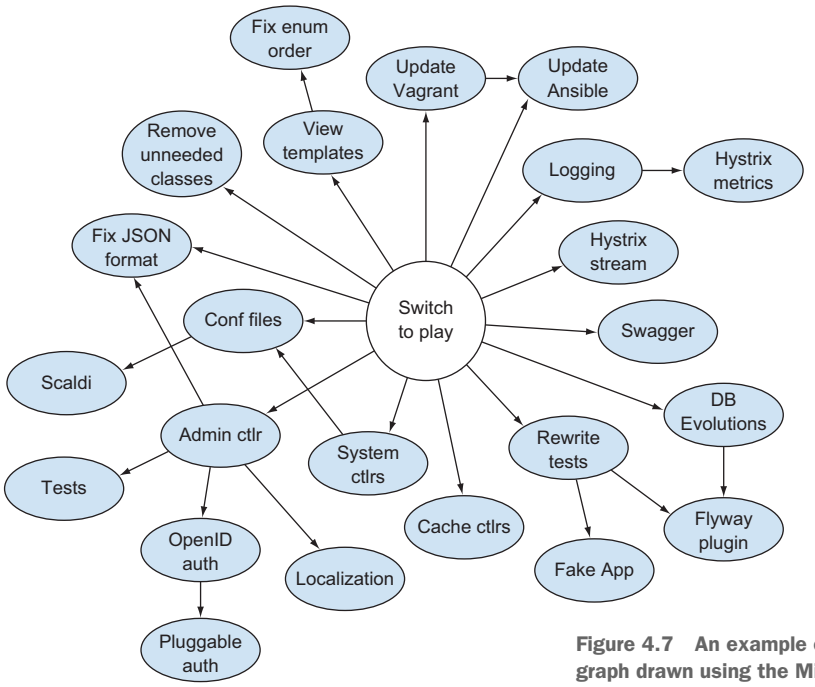


Figure 4.7 An example of a dependency graph drawn using the Mikado Method

4.2 Common legacy code traits and refactorings

Every legacy codebase is different, but a few common traits tend to surface time and time again when reading through legacy code. In this section we'll take a look at a few of these traits and discuss ways in which we can remove them. It's possible to devote entire books to this subject, but I only have one chapter to play with, so I've picked just a few representative examples of problems that have been especially prevalent in my own experience.

Imagine that you maintain World of RuneQuest, an online fantasy RPG. You're planning to start development on a new version of the game, but recently you've noticed that the code has become bloated and disorganized, and development velocity has dropped as a result. You want to give the codebase a thorough spring clean before you start developing the new version. Let's look at some areas you could tackle.

4.2.1 Stale code

Stale code is any code that remains in the codebase even though it's no longer needed. Deleting this unneeded code is one of the easiest, safest, and most satisfying refactoring tasks you can hope for. Because it's so easy, it's often a good way to warm up before embarking on more serious refactoring.

Removal of stale code has a number of benefits:

- It makes the code easier to understand, because there's now less code to read.
- It reduces the chance of somebody wasting time on fixing or refactoring code that isn't even used.
- As a satisfying bonus, every time you delete some code, the project's test coverage increases.

Stale code can be divided into a few categories.

COMMENTED-OUT CODE

This is the lowest of all low-hanging fruit. If you see a block of code that has been commented out, don't think twice, just delete it! There's absolutely no reason to leave commented-out code lying around. It's often left deliberately as a record of how the code was changed, but that's exactly what the version control system is for. It's pointless noise, making the surrounding code more difficult to read.

DEAD CODE

Dead code is any code in your software that will definitely never be executed. It might be a variable that's never used, a branch of an `if` statement that's never taken, or even a whole package of code that's never referenced.

In the following simple example of dead code, the `armorStrength` variable can never have a value greater than 7, so it's always less than 10 and the `else` block will never run. It's dead code and thus can and should be removed.

```
int armorStrength = 5;
if (player.hasArmorBoost()) {
    armorStrength += 2;
}
...
if (armorStrength < 10) {
    defenceRatio += 0.1;
} else {
    defenceRatio += 0.2;
}
```

Armor strength is always 7 or less, so this branch will always run.

This branch is dead code.

There are many tools that can help you find and remove dead code from your codebase. Most IDEs will point out fields, methods, and classes that aren't referenced, and tools such as FindBugs (discussed in chapter 2) also include relevant rules.

ZOMBIE CODE

Some code that's actually dead may look very much alive. I call this zombie code. Its liveliness (or lack of) is impossible to discover just from reading the surrounding source. Examples include

- Code that branches based on data received from an external source such as a database, in which some of the branches are never triggered
- Pages of a website or screens of a desktop application that are no longer linked from anywhere

As an example of the first point, let's alter the previous code sample so that the value of `armorStrength` is read from a DB. Just by reading the code, you have no idea what value `armorStrength` might have, so the code looks alive and reasonable.

```
int armorStrength = DB.getArmorStrength(play.getId());
...
if (armorStrength < 10) {
    defenceRatio += 0.1;
} else {
    defenceRatio += 0.2;
}
```

Is this branch dead or alive?

But when you look at the actual data in the DB, you might find that all 5 million players have an armor strength of less than 10, so in fact the `else` will never run.

In this case, you should check all the places that are setting this value in the DB (to make sure that a strength of 10 or more is impossible), and then add a DB constraint to act as documentation for your understanding of the model, before finally deleting the unneeded `else` block.

In the case of pages or screens that aren't linked from anywhere, it's often difficult and time-consuming to confirm that a given page of a web application is dead. Even if there are no links to it, users might still be accessing it directly, perhaps via browser bookmarks, so you may have to trawl through web server access logs to check that the page is safe to delete.

I was affected by a real-world example of zombie code when I joined a team that maintained a large legacy web application. Before I joined the team, they'd been A/B testing a major update to the site's top page. There were two separate versions of the page, and users were directed to one or the other based on a per-user flag in the DB. Little did I know, the A/B test had already finished (the flags in the DB had all been set to the same value) but nobody had deleted the losing version of the page. So every time I had to make a change to the top page, I faithfully replicated my changes across both versions, wasting a lot of time in the process.

A/B TESTING A/B testing is a commonly used process to investigate the effect that a change to a website will have on users' behavior. The basic idea is to introduce the change to only a limited segment of the site's users at first. Users are separated into two buckets, A and B, and one group is served the normal site, while the other is served a version of the site that includes the change. You then measure key metrics (page views, attention time, scroll depth, and so on) for each user segment and compare the results.

EXPIRED CODE

It's common for business logic to apply only within a certain timespan, especially in web applications. For example, you might run a particular ad campaign or A/B test for a few weeks. In World of RuneQuest, perhaps you run half-price sales on in-game purchases occasionally. This often requires corresponding date-specific logic in the code, so it's common to see code that looks like this:

```
if (new DateTime("2014-10-01").isBeforeNow() &&
    new DateTime("2014-11-01").isAfterNow()) {
  // do stuff ...
}
```

While there's nothing wrong with temporary code like this, developers often forget to go back and delete it after it has served its purpose, leading to a codebase littered with expired code.

There are a couple of ways of avoiding this problem. One is to file a ticket in your issue tracker to remind yourself to delete the code. In this case, make sure that the ticket has a deadline and is assigned to a specific person; otherwise it's easily ignored.

A smarter solution is to automate the check for expired code.

1. When writing code that has a limited lifespan, add a code comment in a specific format to mark it as expiring code.
2. Write a script that can search through the codebase, parse these comments, and flag any expired ones.
3. Set up your CI server to run this script regularly and fail the build if it finds any expired code. This should be much harder to ignore than a ticket in the issue tracker.

The following code shows an example of one of these comments.

```
// EXPIRES: 2014-11-01
if (new DateTime("2014-10-01").isBeforeNow() &&
    new DateTime("2014-11-01").isAfterNow()) {
  // do stuff ...
}
```

AUTOMATING EXPIRY CHECKS USING MACROS In languages that support macros (running code at compile time), it's possible to make the project fail to compile if there's any expired code lying around. I know of a couple of Scala libraries that can do this for you: Fixme (https://github.com/tysonjh/fixme) and DoBy (https://github.com/leanovate/doby).

4.2.2 Toxic tests

When you're handed a legacy project to maintain, you might count yourself lucky if it includes some automated tests. They can often act as a good substitute for documentation, and the presence of tests gives a hint that the code quality might be reasonable. But be careful: there are some kinds of tests that are worse than no tests at all. I call these toxic tests. Let's have a look at some of them.

TESTS THAT DON'T TEST ANYTHING

The basic template for a good software test, no matter whether it's manual or automated, unit, functional, system, or whatever, can be summed up in just three words: given-when-then.

- *Given* some preliminary conditions and assumptions,
- *when* I do this
- *then* this should be the result.

Surprisingly often I encounter tests in legacy projects that don't fit this simple pattern. I see a lot of tests that are missing the "then" part, meaning that they don't include any assertions to check the result of the test against what was expected.

Imagine that World of RuneQuest uses an event bus to manage in-game events and their corresponding notifications. For example, when a player proposes a treaty with another player, an event is posted to the event bus. A listener might pick up this event and send a notification mail to the player in question. The event bus is implemented using a bounded queue data structure that automatically discards the oldest element when it becomes full, in order to bound the amount of memory used. Here's the JUnit 3 test that the original developer wrote to check that the bounded queue worked as expected.

```
public void testWorksAsItShould() {
    int queueSize = 5;
    BoundedQueue<Integer> queue =
            new BoundedQueue<Integer>(queueSize);
    for (int i = 1; i <= 20; i++) {
        queue.enqueue(i);
    }
    while (!queue.isEmpty()) {
        System.out.println(queue.dequeue());
    }
}
```

Most likely the test was written before the days of CI, so the author never expected it to be run more than once. They ran the test, manually verified that the numbers printed to the screen matched what they expected, and then forgot about it.

But these days that kind of testing just doesn't cut the mustard. We want our automated tests to guard against regressions, but the preceding test doesn't. Even if you accidentally change the behavior of the `BoundedQueue` class, the test won't fail; it will simply spit out a different set of numbers to the console.

Tests like this one are particularly toxic because they look and feel like a test, even though they aren't testing anything at all. They falsely inflate the project's test count and test coverage, giving developers a false sense of security. The solution is simple: either fix the test by adding proper assertions, or delete it. (In this particular case, an even better solution would be to delete both the test and the `BoundedQueue` class itself, replacing it with a trustworthy third-party implementation such as Guava's `EvictingQueue`.)

BRITTLE TESTS

Good unit tests can prove valuable when refactoring, by providing assurances that the behavior of given parts of the codebase is preserved. But if you find that tests often break when you refactor, it may be a sign that the tests are too brittle. In this case, the tests become a hindrance, as you end up spending more time fixing them than refactoring.

A common cause of fragility is unit testing at too fine-grained a level. Continuing our `BoundedQueue` example, imagine you wrote a test for it like the following (this time using JUnit 4 syntax).

```
@Test
public void wibbleFlagIsSet() throws Exception {
    int queueSize = 5;
    BoundedQueue<Integer> queue =
            new BoundedQueue<Integer>(queueSize);

    Field wibble =
            BoundedQueue.class.getDeclaredField("wibble");
    wibble.setAccessible(true);

    assertThat(wibble.getBoolean(queue), is(false));

    for (int i = 1; i <= queueSize; i++) {
        queue.push(i);
    }

    assertThat(wibble.getBoolean(queue), is(true));
}
```

Expose the private field "wibble"

Flag should start off false

Fill the queue

Flag should now be true

This test uses Java reflection hackery to expose a private field and check its value. So if you ever remove or rename this field in the course of a refactoring, the test will break.

In general there's no need to write tests like this. We should be testing the behavior that components expose to each other, not any internal state they might be holding. If you ever find a test that's accessing private members of a class, or you find yourself wanting to write one, it might be a hint that the class contains too much state or is doing too much. You should consider splitting it into smaller classes that are easier to test.

RANDOMLY FAILING TESTS

A good test is completely deterministic, meaning that its result shouldn't be affected by changes in CPU load, thread scheduling, network congestion, other tests running in parallel, or any other external factor. But some tests don't achieve this gold standard

and will fail occasionally. Examples include concurrency tests that depend on processing completing within a certain timeout, and integration tests that depend on the contents of an external database or filesystem.

These tests are dangerous, as they lead developers to start treating a test suite with a few failing tests as normal. Your test suite should be as simple as possible to understand: zero failing tests = GOOD, anything else = THE SKY IS FALLING! It's difficult to maintain this sense of urgency if two or three tests in your suite fail occasionally. Consequently, any randomly failing tests should be

- *Fixed*—If it is easy to do so
- *Disabled*—If they can be fixed but you don't have time to do so right now
- *Deleted or rewritten*—If they look very difficult to fix

4.2.3 A glut of nulls

Tony Hoare, the inventor of the null reference, calls the following his "billion-dollar mistake."

```
if (x == 0) {
    return null;              <-- NOOOOOOO!!!
}
```

Null references are the bane of the programmer's existence, and my heart sinks every time I see a `NullPointerException` (or .NET's equivalent `NullReferenceException`).

The use of `null` makes it more difficult to read and write code because nullability is not made explicit, at least in languages like Java. When reading a block of code, it's not obvious that a given variable might be `null`, so the reader must remember to keep the implicit nullability of references in mind at all times.

Modern languages strive to make developers' lives easier concerning `null`. Kotlin, for example, builds the concept of nullability into its type system, so that `String` and `String?` are separate types (non-nullable and nullable strings, respectively). The compiler is also smart enough to know whether you've performed a `null` check on a nullable reference, so that this will fail to compile:

```
print(player.getCharacterId())   <-- Assuming player is of type Player?
```

In contrast, the next example will compile just fine:

```
if (player != null) {
    print(player.getCharacterId())
}
```

Scala provides an `Option` type in its standard library to reduce the need for `null`. A value with `Option` type can be either a `Some(thing)` or a `None`, where `None` assumes the role for which `null` is used in other languages. You might wonder if there's actually any benefit to replacing a `null` with a `None`, but the point is that the `Option` type

makes the "there was no result" case more explicit and forces the developer to deal with it, whereas a `null` result can be easily overlooked.

Compare the following Java and Scala code for retrieving a `Player` from a database. First the Java:

```
Player player = playerDao.findById(123);
System.out.println("Player name: " + player.getName());
```

Returns null if player with ID 123 does not exist

In the Java case, the developer has forgotten to include a `null` check, so if player 123 isn't found in the database, this code will throw a `NullPointerException`.

Now let's look at the same code written in Scala.

```
val maybePlayer = playerDao.findById(123)
// Do a pattern-match on the result
maybePlayer match {
  case Some(player) => println("Player name: " + player.getName())
  case None => println("No player with ID 123")
}
```

Returns an Option[Player]

In this case, because the DAO gives us an `Option`, both the "player exists" and "player does not exist" cases are obvious, and we're forced to handle both cases appropriately.

In Java it's possible to emulate the approaches used in languages such as Scala. If you're using Java 8 (which is unlikely, if you're working with legacy code), you can use the `java.util.Optional` class. Otherwise, Google's Guava library contains, among a host of other useful utilities, a class called `com.google.common.base.Optional`. The following code shows one way you could rewrite the previous code using Java 8's `Optional`.

```
Optional<Player> maybePlayer = playerDao.findById(123);
if (maybePlayer.isPresent()) {
    System.out.println("Player name: " + maybePlayer.get().getName());
} else {
    System.out.println("No player with ID 123");
}
```

If you have a lot of legacy Java code that uses `null` extensively and you don't want to rewrite it all to use `Optional`, there's a simple way to keep track of null-ness and thus make your code more readable. JSR 305 standardized a set of Java annotations that you can use to document the nullability (or otherwise) of various parts of your code. This can be useful purely as documentation, to make the code more readable, but the annotations are also recognized by tools such as FindBugs and IntelliJ IDEA, which will use them to aid static analysis and thus find potential bugs.

To use these annotations, first add them to your project's dependencies:

```
<dependency>
    <groupId>com.google.code.findbugs</groupId>
    <artifactId>jsr305</artifactId>
    <version>3.0.0</version>
</dependency>
```

Once you've done that, you can add annotations such as `@Nonnull`, `@Nullable`, and `@CheckForNull` to your code. It's good to get into the habit of adding these annotations whenever you read through legacy code, both to aid your own understanding and to make life easier for the next reader. The following sample shows a method with JSR 305 annotations added.

```
@CheckForNull
public List<Player> findPlayersByName(@Nonnull String lastName,
                                      @Nullable String firstName) {
  ...
}
```

Here the `@CheckForNull` annotation means that the method might return `null` (perhaps if there are no matches or if an error occurred), the `@Nonnull` annotation means that the first parameter must not be `null`, and the `@Nullable` annotation means that it's OK to pass `null` as the second parameter.

Null in other languages

Languages other than Java treat `null` in different ways. Ruby, for example, has the `nil` object, which acts in a "falsey" way, so you often don't need to check whether a variable is `nil` before referencing it.

Regardless of the language, you can generally use the Null Object pattern, whereby you define your own object to represent the absence of a value, instead of relying on the language's built-in `null`. Wikipedia has some simple examples of the Null Object pattern in various languages here: https://en.wikipedia.org/wiki/Null_Object_pattern.

4.2.4 Needlessly mutable state

Unnecessary use of mutability ranks alongside overuse of `null` in terms of making code difficult to read and debug. In general, making objects immutable makes it easier for a developer to keep track of the state of a program. This is especially true in multithreaded programming—there's no need to worry about what happens when two threads try to alter the same object at the same time, because the object is immutable and can't be altered in the first place.

Mutable state is common in legacy Java code for a couple of reasons:

- *Historical*—Back in the day when Java Beans were cool, it was standard practice to make all model classes mutable, with getters and setters.
- *Performance*—Using immutable objects often results in more short-lived objects being created and destroyed. This object churn caused early Java GCs to struggle, but it usually isn't a problem for modern GCs such as HotSpot's G1.

Mutability certainly has its place (for example, modeling a system as a finite state machine is a useful technique that entails mutability), but I usually design code to be immutable by default, only introducing mutability if it makes the code easier to reason about or if profiling has shown the immutable code to be a performance bottleneck.

Taking an existing mutable class and making it immutable usually goes something like this.

1. Mark all fields as `final`.
2. Add constructor arguments to initialize all fields. You may also want to introduce a builder, as shown earlier in the chapter.
3. Update all setters to create a new version of the object and return it. You might want to rename the methods to reflect this change in behavior.
4. Update all client code to make it use instances of the class in an immutable fashion.

Imagine that players of World of RuneQuest can acquire and use magic spells. There are only a few different spells, and they're large, heavyweight objects, so for memory efficiency it would be nice if you could have only one singleton object in memory for each spell, and share them among many different players. However, the spells are currently implemented in a mutable fashion, whereby the `Spell` object keeps track of how many times its owner has used it, so you can't share a given spell object between multiple users. The following sample shows the current, mutable implementation of `Spell`.

```
class Spell {
    private final String name;
    private final int strengthAgainstOgres;
    private final int wizardry;
    private final int magicalness;

    private int timesUsed = 0;

    public void useOnce() {
        this.timesUsed += 1;
    }
}
```

Lots more fields

Only this field is mutable

Constructor, other methods ...

If, however, we move the `timesUsed` field out of `Spell`, the class will become completely immutable, and thus safe to share among all users. We could create a new class `SpellWithUsageCount` that holds the `Spell` instance and the usage count, as shown in the following sample. Note that the new `SpellWithUsageCount` class is also immutable.

```
class SpellWithUsageCount {
    public final Spell spell;
    public final int timesUsed;

    public SpellWithUsageCount(Spell spell, int timesUsed) {
        this.spell = spell;
        this.timesUsed = timesUsed;
```

Spell#timesUsed field has been removed

```
    }

    /**
     * Increment the usage count.
     * @return a copy of this object, with the usage count incremented by one
     */
    public SpellWithUsageCount useOnce() {
        return new SpellWithUsageCount(spell, timesUsed + 1);
    }
}
```

This is an improvement over the original code for a couple of reasons. First, we can now share the heavyweight `Spell` objects between all players in the system, with no danger of one player's actions accidentally affecting another player's state, so we can save a lot of memory. We're also safe from any potential concurrency bugs whereby two threads try to update the same `Spell` at the same time, resulting in corrupted state. Immutable objects are safe to share both between multiple objects and between threads.

Immutability in other languages

Mainstream languages other than Java provide differing degrees of support for immutability.

- C# has good support for immutability. It has the `readonly` keyword to mark a specific field as write-once (meaning it's immutable once it has been initialized), and anonymous types are an easy way to create immutable objects. The standard library also contains some immutable collections.
- Dynamic languages such as Python, Ruby, and PHP don't provide much support for immutability, and idiomatic code written in those languages tends to be written in a mutable style. Python at least provides the ability to "freeze" instances of some built-in types, such as `set`. For Ruby, Hamster (https://github.com/hamstergem/hamster) is a nice library of immutable collections.

4.2.5 Byzantine business logic

The business logic in legacy applications can often seem very complicated and difficult to follow. This is usually for a couple of reasons.

- The business rules really are complicated. Or rather, they started off simple and gradually became more complicated over time. Over the years that the system has been in production, more and more special cases and exemptions have been added.
- Business logic is intertwined with other processing such as logging and exception handling.

Let's look at an example. Imagine that World of RuneQuest generates some of its revenue from banner ads, and the following class is responsible for choosing a banner ad to display to a given player on a given page.

```
public class BannerAdChooser {
    private final BannerDao bannerDao = new BannerDao();
    private final BannerCache cache = new BannerCache();

    public Banner getAd(Player player, Page page) {
        Banner banner;
        boolean showBanner = true;

        // First try the cache
        banner = cache.get(player, page);

        if (player.getId() == 23759) {
            // This player demands not to be shown any ads.
            // See support ticket #4839
            showBanner = false;
        }

        if (page.getId().equals("profile")) {
            // Don't show ads on player profile page
            showBanner = false;
        }

        if (page.getId().equals("top") &&
            Calendar.getInstance().get(DAY_OF_WEEK) == WEDNESDAY) {
            // No ads on top page on Wednesdays
            showBanner = false;
        }

        if (player.getId() % 5 == 0) {
            // A/B test - show banner 123 to these players
            banner = bannerDao.findById(123);
        }

        if (showBanner && banner == null) {
            banner = bannerDao.chooseRandomBanner();
        }

        if (banner.getClientId() == 393) {
            if (player.getId() == 36645) {
                // Bad blood between this client and this player!
                // Don't show the ad.
                showBanner = false;
            }
        }

        // cache our choice for 30 minutes
        cache.put(player, page, banner, 30 * 60);

        if (showBanner) {
            // make a record of what banner we chose
            logImpression(player, page, banner);
        }

        return banner;
    }

}
```

Dozens more checks and conditions ...

All of these special cases that have accumulated over the years have made the method very long and unwieldy. They're all necessary, as far as we know, so we can't just delete them, but we can refactor the code to make it easier to read, test, and maintain. Let's combine a couple of standard design patterns, Decorator and Chain of Responsibility, to refactor the `BannerAdChooser`. The plan is as follows.

1. Use the Chain of Responsibility pattern to separate business rules into their own testable unit.
2. Use the Decorator pattern to separate the implementation details (caching and logging) from the business logic.

Once we're finished, a conceptual view of our code should look something like figure 4.8.

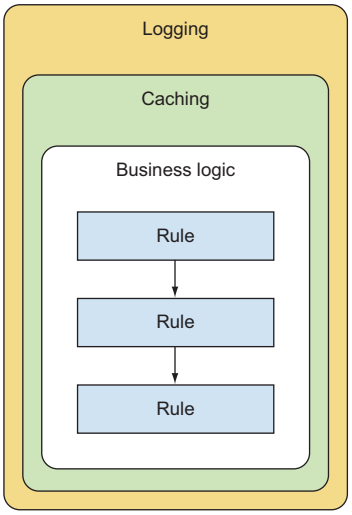


Figure 4.8 Our plan for refactoring the `BannerAdChooser` class using the Chain of Responsibility and Decorator patterns

First, we'll create an abstract `Rule` class that each of our business rules will extend. Each concrete subclass will have to implement two methods: one to decide whether the rule applies to a given player and page, and another to actually apply the rule.

```
abstract class Rule {
    private final Rule nextRule;

    protected Rule(Rule nextRule) {
        this.nextRule = nextRule;
    }

    /**
     * Does this rule apply to the given player and page?
     */
    abstract protected boolean canApply(Player player, Page page);

    /**
     * Apply the rule to choose a banner to show.
     * @return a banner, which may be null
     */
    abstract protected Banner apply(Player player, Page page);

    Banner chooseBanner(Player player, Page page) {
        if (canApply(player, page)) {
            // apply this rule
            return apply(player, page);
        } else if (nextRule != null) {
            // try the next rule
            return nextRule.chooseBanner(player, page);
        } else {
            // ran out of rules to try!
            return null;
        }
    }
}
```

Next, we'll write a concrete subclass of `Rule` for each of our business rules. I'll show a couple of examples.

```
final class ExcludeCertainPages extends Rule {

    // Pages on which banners should not be shown
    private static final Set<String> pageIds =
        new HashSet<>(Arrays.asList("profile"));

    public ExcludeCertainPages(Rule nextRule) {
        super(nextRule);
    }

    protected boolean canApply(Player player, Page page) {
        return pageIds.contains(page.getId());
    }

    protected Banner apply(Player player, Page page) {
        return null;
    }
}

final class ABTest extends Rule {
    private final BannerDao dao;

    public ABTest(BannerDao dao, Rule nextRule) {
        super(nextRule);
        this.dao = dao;
    }

    protected boolean canApply(Player player, Page page) {
        // check if player is in A/B test segment
        return player.getId() % 5 == 0;
    }

    protected Banner apply(Player player, Page page) {
        // show banner 123 to players in A/B test segment
        return dao.findById(123);
    }
}
```

Once we have our `Rule` implementations, we can chain them together into a Chain of Responsibility.

```
Rule buildChain(BannerDao dao) {
    return new ABTest(dao,
           new ExcludeCertainPages(
           new ChooseRandomBanner(dao)));
}
```

Only showing a few links of the chain, for brevity

Whenever we want to choose a banner to show, each rule will be tried in turn until a matching one is found.

Now that we have our business rules cleanly isolated from each other, the next step of our plan is to move the caching and logging code into decorators. First let's extract an interface from the `BannerAdChooser` class. Each of our decorators will implement this interface.

We'll use the name `BannerAdChooser` for the interface and rename the concrete class to `BannerAdChooserImpl`. (This is a horrible name, but we're about to replace this class anyway.)

```
interface BannerAdChooser {

    public Banner getAd(Player player, Page page);

}

final class BannerAdChooserImpl implements BannerAdChooser {

    public Banner getAd(Player player, Page page) {
        ...
    }

}
```

Next we'll split the method into a base case and a couple of decorators. The base case will be the main Chain of Responsibility-based implementation.

```
final class BaseBannerAdChooser implements BannerAdChooser {
    private final BannerDao dao = new BannerDao();
    private final Rule chain = createChain(dao);

    public Banner getAd(Player player, Page page) {
        return chain.chooseBanner(player, page);
    }
}
```

We'll also have decorators that transparently take care of caching and logging respectively.

The following code shows a decorator for wrapping the existing banner ad logic with caching. When asked for an ad, it first checks if it already has an appropriate ad in its cache. If so, it returns it. Otherwise, it delegates the choice of ad to the underlying `BannerAdChooser`, and then caches the result.

```
final class CachingBannerAdChooser implements BannerAdChooser {
    private final BannerCache cache = new BannerCache();
    private final BannerAdChooser base;

    public CachingBannerAdChooser(BannerAdChooser base) {
        this.base = base;
    }

    public Banner getAd(Player player, Page page) {
        Banner cachedBanner = cache.get(player, page);
        if (cachedBanner != null) {
            return cachedBanner;
        } else {
            // Delegate to next layer
            Banner banner = base.getAd(player, page);
            // Store the result in the cache for 30 minutes
            cache.put(player, page, banner, 30 * 60);
            return banner;
        }
    }
}
```

The next code segment shows another decorator, this time for adding logging. The choice of ad is delegated to the underlying `BannerAdChooser`, and then the result is logged before being returned to the caller.

```
final class LoggingBannerAdChooser implements BannerAdChooser {
    private final BannerAdChooser base;

    public LoggingBannerAdChooser(BannerAdChooser base) {
        this.base = base;
    }

    public Banner getAd(Player player, Page page) {
        // Delegate to next layer
        Banner banner = base.getAd(player, page);
        if (banner != null) {
            // Make a record of what banner we chose
            logImpression(player, page, banner);
        }
        return banner;
    }

    private void logImpression(...) {
        ...
    }
}
```

Finally, we need a factory to take care of wiring up all our decorators in the correct order.

```
final class BannerAdChooserFactory {

    public static final BannerAdChooser create() {
        return new LoggingBannerAdChooser(
                  new CachingBannerAdChooser(
                    new BaseBannerAdChooser()));
    }

}
```

Now that we've separated each business rule into a separate class and separated the implementation concerns of caching and logging from the business logic, the code should be easier to read, maintain, and extend. Both the Chain of Responsibility and Decorator patterns make it very easy to add, remove, or reorder layers as needed. Also, each business rule and implementation concern can now be tested in isolation, which was not possible before.

4.2.6 Complexity in the view layer

The Model-View-Controller pattern is commonly used in applications that provide a GUI, especially web applications. In theory, all business logic is kept out of the view and encapsulated inside the model, while the controller takes care of the details of accepting user input and manipulating the model.

In practice, however, it's easy for logic to infect the view layer. This is often a consequence of trying to reuse the same model for multiple purposes. For example, a model that's designed for easy serialization to a relational database will directly reflect the DB schema, but this model is probably not suitable for being passed as-is to the view layer. If you try to do so, you'll end up having to put a lot of logic into the view layer to transform the model into a form suitable for showing to the user.

This accumulation of logic in the view layer is a problem for a few reasons:

- The technologies used in the view layer (such as JSP in a Java web application) are usually not amenable to automated testing, so the logic contained within them can't be tested.
- Depending on the technology used, the files in the view layer might not be compiled, so errors can't be caught at compile time.
- You might want people such as visual designers or front-end engineers to work on the view layer, but this is difficult if the markup is interspersed with snippets of source code.

We can alleviate these problems by introducing a transformation layer between the model and the view. This layer, as shown in figure 4.9, is sometimes called a *presentation model* or a *ViewModel*, but I tend to call it a *view adapter*. By moving the logic out of the view and into the view adapter, we can simplify the view templates, making them more readable and easier to maintain. This also makes the transformation logic easier to test, because the view adapters are plain old objects, with no dependencies on the view technology, and can thus be tested just like any other source code.

Let's look at an example. World of RuneQuest has a `CharacterProfile` object that holds information about a player's character: name, species, special skills, and so on.

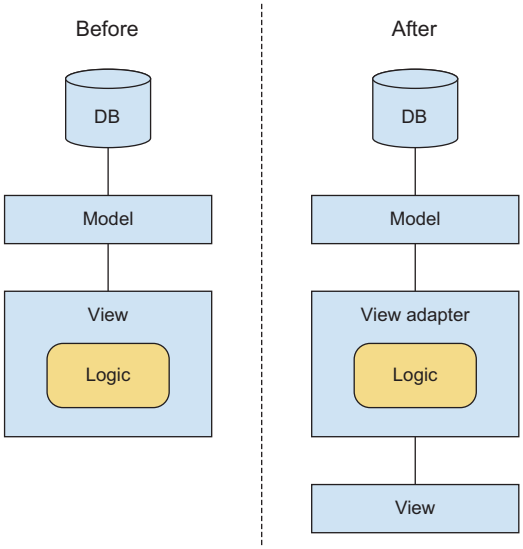


Figure 4.9 Introducing a view adapter

This model is passed to a JSP in order to render the character profile page. The `CharacterProfile` is shown here.

```
class CharacterProfile {
    String name;
    Species species;
    DateTime createdAt;
       ...
}
```

The following code is a snippet of the JSP.

```
<table>
  <tr>
    <td>Name</td>
    <td>${profile.name}</td>
  </tr>

  <c:choose>
    <c:when test="${species.name == 'orc'>
      <c:set var="speciesTextColor" value="brown" />
    </c:when>
    <c:when test="${species.name == 'elf'>
      <c:set var="speciesTextColor" value="green" />
    </c:when>
    <c:otherwise>
      <c:set var="speciesTextColor" value="black" />
    </c:otherwise>
  </c:choose>
  <tr>
    <td>Species</td>
    <td style="color: $speciesTextColor">${profile.species.name}</td>
  </tr>

  <%
    CharacterProfile profile = (CharacterProfile)(request.getAttribute("profi
     le"));
    DateTime today = new DateTime();
    Days days = Days.daysBetween(profile.createdAt, today);
    request.setAttribute("ageInDays", days.getDays());
  %>
  <tr>
    <td>Age</td>
    <td>${ageInDays} days</td>
  </tr>
</table>
```

This JSP is horrible! It has logic jumbled together with presentation, making it very hard to read. Let's introduce a view adapter and pass that to the JSP, instead of passing the `CharacterProfile` model directly.

In the following code, I've extracted all the logic from the JSP and put it into a view adapter. I could have called the class `CharacterProfileViewAdapter`, but that's a bit of a mouthful. For brevity's sake I usually follow the view adapter class for a `Foo` model as `FooView`.

```
class CharacterProfileView {
    private final CharacterProfile profile:

    public CharacterProfileView(CharacterProfile profile) {
        this.profile = profile;
    }

    public String getName() {
        // return the underlying model's property as is
        return profile.getName();
    }

    public String getSpeciesName() {
        return profile.getSpecies().getName();
    }

    public String getSpeciesTextColor() {
        if (profile.getSpecies().getName().equals("orc")) {
            return "brown";
        } else if (profile.getSpecies().getName().equals("elf")) {
            return "green";
        } else {
            return "black";
        }
    }

    public int getAgeInDays() {
        DateTime today = new DateTime();
        Days days = Days.daysBetween(profile.createdAt, today);
        return days.getDays();
    }

        ...
}
```

The next code snippet shows how the JSP looks when we make use of the view adapter.

```
<table>
  <tr>
    <td>Name</td>
    <td>${profile.name}</td>
  </tr>
  <tr>
    <td>Species</td>
    <td style="color: ${profile.speciesTextColor}">${profile.speciesName}</td
     >
  </tr>
  <tr>
    <td>Age</td>
    <td>${profile.ageInDays} days</td>
  </tr>
</table>
```

There, that's better! The logic is now contained in a testable Java class, and the template is much more readable than before.

It's worth noting that if you don't trust yourself to keep logic out of the view layer, you can force yourself to do by choosing a logic-less template technology for your

views. I've had success with logic-less template languages such as Mustache for building simple, readable views for web applications. The templates can be written and maintained by web designers, allowing the developers to focus on the business logic.

APPLICABILITY TO OTHER LANGUAGES The View Adapter pattern is not specific to Java and JSP templates. It's useful no matter whether you're using Ruby and ERB, ASP.NET, or any other technology. Whenever you have an application with a UI of some kind, you can and should keep complex logic out of the view layer.

Further reading

I've just scratched the surface in this brief foray into refactoring. If you'd like to learn more about refactoring, there are plenty of excellent books dedicated to the subject. Here are three recommendations.

- *Refactoring: Improving the Design of Existing Code* by Martin Fowler et al. (Addison-Wesley Professional, 1999). Although it's getting a little dated (it was written when Java was at version 1.2), it's a classic and still a great reference. It takes a pattern-based approach, describing in what situation you might want to use a particular refactoring.
- *Refactoring to Patterns* by Joshua Kerievsky (Addison-Wesley Professional, 2004). This book cleverly shows how you can take legacy code that lacks a clear structure, and migrate it toward well-known design patterns using the refactorings described in Martin Fowler's book. If you want to brush up on design patterns before reading this book, read *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional, 1994), also known as the Gang of Four book.
- *Principle-Based Refactoring* by Steve Halladay (Principle Publishing, 2012). This book is full of useful refactoring techniques, but it takes more of a "teach a man to fish" approach, promoting the value of studying the underlying principles of software design rather than slavishly learning dozens of rules by rote.

4.3 Testing legacy code

When refactoring legacy code, automated tests can provide valuable assurances that the refactoring has not inadvertently affected the behavior of the software. In this section I'll talk about how to write these automated tests, and what to do when you're faced with untestable code.

4.3.1 Testing untestable code

Before you start refactoring, you want to have unit tests in place. But before you can write unit tests, you need to refactor the code to make it testable. But before you start refactoring, you want to have unit tests in place ...

This chicken-and-egg situation is something we often face when trying to retroactively add tests to legacy code. If we insist on having unit tests in place before refactoring, then it seems like an unbreakable paradox. But it's worth remembering that if we can manage to get a few tests in place, we can start refactoring, making the software more testable, allowing us to write more tests, in turn allowing more refactoring, and so on.

Think of it like peeling an orange. At first it seems perfectly round and impenetrable, but once you apply a little force to break the skin, the thing practically peels itself. So we need to lower our standards temporarily in order to break the skin and get our first few tests in place. When we do this, we can use code review to make up for the lack of tests.

Our first priority when trying to make code testable is to isolate it from its dependencies. We want to replace all of the objects that the code interacts with, with objects that we control. That way we can feed it whatever inputs we like and measure its response, whether it consists of returning a value or calling methods on other objects.

Let's look at an example of how we can refactor a piece of legacy code in order to get it into a test harness. Imagine we want to write tests for the `Battle` class, which is used when two players fight each other in World of RuneQuest. Unfortunately, `Battle` is riddled with dependencies on a so-called "God class," a 3,000-line monster of a class called `Util`. This class is filled with static methods that do all kinds of useful things, and it's referenced from all over the place.

BEWARE THE UTIL Whenever you see a class with `Util` in the name, alarm bells should start ringing in your head. It may well be a good candidate for refactoring, if only to rename it to something more meaningful.

Here's how the code looks before we start.

```
public class Battle {
    private BattleState = new BattleState();
    private Player player1, player2;

    public Battle(Player player1, Player player2) {
        this.player1 = player1;
        this.player2 = player2;
    }

        ...

    public void registerHit(Player attacker, Weapon weapon) {
        Player opponent = getOpponentOf(attacker);
        int damageCaused = calculateDamage(opponent, weapon);
        opponent.setHealth(opponent.getHealth() - damageCaused);

        Util.updatePlayer(opponent);

        updateBattleState();
    }
```

```
    public BattleState getBattleState() {
        return battleState;
    }

    ...
}
```

The class has a nice, simple constructor, so we can easily construct an instance in order to test it. It also has a public method exposing its internal state, which should be useful for our tests. We have no idea what that suspicious call to `Util.updatePlayer(opponent)` is doing, but let's ignore it for now and try writing a test.

```
public class BattleTest {

    @Test
    public void battleEndsIfOnePlayerAchievesThreeHits() {
        Player player1 = ...;
        Player player2 = ...;
        Weapon axe = new Axe();
        Battle battle = new Battle(player1, player2);

        battle.registerHit(player1, axe);
        battle.registerHit(player1, axe);
        battle.registerHit(player1, axe);

        BattleState state = battle.getBattleState();
        assertThat(state.isFinished(), is(true));
    }

}
```

OK, let's run the test, and ... whoops! It turns out that the `Util.updatePlayer(player)` method not only writes the `Player` object to a database, it may also send an email to the user to inform them that their character is unhealthy/lonely/running out of gold. These are side effects that we definitely want to avoid in our tests. Let's see how we can fix this.

Because the `Battle` class's dependency is on a static method, we can't use any tricks such as subclassing `Util` and overriding the method. Instead, we'll have to create a new class with a method that wraps the static method call, and then have `Battle` call the method on the new class. In other words, we'll introduce a layer of indirection between `Battle` and `Util`. In tests, we'll be able to substitute our own implementation of this buffer class, thus avoiding any unwanted side effects.

First, let's create an interface.

```
interface PlayerUpdater {

    public void updatePlayer(Player player);

}
```

We'll also create an implementation of this interface for use in production code:

```
public class UtilPlayerUpdater implements PlayerUpdater {

    @Override
    public void updatePlayer(Player player) {
        Util.updatePlayer(player);
    }

}
```

We now need a way to pass the `PlayerUpdater` to `Battle`, so let's add a constructor parameter. Notice how we create a `protected` constructor for use in tests, and we avoid changing the signature of the existing `public` constructor.

```
public class Battle {
    private BattleState = new BattleState();
    private Player player1, player2;
    private final PlayerUpdater playerUpdater;

    public Battle(Player player1, Player player2) {
        this(player1, player2, new UtilPlayerUpdater());
    }

    protected Battle(Player player1, Player player2,
                     PlayerUpdater playerUpdater) {
        this.player1 = player1;
        this.player2 = player2;
        this.playerUpdater = playerUpdater;
    }

        ...

    public void registerHit(Player attacker, Weapon weapon) {
        Player opponent = getOpponentOf(attacker);
        int damageCaused = calculateDamage(opponent, weapon);
        opponent.setHealth(opponent.getHealth() - damageCaused);

        playerUpdater.updatePlayer(opponent);

        updateBattleState();
    }

        ...

}
```

PROTECTED METHODS IN JAVA Because we added the new constructor with `protected` visibility, it will only be visible to subclasses of `Battle` or classes in the same package. We should put our test class in the same package as `Battle` so that it can call the constructor we added.

So far we've made changes to the `Battle` class, but we think we've maintained the existing behavior. Now is the time to pause, commit what we have so far, and ask a colleague for a code review, just to check that we haven't done anything silly. Once that's complete, we can move on to fixing our test.

In the test, we could create a dummy implementation of `PlayerUpdater` that does nothing, and pass it to the `Battle` constructor. But actually we can do better. If we use

a mock implementation, we can also check that `Battle` calls our `updatePlayer()` method as we expected. Let's use the Mockito library (http://mockito.github.io/) to create our mock implementation.

```
import static org.mockito.Mockito.*;

public class BattleTest {

    @Test
    public void battleEndsIfOnePlayerAchievesThreeHits() {
        Player player1 = ...;
        Player player2 = ...;
        Weapon axe = new Axe();

        PlayerUpdater updater = mock(PlayerUpdater.class);

        Battle battle = new Battle(player1, player2, updater);

        battle.registerHit(player1, axe);
        battle.registerHit(player1, axe);
        battle.registerHit(player1, axe);

        BattleState state = battle.getBattleState();
        assertThat(state.isFinished(), is(true));

        verify(updater, times(3)).updatePlayer(player2);
    }

}
```

Creates a mock implementation of PlayerUpdater interface

Passes the mock to the Battle instance

Checks that the updatePlayer() method was called 3 times

Yay, we've broken the skin and our first working test is in place. Not only did we break the dependency on the `Util` class, we also managed to verify the test subject's interactions with other classes.

> **Further reading**
>
> For a whole book dedicated to examples like this one, as well as detailed explanations of the reasoning behind the approach taken in each case, I highly recommend *Working Effectively with Legacy Code* by Michael Feathers (Prentice Hall, 2004).

4.3.2 Regression testing without unit tests

Here's a deliberately inflammatory statement for you:

Writing unit tests before refactoring is sometimes impossible and often pointless.

Of course I'm exaggerating, but I wanted to get two points across here:

- "Sometimes impossible" is a reference to the difficulty of retroactively adding unit tests to legacy code that wasn't designed with testability in mind, as seen in the previous section. Although you can try to exploit a seam in order to inject mocks and stubs and test a piece of code in isolation, in practice this often requires a lot of effort.

- Writing tests is "often pointless" because refactoring isn't always restricted to an individual unit (a single class, in object-oriented languages). If your refactoring affects multiple units, then the very refactoring that you're about to perform can wipe out the value of the unit tests you're writing.

  For example, if you're about to perform a refactoring that combines existing classes `A` and `B` into a new class `C`, then there's little point in writing tests for `A` and `B` beforehand. As part of the refactoring, `A` and `B` will be deleted, so their tests will no longer compile, and you'll have to write tests for the newly created class `C` anyway.

UNIT TESTS ARE NOT A SILVER BULLET

If your refactoring is going to break unit tests, you need to have a backup—functional tests for the module that contains those units. Likewise, if you're planning a larger-scale refactoring of a whole module, then you need to be prepared for your refactoring to break all of that module's tests. You'll need to have tests in place at a higher level, that can survive the refactoring. As a general rule, you should make sure that you have tests at a level of modularity *one level higher* than the code that will be affected by your refactoring.

For this reason, it's important to build up a suite of tests at multiple levels of modularity (see figure 4.10). When working with legacy code that wasn't designed for testability, it's often easiest to start at the outside, writing system tests, and then working your way in as far as you can.

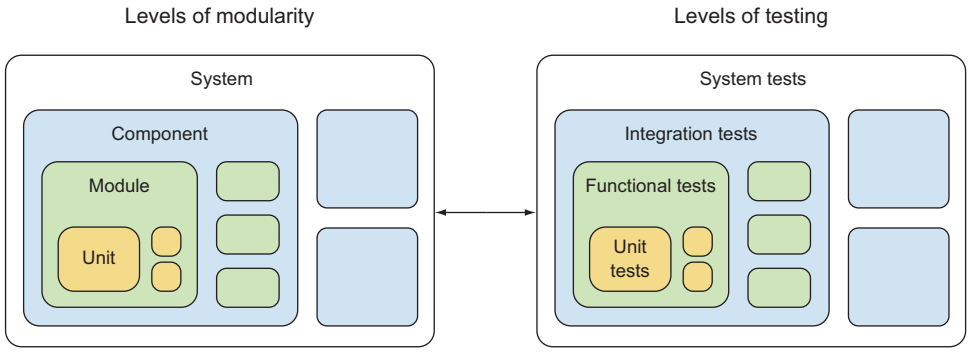


Figure 4.10 Levels of modularity and their corresponding tests

DON'T GET HUNG UP ON COVERAGE

Because test coverage is easy to measure, and increasing it is quite a satisfying pastime, it's easy to focus on it too much. But when you've inherited code with a very low test coverage, and you're trying to add tests retroactively to untestable code, getting test coverage up to what you consider an acceptable level can require a mammoth effort. I've seen a number of teams inherit code with test coverage of less than 10%, spend

weeks trying to increase the coverage, and then give up at around 20% with no noticeable improvement in quality or maintainability. (I also saw a team inherit a large C# codebase with no tests, and 18 months later they had achieved their goal of 80% coverage, so there are exceptions to every rule!)

The problem with setting an arbitrary goal of improved test coverage is that you'll start by writing the easiest tests first. In other words, you'll write dozens of tests for

- Code that happens to be easily testable, neglecting more important but less testable parts of the codebase
- Code that is well written and easy to reason about, even though a code review might be enough to verify that this code works as expected

AUTOMATE ALL YOUR TESTS

Most developers would agree that unit tests should be fully automated, but the level of automation for other kinds of tests (such as integration tests) is often much lower. Although we want to run these tests as often as possible when refactoring, in order to spot regressions quickly, we can't do so if they rely on manual labor. Even if we had an army of willing testers to rerun the whole integration test suite every time we made a commit, it's possible that they would forget to run a test, or misinterpret the results. Plus it would slow down the development cycle. Ideally we want all of our regression tests, not just the unit tests, to be 100% automated.

One area that cries out for automation is UI testing. Whether you're testing a desktop application, a website, or a smartphone app, there's a huge selection of tools available to help you automate your tests. For example, tools such as Selenium and Capybara make it easy to write automated web UI tests. The following code sample shows a Capybara script that you could use to test World of RuneQuest's player profile page, which you saw earlier in the chapter. This simple Ruby script opens a web browser, logs in to World of RuneQuest, opens the My Profile page, and checks that it contains the correct content, all within a matter of seconds.

```
require "rspec"
require "capybara"
require "capybara/dsl"
require "capybara/rspec"

Capybara.default_driver = :selenium
Capybara.app_host = "http://localhost:8080"

describe "My Profile page", :type => :feature do

  it "contains character's name and species" do
    visit "/"
    fill_in "Username", :with => "test123"
    fill_in "Password", :with => "password123"
    click_button 'Login'

    visit "/profile"
    expect(find("#playername")).to have_content "Test User 123"
    expect(find("#speciesname")).to have_content "orc"
  end

end
```

Login as a known test user

Opens the 'My Profile' page and checks its content

This test can easily be run by a developer on their local machine or by a CI server such as Jenkins. It could also be configured to run in headless mode instead of opening and manipulating a web browser, in order to speed up the test execution.

Of course, it's not possible to test everything in your application using UI tests alone, but they make a valuable addition to your test suite, especially when working with legacy code that may be difficult to test by other means.

4.3.3 Make the users work for you

You've done pair programming, you've conducted code reviews, you've run your unit tests, functional tests, integration tests, system tests, UI tests, performance tests, load tests, smoke tests, fuzz tests, wobble tests (OK, I made that last one up), and they've all passed. This means your software has no bugs, right?

No, of course not! No matter how much testing you do, there will always be a pattern you haven't managed to check. Every test that you run before release is, in a sense, an attempt to emulate the actions of a typical user, based on your best guess about how users will use the software. But the quality and rigor of this simulacrum will never match the real thing—the users themselves. So why not put this army of unwitting testers to good use?

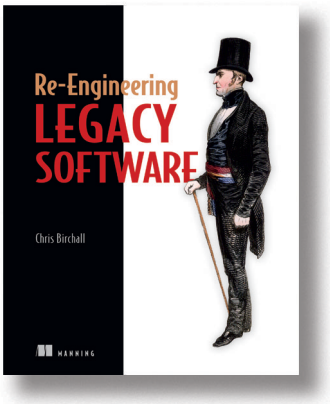
You can make use of user data to help ensure the quality of your software in a few ways.

- *Perform gradual rollouts of new releases, while monitoring for errors and regressions.* If you start to see unusually high error counts, you can stop the rollout, investigate the cause, and either roll back to the previous version or fix the problem before continuing the release. Of course, error monitoring and subsequent rollback can be automated. Google is one company known for its dedication to gradual rollouts, with major releases of Android taking many weeks to reach all devices.
- *Gather real user data and use it to make your tests more productive.* When load testing a web application, it's difficult to generate traffic that reflects real usage patterns, so why not record the traffic of a few real users and feed that into your test scripts?
- *Perform stealth releases of new versions, whereby software is released into the production environment but not yet visible to the users.* All traffic is sent to both the old and new versions, so you can see how the new version works against real user data.

4.4 Summary

- Successful refactoring takes discipline. Perform refactorings in a structured way and avoid combining them with other work.
- Removal of stale code and low-quality tests is a nice way to get the refactoring ball rolling.
- Use of `null` pointers is a very common source of bugs, no matter what language you're using.

- Prefer immutable state over mutable.
- Use standard design patterns to separate business logic from implementation details or to make complex business logic more manageable and composable.
- Use the View Adapter pattern to keep complex logic out of your application's view layer.
- Beware any class or module with `Util` in the name.
- Introduce a layer of indirection in order to inject mock dependencies in tests.
- Unit tests aren't a silver bullet. You need tests at multiple levels of abstraction to protect against regressions caused by refactoring.
- Automate as many tests as possible—not only the unit tests.



As a developer, you may inherit projects built on existing codebases with design patterns, usage assumptions, infrastructure, and tooling from another time and another team. Fortunately, there are ways to breathe new life into legacy projects so you can maintain, improve, and scale them without fighting their limitations.

Re-Engineering Legacy Software is an experience-driven guide to revitalizing inherited projects. It covers refactoring, quality metrics, toolchain and workflow, continuous integration, infrastructure automation, and organizational culture. You'll learn techniques for introducing dependency injection for code modularity, quantitatively measuring quality, and automating infrastructure. You'll also develop practical processes for deciding whether to rewrite or refactor, organizing teams, and convincing management that quality matters. Core topics include deciphering and modularizing awkward code structures, integrating and automating tests, replacing outdated build systems, and using tools like Vagrant and Ansible for infrastructure automation.

What's inside:

- Refactoring legacy codebases
- Continuous inspection and integration
- Automating legacy infrastructure
- New tests for old code
- Modularizing monolithic projects

This book is written for developers and team leads comfortable with an OO language like Java or C#.

Identifying and Scoping Microservices

Microservices are more than just the latest buzzword. They allow independent scaling, enforce componentization, and fix many of the issues associated with monolithic n-tier architectures. In this chapter from *Microservices in .NET Core* by Christian Horsdal Gammelgaard, you'll learn how to carve out microservices from a design or an existing application. Whether refactoring legacy software or building a new application, microservices offer a lot of advantages that are introduced in this selection.

Chapter 3 from *Microservices in .NET Core*
by Christian Horsdal Gammelgaard.

Identifying and scoping microservices

This chapter covers

- Scoping microservices for business capability
- Scoping microservices to support technical capabilities
- Managing when scoping microservices is difficult
- Carving out new microservices from existing ones

To succeed with microservices, it's important to be good at scoping each microservice appropriately. If your microservices are too big, the turnaround on creating new features and implementing bug fixes becomes too long. If they're too small, the coupling between microservices tends to grow. If they're the right size but have the wrong boundaries, coupling also tends to grow, and higher coupling leads to longer turnaround. In other words, if you aren't able to scope your microservices correctly, you'll lose much of the benefit microservices offer. In this chapter, I'll teach you how to find a good scope for each microservice so they stay loosely coupled.

The primary driver in identifying and scoping microservices is business capabilities; the secondary driver is supporting technical capabilities. Following these two

drivers leads to microservices that align nicely with the list of microservice characteristics from chapter 1:

- A microservice is responsible for a single capability.
- A microservice is individually deployable.
- A microservice consists of one or more processes.
- A microservice owns its own data store.
- A small team can maintain a handful of microservices.
- A microservice is replaceable.

Of these characteristics, the first two and last two can only be realized if the microservice's scope is good. There are also implementation-level concerns that come into play, but getting the scope wrong will prevent the service from adhering to those four characteristics.

3.1 The primary driver for scoping microservices: business capabilities

Each microservice should implement exactly one capability. For example, a Shopping Cart microservice should keep track of the items in the user's shopping cart. The primary way to identify capabilities for microservices is to analyze the business problem and determine the business capabilities. Each business capability should be implemented by a separate microservice.

3.1.1 What is a business capability?

A *business capability* is something an organization does that contributes to business goals. For instance, handling a shopping cart on an e-commerce website is a business capability that contributes to the broader business goal of allowing users to purchase items. A given business will have a number of business capabilities that together make the overall business function.

When mapping a business capability to a microservice, the microservice models the business capability. In some cases, the microservice implements the entire business capability and automates it completely. In other cases, the microservice implements only part of the business capability and thus only partly automates it. In both cases, the scope of the microservice is the business capability.

Business capabilities and bounded contexts

Domain-driven design is an approach to designing software systems that's based on modeling the business domain. An important step is identifying the language used by domain experts to talk about the domain. It turns out that the language used by domain experts isn't consistent in all cases.

(continued)

In different parts of a domain, different things are in focus, so a given word like *customer* may have different focuses in different parts of the domain. For instance, for a company selling photocopiers, a *customer* in the sales department may be a company that buys a number of photocopiers and may be primarily represented by a procurement officer. In the customer service department, a *customer* may be an end user having trouble with a photocopier. When modeling the domain of the photocopier company, the word *customer* means different things in different parts of the model.

A *bounded context* in domain-driven design is a part of a larger domain within which words mean the same things. Bounded contexts are related to but different from business capabilities. A bounded context defines an area of a domain within which the language is consistent. Business capabilities, on the other hand, are about what the business needs to get done. Within one bounded context, the business may need to get several things done. Each of these things is likely a business capability.

3.1.2 Identifying business capabilities

A good understanding of the domain will enable you to understand how the business functions. Understanding how the business functions means you can identify the business capabilities that make up the business and the processes involved in delivering the capabilities. In other words, the way to identify business capabilities is to learn about the business's domain. You can gain this type of knowledge by talking with the people who know the business domain best: business analysts, the end users of your software, and so on—all the people directly involved in the day-to-day work that drives the business.

A business's organization usually reflects its domain. Different parts of the domain are handled by different groups of people, and each group is responsible for delivering certain business capabilities; so, this organization can give you hints about how the microservices should be scoped. For one thing, a microservice's responsibility should probably lie within the purview of only one group. If it crosses the boundary between two groups, it's probably too widely scoped and will be difficult to keep cohesive, leading to low maintainability. These observations are in line with what is known as *Conway's Law*:[1]

> *Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.*

Sometimes you may uncover parts of the domain where the organization and the domain are at odds. In such situations, there are two approaches you can take, both of which respect Conway's Law. You can accept that the system can't fully reflect the domain, and implement a few microservices that aren't well aligned with the domain but are well aligned with the organization; or you can change the organization to reflect the domain. Both approaches can be problematic. The first risks building

[1] Melvin Conway, "How Do Committees Invent?" *Datamation Magazine* (April 1968).

microservices that are poorly scoped and that might become highly coupled. The second involves moving people and responsibilities between groups. Those kinds of changes can be difficult. Your choice should be a pragmatic one, based on an assessment of which approach will be least troublesome.

To get a better understanding of what business capabilities are, it's time to look at an example.

3.1.3 Example: point-of-sale system

The example we'll explore in this chapter is a point-of-sale system, illustrated in figure 3.1. I'll briefly introduce the domain, and then we'll look at how to identify business capabilities within it. Finally, we'll consider in more detail the scope of one of the microservices in the system.

This point-of-sale system is used in all the stores of a large chain. Cashiers at the stores interact with the system through a thin GUI client—it could be a tablet application, a web application, or a purpose-built till (or register, if you prefer). The GUI client is just a thin layer in front of the backend. The backend is where all the business logic (the business capabilities) is implemented, and it will be our focus.

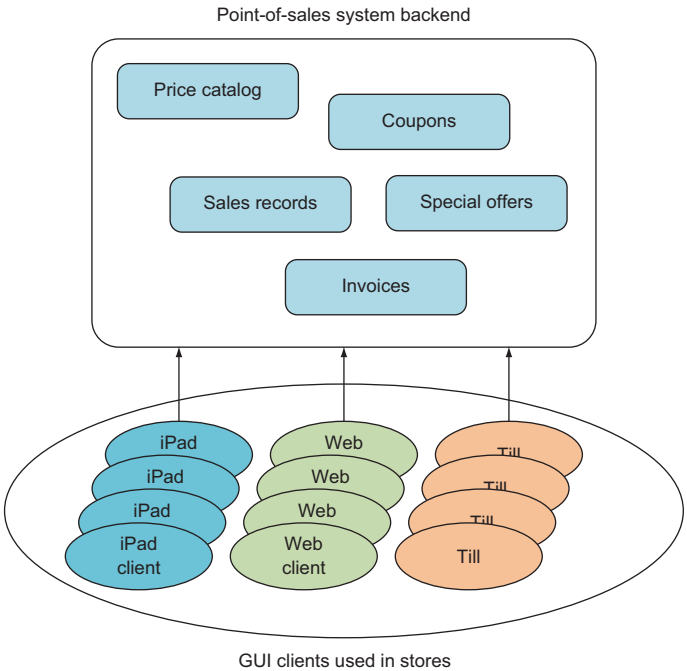


Figure 3.1 A point-of-sale system for a large chain of stores, consisting of a backend that implements all the business capabilities in the system and thin GUI clients used by cashiers in the stores. Microservices in the backend implement the business capabilities.

The system offers cashiers a variety of functions:

- Scan products and add them to the invoice
- Prepare an invoice
- Charge a credit card via a card reader attached to the client
- Register a cash payment
- Accept coupons
- Print a receipt
- Send an electronic receipt to the customer
- Search in the product catalog
- Scan one or more products to show prices and special offers related to the products

These functions are things the system does for the cashier, but they don't directly match the business capabilities that drive the point-of-sale system.

IDENTIFYING BUSINESS CAPABILITIES IN THE POINT-OF-SALE DOMAIN

To identify the business capabilities that drive the point-of-sale system, you need to look beyond the list of functions. You must determine what needs to go on behind the scenes to support the functionality.

Starting with the "Search in the product catalog" function, an obvious business capability is maintaining a product catalog. This is the first candidate for a business capability that could be the scope of a microservice. Such a Product Catalog microservice would be responsible for providing access to the current product catalog. The product catalog needs to be updated every so often, but the chain of stores uses another system to handle that functionality. The Product Catalog microservice would need to reflect the changes made in that other system, so the scope of the Product Catalog microservice would include receiving updates to the product catalog.

The next business capability you might identify is applying special offers to invoices. Special offers give the customer a discounted price when they buy a bundle of products. A bundle may consist of a certain number of the same product at a discounted price (for example, three for the price of two) or may be a combination of different products (say, buy A and get 10% off B). In either case, the invoice the cashier gets from the point-of-sale GUI client must take any applicable special offers into account automatically. This business capability is the second candidate to be the scope for a microservice. A Special Offers microservice would be responsible for deciding when a special offer applies and what the discount for the customer should be.

Looking over the list of functionality again, notice that the system should allow cashiers to "Scan one or more products to show prices and special offers related to the products." This indicates that there's more to the Special Offers business capability than just applying special offers to invoices: it also includes the ability to look up special offers based on products.

If you continued the hunt for business capabilities in the point-of-sale system, you might end up with this list:

- Product Catalog
- Price Catalog
- Price Calculation
- Special Offers
- Coupons
- Sales Records
- Invoice
- Payment

Figure 3.2 shows a map from functionalities to business capabilities. The map is a logical one, in the sense that it shows which business capabilities are needed to implement each function, but it doesn't indicate any direct technical dependencies. For instance, the arrow from Prepare Invoice to Coupons doesn't indicate a direct call from some Prepare Invoice code in a client to a Coupons microservice. Rather, the arrow indicates that in order to prepare an invoice, coupons need to be taken into account, so the Prepare Invoice function depends on the Coupons business capability.

I find creating this kind of map to be enlightening, because it forces me to think explicitly about how each function is attained and also what each business capability

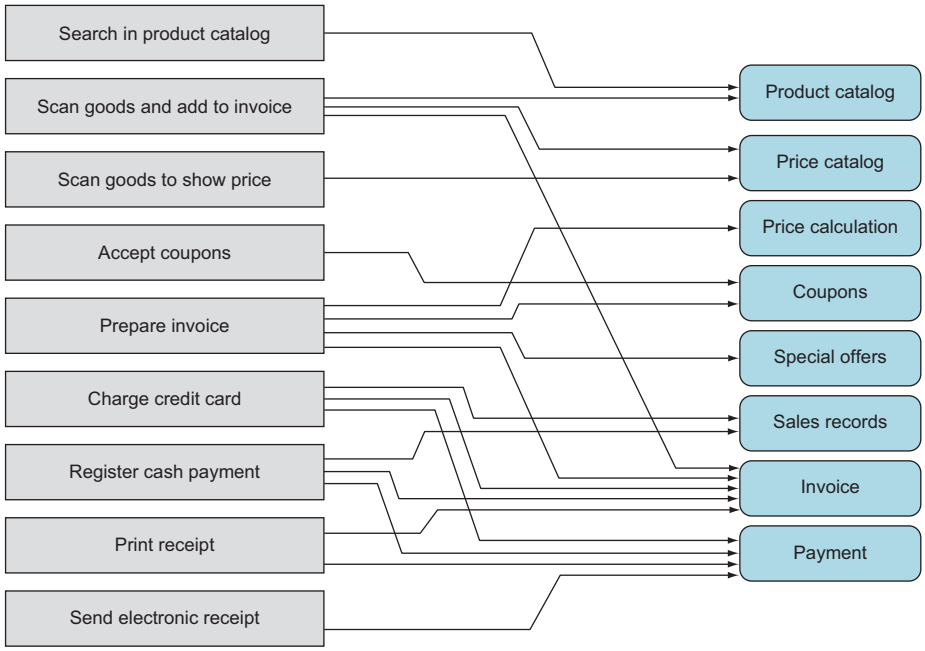


Figure 3.2 The functions on the left depend on the business capabilities on the right. Each arrow indicates a dependency between a function and a capability.

must do. Finding the business capabilities in real domains can be hard work and often requires a good deal of iterating. The list of business capabilities isn't a static list made at the start of development; rather, it's an emergent list that grows and changes over time as your understanding of the domain and the business grows and deepens.

Now that we've gone through the first iteration of identifying business capabilities, let's take a closer look at one of these capabilities and how it defines the scope of a microservice.

THE SPECIAL OFFERS MICROSERVICE

The Special Offers microservice is based on the Special Offers business capability. To narrow the scope of this microservice, we'll dive deeper into this business capability and identify the processes involved, illustrated in figure 3.3. Each process delivers part of the business capability.

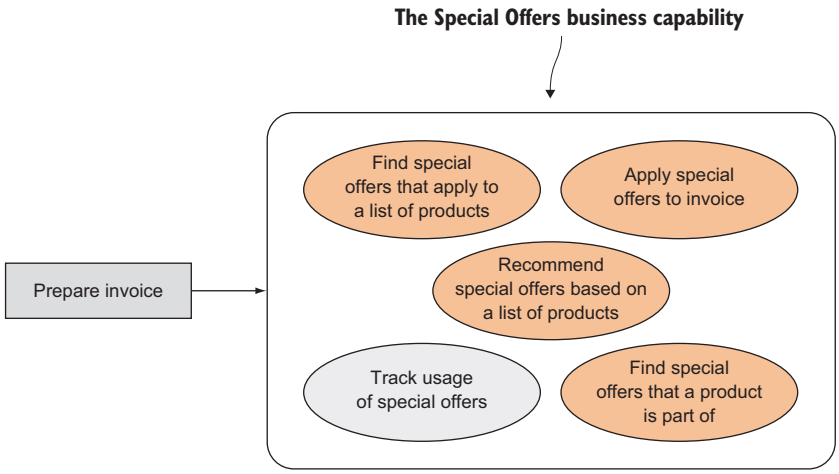


Figure 3.3 The Special Offers business capability includes a number of different processes.

The Special Offers business capability is broken down into five processes. Four of these are oriented toward the point-of-sale GUI clients. The fifth—tracking the use of special offers—is oriented toward the business itself, which has an interest in which special offers customers are taking advantage of.

Implementing the business capability as a microservice means you need to do the following:

- Expose the four client-oriented processes as API endpoints that other microservices can call.
- Implement the usage-tracking process through an event feed. The business-intelligence parts of the point-of-sale system can subscribe to these events and use them to track which special offers are used by customers.

The components of the Special Offers microservice are shown in figure 3.4.

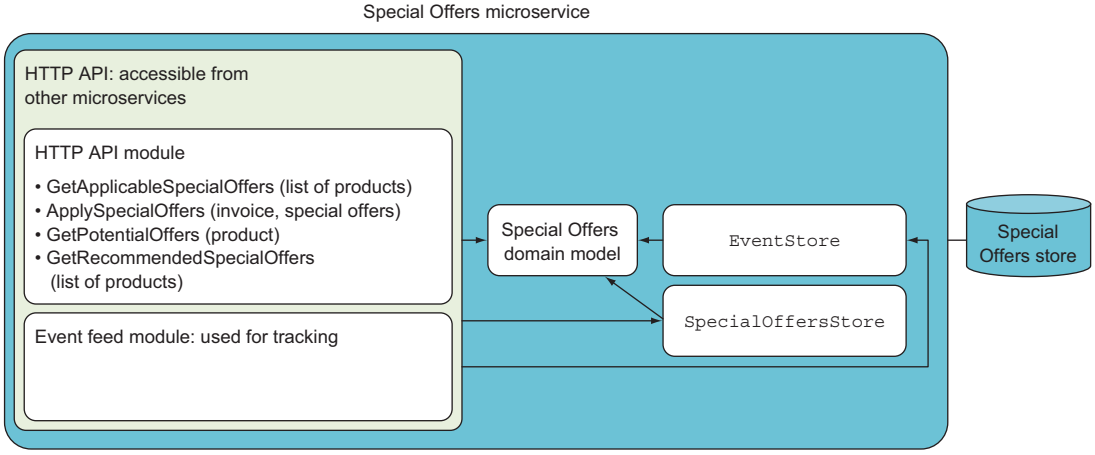


Figure 3.4 The processes in the Special Offers business capability are reflected in the implementation of the Special Offers microservice. The processes are exposed to other microservices through the microservice's HTTP API.

The components of the Special Offers microservice are similar to the components of the Shopping Cart microservice in chapter 2, which is shown again in figure 3.5. This is no coincidence. These are the components microservices typically consist of: an HTTP API that exposes the business capability implemented by the microservice, an event feed, a domain model implementing the business logic involved in the business capability, a data store component, and a database.

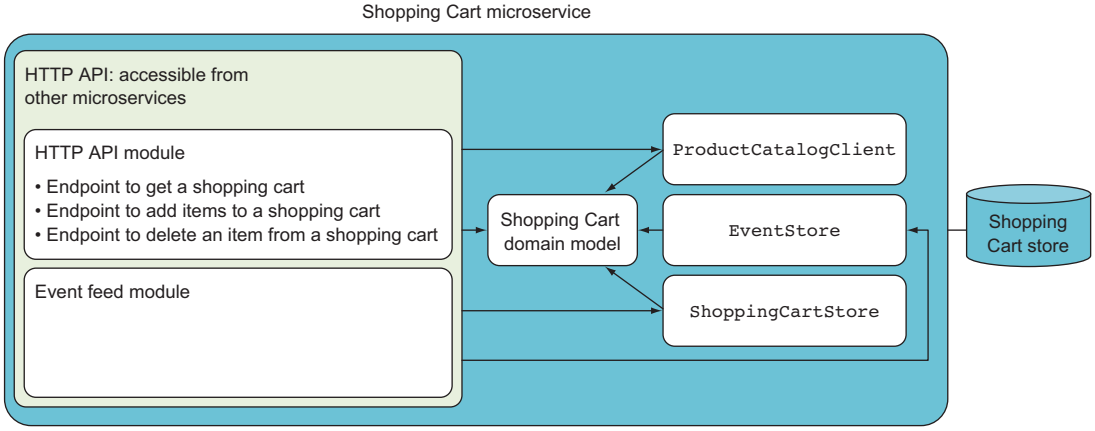


Figure 3.5 The components of the Shopping Cart microservice from chapter 2 are similar to the components of the Special Offers microservice.

3.2 The secondary driver for scoping microservices: supporting technical capabilities

The secondary way to identify scopes for microservices is to look at supporting technical capabilities. A *supporting technical capability* is something that doesn't directly contribute to a business goal but supports other microservices, such as integrating with another system or scheduling an event to happen some time in the future.

3.2.1 What is a technical capability?

Supporting technical capabilities are a secondary driver in scoping microservices because they don't directly contribute to the system's business goals. They exist to simplify and support the other microservices that implement business capabilities.

Remember, one characteristic of a good microservice is that it's replaceable; but if a microservice that implements a business capability also implements a complex technical capability, it may grow too large and too complex to be replaceable. In such cases, you should consider implementing the technical capability in a separate microservice that supports the original one. Before discussing how and when to identify supporting technical capabilities, a couple of examples would probably be helpful.

3.2.2 Examples of supporting technical capabilities

To give you a feel for what I mean by supporting technical capabilities, let's consider two examples: an integration with another system, and the ability to send notifications to customers.

INTEGRATING WITH AN EXTERNAL PRODUCT CATALOG SYSTEM

In the example point-of-sale system, you identified the product catalog as a business capability. I also mentioned that product information is maintained in another system, external to the microservice-based point-of-sale system. That other system is an Enterprise Resource Planning (ERP) system. This implies that the Product Catalog microservice must integrate with the ERP system, as illustrated in figure 3.6. The integration can be handled in a separate microservice.

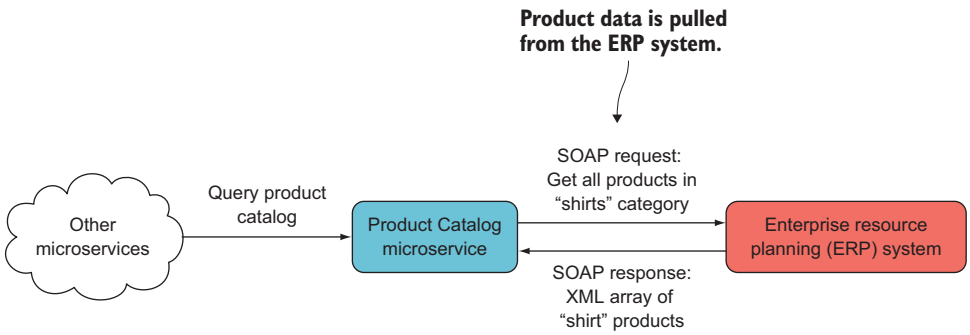


Figure 3.6 Product data flows from the ERP system to the Product Catalog microservice. The protocol used to get product information from the ERP system is defined by the ERP system. It could expose a SOAP web service for fetching the information, or it might export product information to a proprietary file format.

Let's assume that you aren't in a position to make changes to the ERP system, so the integration must be implemented using whatever interface the ERP system has. It might use a SOAP web service to fetch product information, or it might export all the product information to a proprietary file format. In either case, the integration must happen on the ERP system's terms. Depending on the interface the ERP system exposes, this may be a smaller or larger task. In any case, it's a task primarily concerned with the technicalities of integrating with some other system, and it has the potential to be at least somewhat complex. The purpose of this integration is to support the Product Catalog microservice.

You'll take the integration out of the Product Catalog microservice and implement it in a separate ERP Integration microservice that's responsible solely for that one integration, as illustrated in figure 3.7. You'll do this for two reasons:

- By moving the technical complexities of the integration to a separate microservice, you keep the scope of the Product Catalog microservice narrow and focused.
- By using a separate microservice to deal with how the ERP data is formatted and organized, you keep the ERP system's view of what a product is separate from the point-of-sale system. Remember that in different parts of a large domain, there are different views of what terms mean. It's unlikely that the Product Catalog microservice and the ERP system agree on how the product entity is modeled. A translation between the two views is needed and is best done by the new microservice. In domain-driven-design terms, the new microservice acts as an *anti-corruption layer*.

NOTE The anti-corruption layer is a concept borrowed from domain-driven design. It can be used when two systems interact; it protects the domain model in one system from being polluted with language or concepts from the model in the other system.

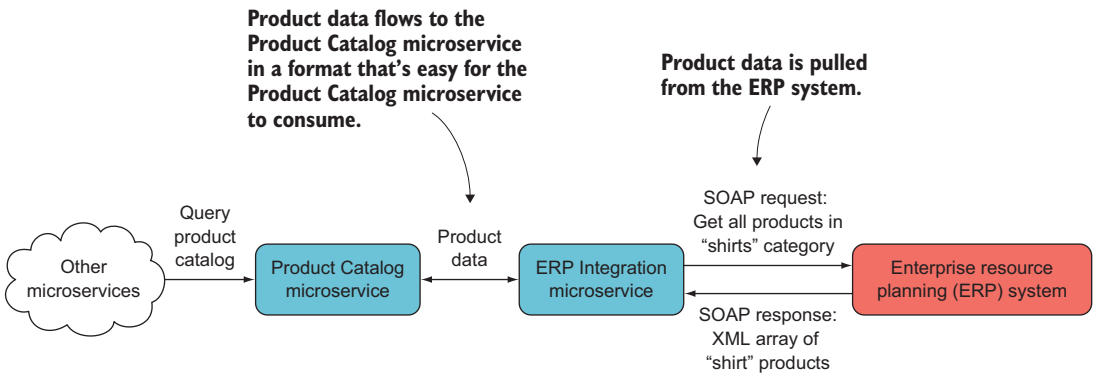


Figure 3.7 The ERP Integration microservice supports the Product Catalog microservice by handling the integration with the ERP system. It translates between the way the ERP system exposes product data and the way the Product Catalog microservice consumes it.

An added benefit of placing the integration in a separate microservice is that it's a good place to address any reliability issues related to integration. If the ERP system is unreliable, the place to handle that is in the ERP Integration microservice. If the ERP system is slow, the ERP Integration microservice can deal with that. Over time, you can tweak the policies used in the ERP Integration microservice to address any reliability issues with the ERP system without touching the Product Catalog microservice at all. This integration with the ERP system is an example of a supporting technical capability, and the ERP Integration microservice is an example of a microservice implementing that capability.

SENDING NOTIFICATIONS TO CUSTOMERS

Now let's consider extending the point-of-sale system with the ability to send notifications about new special offers to registered customers via email, SMS, or push notification to a mobile app. You can put this capability into one or more separate microservices.

At the moment, the point-of-sale system doesn't know who the customers are. To drive better customer engagement and customer loyalty, the company decides to start a small loyalty program where customers can sign up to be notified about special offers. The customer loyalty program is a new business capability and will be the responsibility of a new Loyalty Program microservice. Figure 3.8 shows this microservice, which is responsible for notifying registered customers every time a new special offer is available.

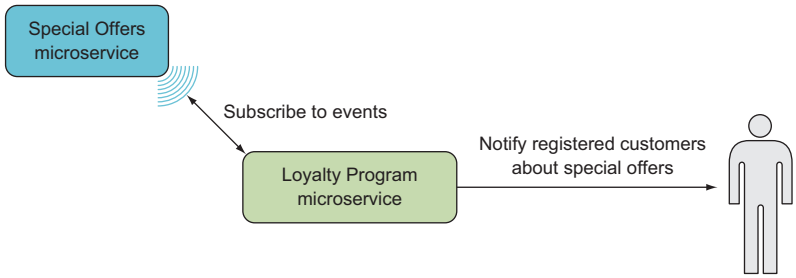


Figure 3.8 The Loyalty Program microservice subscribes to events from the Special Offers microservice and notifies registered customers when new offers are available.

As part of the registration process, customers can choose to be notified by email, SMS, or, if they have the company's mobile app, push notification. This introduces some complexity in the Loyalty Program microservice in that it must not only choose which type of notification to use but also deal with how each one works. As a first step, you'll introduce a supporting technical microservice for each notification type. This is shown in figure 3.9.

This is better. The Loyalty Program microservice doesn't have to implement all the details of dealing with each type of notification, which keeps the microservice's

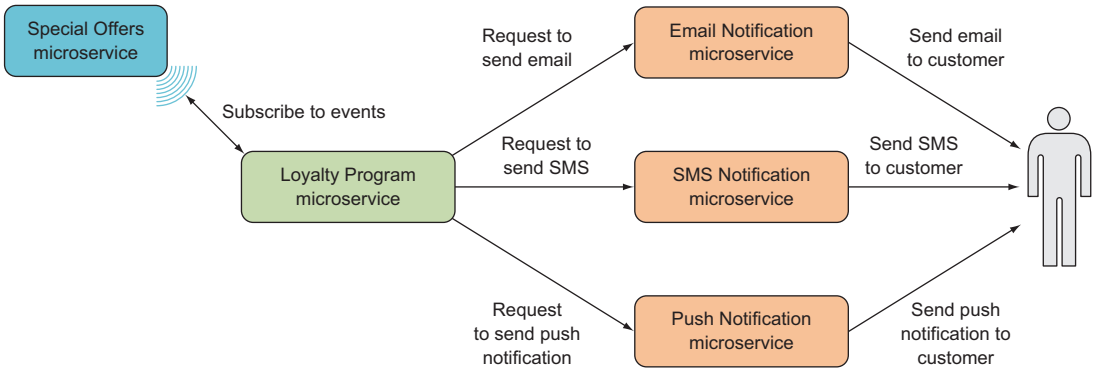


Figure 3.9 To avoid bogging down the Loyalty Program microservice in technical details for handling each type of notification, you'll introduce three supporting technical microservices, one for each type of notification.

scope narrow and focused. The situation isn't perfect, though: the microservice still has to decide which of the supporting technical microservices to call for each registered customer.

This leads you to introducing one more microservice, which acts as a front for the three microservices handling the three types of notifications. This new Notifications microservice is depicted in figure 3.10 and is responsible for choosing which type of notification to use each time a customer needs to be notified. This isn't really a business capability, although it's less technical than dealing with sending SMSs. I consider the Notifications microservice a supporting technical microservice rather than one implementing a business capability.

This example of a supporting technical capability differs from the previous example of the ERP integration in that other microservices may also need to send notifications to specific customers. For instance, one of the functionalities of the point-of-sales system is to send the customer an electronic receipt. The microservice in charge

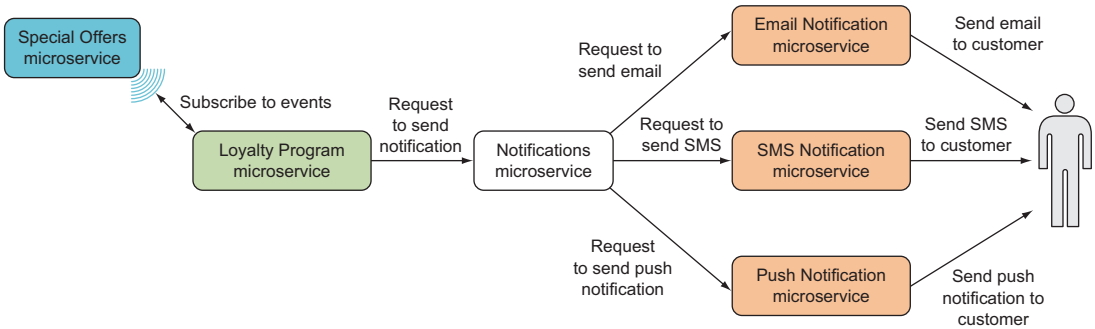


Figure 3.10 To remove more complexity from the Loyalty Program microservice, you'll introduce a Notifications microservice that's responsible for choosing a type of notification based on customer preferences. Introducing this microservice has the added benefit of making notifications easier to use from other microservices.

of that business capability can also take advantage of the Notifications microservice. Part of the motivation for moving this supporting technical capability to separate microservices is that you can reuse the implementation.

3.2.3 Identifying technical capabilities

When you introduce supporting technical microservices, your goal is to simplify the microservices that implement business capabilities. Sometimes—such as with sending notifications—you identify a technical capability that several microservices need, and you turn that into a microservice of its own, so other microservices can share the implementation. Other times—as with the ERP integration—you identify a technical capability that unduly complicates a microservice and turn that capability into a microservice of its own. In both cases, the microservices implementing business capabilities are left with one less technical concern to take care of.

When deciding to implement a technical capability in a separate microservice, be careful that you don't violate the microservice characteristic of being individually deployable. It makes sense to implement a technical capability in a separate microservice only if that microservice can be deployed and redeployed independently of any other microservices. Likewise, deploying the microservices that are supported by the microservice providing the technical capability must not force you to redeploy the microservice implementing the technical capability.

Identifying business capabilities and microservices based on business capabilities is a strategic exercise, but identifying technical supporting capabilities that could be implemented by separate microservices is an opportunistic exercise. The question of whether a supporting technical capability should be implemented in its own microservice is about what will be easiest in the long run. You should ask these questions:

- If the supporting technical capability stays in a microservice scoped to a business capability, is there a risk that the microservice will no longer be replaceable with reasonable effort?
- Is the supporting technical capability implemented in several microservices scoped to business capabilities?
- Will a microservice implementing the supporting capability be individually deployable?
- Will all microservices scoped to business capabilities still be individually deployable if the supporting technical capability is implemented in a separate microservice?

If your answer is "Yes" to the last two questions and to at least one of the others, you have a good candidate for a microservice scope.

3.3 What to do when the correct scope isn't clear

At this point, you may be thinking that scoping microservices correctly is difficult: you need to get the business capabilities just right, which requires a deep understanding of the business domain, and you also have to judge the complexity of supporting technical

capabilities correctly. And you're right: it *is* difficult, and you *will* find yourself in situations where the right scoping for your microservices isn't clear.

This lack of clarity can have several causes, including the following:

- *Insufficient understanding of the business domain*—Analyzing a business domain and building up a deep knowledge of that domain is difficult and time consuming. You'll sometimes need to make decisions about the scope of microservices before you've been able to develop sufficient understanding of the business to be certain you're making the correct decisions.
- *Confusion in the business domain*—It's not only the development side that can be unclear about the business domain. Sometimes the business side is also unclear about how the business domain should be approached. Maybe the business is moving into new markets and must learn a new domain along the way. Other times, the existing business market is changing because of what competitors are doing or what the business itself is doing. Either way, on both the business side and the development side, the business domain is ever-changing, and your understanding of it is emergent.
- *Incomplete knowledge of the details of a technical capability*—You may not have access to all the information about what it takes to implement a technical capability. For instance, you may need to integrate with a badly documented system, in which case you'll only know how to implement the integration once you're finished.
- *Inability to estimate the complexity of a technical capability*—If you haven't previously implemented a similar technical capability, it can be difficult to estimate how complex the implementation of that capability will be.

None of these problems means you've failed. They're all situations that occur time and again. The trick is to know how to move forward in spite of the lack of clarity. In this section, I'll discuss what to do when you're in doubt.

3.3.1 Starting a bit bigger

When in doubt about the scope of a microservice, it's best to err on the side of making the microservice's scope bigger than it would be ideally. This may sound weird—I've talked a lot about creating small, narrowly focused microservices and about the benefits that come from keeping microservices small. And it's true that significant benefits can be gained from keeping microservices small and narrowly focused. But you must also look at what happens if you err on the side of too narrow a scope.

Consider the Special Offers microservice discussed earlier in this chapter. It implements the Special Offers business capability in a point-of-sale system and includes five different business processes, as illustrated in figure 3.3 and reproduced on the left side of figure 3.11. If you were uncertain about the boundaries of the Special Offers business capability and chose to err on the side of too small a scope, you might split the business capability as shown on the right side of figure 3.11.

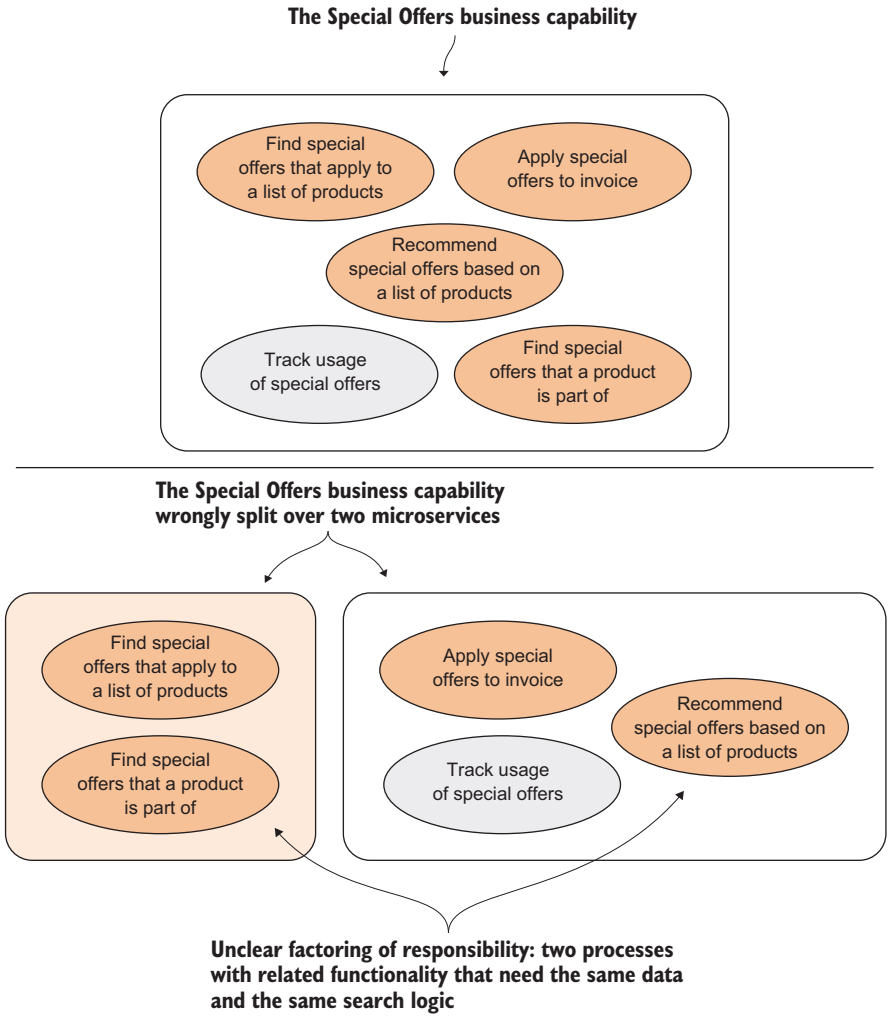


Figure 3.11 If you make the scope of a microservice too small, you'll find that a single business capability becomes split over several highly coupled parts.

If you base the scope of your microservices on only part of the Special Offers business capability, you'll incur some significant costs:

- *Data and data-model duplication between the two microservices*—Both parts of the implementation need to store all the special offers in their data stores.
- *Unclear factoring of responsibility*—One part of the divided business capability can answer whether a given product is part of any special offers, whereas the other part can recommend special offers to customers based on past purchases. These two functions are closely related, and you'll quickly get into a situation where it's unclear in which microservice a piece of code belongs.

- *Obstacles to refactoring the code for the business capability*—This can occur because the code is spread across the code bases for the two microservices. Such cross-code base refactorings are difficult because it's hard to get a complete picture of the consequences of the refactoring and because tooling support is poor.
- *Difficulty deploying the two microservices independently*—After refactoring or implementing a feature that involves both microservices, the two microservices may need to be deployed at the same time or in a particular order. Either way, coupling between versions of the two microservices violates the characteristic of microservices being individually deployable. This makes testing, deployment, and production monitoring more complicated.

These costs are incurred from the time the microservices are first created until you've gained enough experience and knowledge to more correctly identify the business capability and a better scope for a microservice (the entire Special Offers business capability, in this case). Added to those costs is the fact that difficulty refactoring and implementing changes to the business capability will result in you doing less of both, so it will take you longer to learn about the business capability. In the meantime, you pay the cost of the duplicated data and data model and the cost of the lack of individual deployability.

We've established that preferring to err on the side of too narrow a scope easily leads to scoping microservices in a way that creates costly coupling between the microservices. To see if this is better or worse than erring on the side of too big a scope, we need to look at the costs of that approach.

If you err on the side of bigger scopes, you might decide on a scope for the Special Offers microservice that also includes handling coupons. The scope of this bigger Special Offers microservice is shown in figure 3.12.

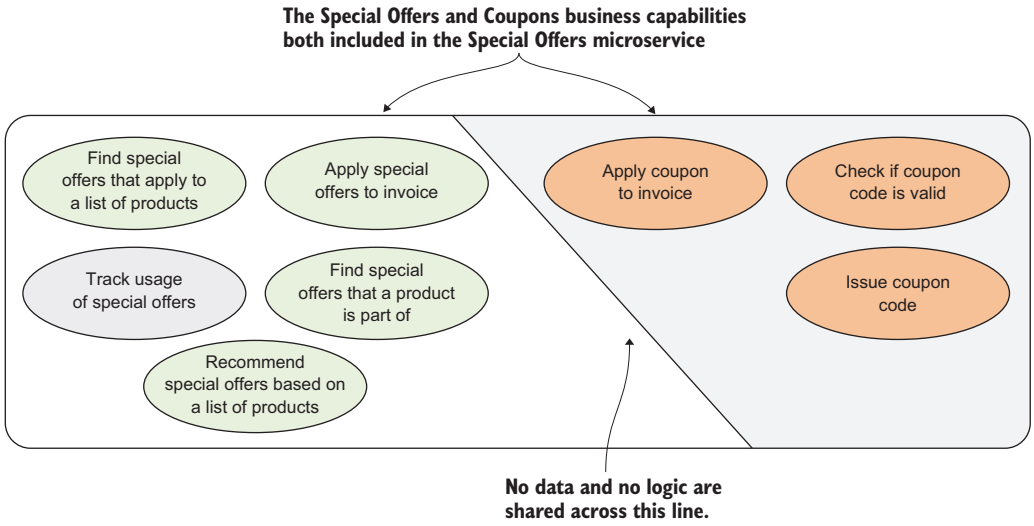


Figure 3.12 If you choose to err on the side of bigger scopes, you might decide to include the handling of coupons in the Special Offers business capability.

There are costs associated with including too much in the scope of a microservice:

- The code base becomes bigger and more complex, which can lead to changes being more expensive.
- The microservice is harder to replace.

These costs are real, but they aren't overwhelming when the scope of the microservice is still fairly small. Beware, though, because these costs grow quickly with the size of each microservice's scope and become overwhelming when the scope is so big that it approaches a monolithic architecture.

Nevertheless, refactoring within one code base is much easier than refactoring across two code bases. This gives you a better chance to experiment and to learn about the business capability through experiments. If you take advantage of this opportunity, you can arrive at a good understanding of both the Special Offers business capability and the Coupons business capability more quickly than if you scoped your microservices too narrowly.

This argument holds true when your microservices are a bit too big, but it falls apart if they're much too big, so don't get lazy and lump several business capabilities together in one microservice. You'll quickly have a large, hard-to-manage code base with many of the drawbacks of a full-on monolith.

All in all, microservices that are slightly bigger than they should ideally be are both less costly and allow for more agility than if they're slightly smaller than they should ideally be. Thus, the rule of thumb is to err on the side of slightly bigger scopes.

Once you accept that you'll sometimes—if not often—be in doubt about the best scope for a microservice and that in such cases you should lean toward a slightly bigger scope, you can also accept that you'll sometimes—if not often—have microservices in your system that are somewhat larger than they should ideally be. This means you should expect to have to carve new microservices out of existing ones from time to time.

3.3.2 Carving out new microservices from existing microservices

When you realize that one of your microservices is too big, you'll need to look at how to carve a new microservice out of it. First you need to identify a good scope for both the existing microservice and the new microservice. To do this, you can use the drivers described earlier in this chapter.

Once you've identified the scopes, you must look at the code to see if the way it's organized aligns with the new scopes. If not, you should begin refactoring toward that alignment. Figure 3.13 illustrates on a high level the refactorings needed to prepare to carve out a new microservice from an existing one. First, everything that will eventually go into the new microservice is moved to its own class library. Then, all communication between code that will stay in the existing microservice and code that will be moved to the new microservice is refactored to go through an interface. This interface will become part of the public HTTP interface of the two microservices once they're split apart.

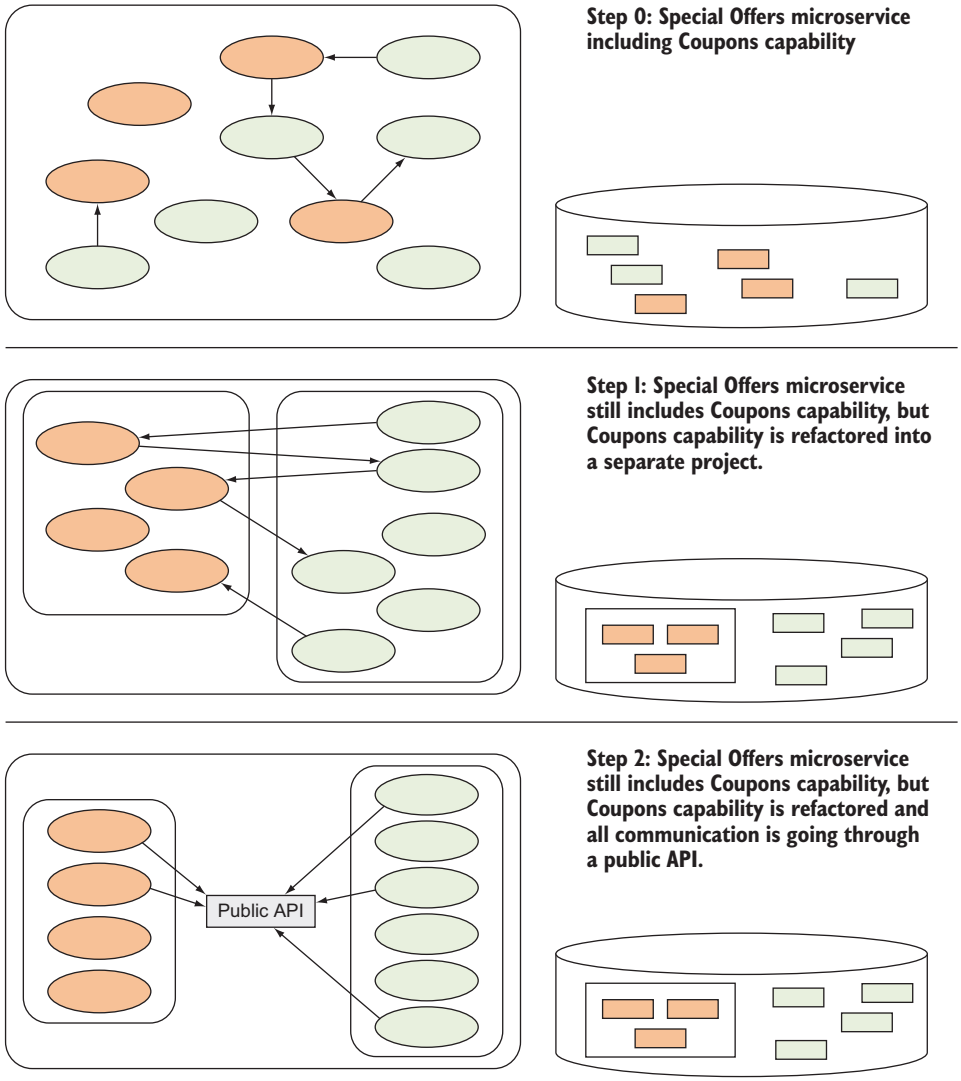


Figure 3.13 Preparing to carve out a new microservice by refactoring: first move everything belonging to the new microservice into its own project, and then make all communication go through a public API similar to the one the new microservice will end up having.

When you've reached step 2 in figure 3.13, the new microservice can be split out from the old one with a manageable effort. Create a new microservice, move the code that needs to be carved out of the existing microservice over to the new microservice, and change the communication between the two parts to go over HTTP.

3.3.3 Planning to carve out new microservices later

Because you consciously err on the side of making your microservices a bit too big when you're in doubt about the scope of a microservice, you have a chance to foresee which microservices will have to be divided at some point. If you know a microservice is likely to be split later, it would be nice if you could plan for that split in a way that will save you one or two of the refactoring steps shown in figure 3.13. It turns out you can often make that kind of plan.

Often you'll be unsure whether a particular function is a separate business capability, so you'll follow the rule of thumb and include it in a larger business capability, implemented within a microservice scoped to that larger business capability. But you can remain conscious of the fact that this area *might* be a separate business capability.

Think about the definition of the Special Offers business capability that includes processes for dealing with coupons. You may well have been in doubt about whether handling coupons was a business capability on its own, so the Special Offers business capability was modeled as including all the processes shown in figure 3.12.

When you first implement a Special Offers microservice scoped to the understanding of the Special Offers business capability illustrated in figure 3.12, you don't know whether the coupons functionality will eventually be moved to a Coupons microservice. You do know, however, that the coupons functionality isn't as closely related to the rest of the microservice as some of the other areas. It's therefore a good idea to put a clear boundary around the coupons code in the form a well-defined public API and to put the coupons code in a separate class library. This is sound software design, and it will also pay off if one day you end up carving out the coupons code to create a new Coupons microservice.

3.4 Well-scoped microservices adhere to the microservice characteristics

I've talked about scoping microservices by identifying business capabilities first and supporting technical capabilities second. In this section, I'll discuss how this approach to scoping aligns with these four characteristics of microservices mentioned at the beginning of this chapter:

- A microservice is responsible for a single capability.
- A microservice is individually deployable.
- A small team can maintain a handful of microservices.
- A microservice is replaceable.

NOTE It's important to note that the relationship between the drivers for scoping microservices and the characteristics of microservices goes both ways. The primary and secondary drivers lead toward adhering to the characteristics, but the characteristics also tell you whether you've scoped your microservices well or need to push the drivers further to find better scopes for your microservices.

3.4.1 Primarily scoping to business capabilities leads to good microservices

The primary driver for scoping microservices is identifying business capabilities. Let's see how that makes for microservices that adhere to the microservice characteristics.

Responsible for a single capability

A microservice scoped to a single business capability by definition adheres to the first microservice characteristic: it's responsible for a single capability. As you saw in the examples of identifying supporting technical capabilities, you have to be careful: it's easy to let too much responsibility slip into a microservice scoped to a business capability. You have to be diligent in making sure that what a microservice implements is just one business capability and not a mix of two or more. You also have to be careful about putting supporting technical capabilities in their own microservices. As long as you're diligent, microservices scoped to a single business capability adhere to the first characteristic of microservices.

Individually deployable

Business capabilities are those that can be performed by largely independent groups within an organization, so the business capabilities themselves must be largely independent. As a result, microservices scoped to business capabilities are largely independent. This doesn't mean there's no interaction between such microservices—there can be a lot of interaction, both through direct calls between services and through events. The point is that the interaction happens through well-defined public interfaces that can be kept backward compatible. If implemented well, the interaction is such that other microservices continue to work even if one has a short outage. This means well-implemented microservices scoped to business capabilities are individually deployable.

Replaceable and maintainable by a small team

A business capability is something a small group in an organization can handle. This limits its scope and thus also limits the scope of microservices scoped to business capabilities. Again, if you're diligent about making sure a microservice handles only one business capability and that supporting technical capabilities are implemented in their own microservices, the microservices' scope will be small enough that a small team can maintain at least a handful of microservices and a microservice can be replaced fairly quickly if need be.

3.4.2 Secondarily scoping to supporting technical capabilities leads to good microservices

The secondary driver for scoping microservices is identifying supporting technical capabilities. Let's see how that makes for microservices that adhere to the microservice characteristics.

RESPONSIBLE FOR A SINGLE CAPABILITY

Just as with microservices scoped to business capabilities, scoping a microservice to a single supporting technical capability by definition means it adheres to the first characteristic of microservices: it's responsible for a single capability.

INDIVIDUALLY DEPLOYABLE

Before you decide to implement a technical capability as a separate supporting technical capability in a separate microservice, you need to ask whether that new microservice will be individually deployable. If the answer is "No," you shouldn't implement it in a separate microservice. Again, by definition, a microservice scoped to a supporting technical capability adheres to the second microservice characteristic.

REPLACEABLE AND MAINTAINABLE BY A SMALL TEAM

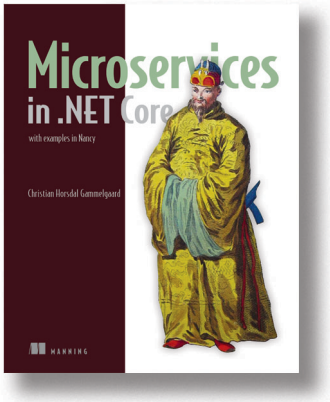
Microservices scoped to a supporting technical capability tend to be narrowly and clearly scoped. On the other hand, part of the point of implementing such capabilities in separate microservices is that they can be complex. In other words, microservices scoped to a supporting technical capability tend to be small, which points toward adhering to the microservice characteristics of replaceability and maintainability; but the code inside them may be complex, which makes them harder to maintain and replace.

This is an area where there's a certain back and forth between using supporting technical capabilities to scope microservices on one hand, and the characteristics of microservices on the other. If a supporting technical microservice is becoming so complex that it will be hard to replace, this is a sign that you should probably look closely at the capability and try to find a way to break it down further. As in the example about notification (see section 3.2.2), it's fine to have one supporting technical microservice use others behind the scenes.

3.5 Summary

- The primary driver in scoping microservices is identifying business capabilities. Business capabilities are the things an organization does that contribute to fulfilling business goals.
- You can use techniques from domain-driven design to identify business capabilities. Domain-driven design is a powerful tool for gaining better and deeper understanding of a domain. That kind of understanding enables you to identify business capabilities.
- The secondary driver in scoping microservices is identifying supporting technical capabilities. A supporting technical capability is a technical function needed by one or more microservices scoped to business capabilities.
- Supporting technical capabilities should be moved to their own microservices only if they're sufficiently complex to be a problem in the microservices they would otherwise be part of, and if they can be individually deployed.

- Identifying supporting technical capabilities is an opportunistic form of design. You should only pull a supporting technical capability into a separate microservice if it will be an overall simplification.
- When you're in doubt about the scope of a microservice, lean toward making the scope slightly bigger rather than slightly smaller.
- Because scoping microservices well is difficult, you'll probably be in doubt sometimes. You're also likely to get some of the scopes wrong in your first iteration.
- You must expect to have to carve new microservices out of existing ones from time to time.
- You can use your doubt about scope to organize the code in your microservices so that they lend themselves to carving out new microservices at a later stage.



Microservice applications are built by connecting single-capability, autonomous components that communicate via APIs. These systems can be challenging to develop because they demand clearly defined interfaces and reliable infrastructure. Fortunately for .NET developers, OWIN (the Open Web Interface for .NET), and the Nancy web framework help minimize plumbing code and simplify the task of building microservice-based applications.

Microservices in .NET Core provides a complete guide to building microservice applications. After a crystal-clear introduction to the microservices architectural style, the book will teach you practical development skills in that style, using OWIN and Nancy. You'll design and build individual services in C# and learn how to compose them into a simple but functional application back end. Along the way, you'll address production and operations concerns like monitoring, logging, and security.

What's inside:

- Design robust and ops-friendly services
- Build HTTP APIs with Nancy
- Expose events via feeds with Nancy
- Use OWIN middleware for plumbing

This book is written for C# developers. No previous experience with microservices required.

Creating and Communicating with Web Services

Much of software development involves writing and using web services. After learning about microservices in the previous selection, this chapter from *.NET Core in Action* walks you through the creation of an ASP.NET Core microservice that communicates with Microsoft Azure blob storage.

Chapter 7 from *.NET Core in Action* by Dustin Metzgar.

Creating a Microservice

This chapter covers

- Writing web services with ASP.NET
- Making HTTP requests to web services
- Guidelines for creating microservices

My personal blog's written in .NET Core (http://mode19.net). Originally, I wrote each post in its own page. Those pages were part of the source code of the blog and had corresponding metadata in a database. As the number of posts increased, the site became hard to manage because older pages were written using older libraries and techniques. The contents of the blog posts didn't change—only the formatting changed.

That's when I decided to convert my blog posts to Markdown. Markdown allows me to write the content of the blog post without having to worry about the formatting. That way, I could store my blog posts in a database or blob storage and without having to rebuild the web application every time I posted a new entry. I could convert every page on the blog to use the latest libraries I wanted to try out without touching the posts' content.

To handle the storing of posts and conversion from Markdown to HTML, I created a microservice. To best describe what a microservice is, I'll borrow the characteristics listed on the first page of Christian Horsdal Gammelgaard's *Microservices in .NET Core* (also from Manning publications). A microservice is:

- Responsible for a single piece of functionality (blog posts)
- Individually deployable (separate from blog web app)
- Singularly responsible for its datastore (creates, updates, and deletes posts in Azure blob storage)
- Replaceable (another service can replace it if it implements the same interface)

In this chapter, we'll create a blog post microservice. The data store will be Azure blob storage.. I picked Azure blob storage because it presents a challenge. While the Azure SDK is available for the .NET Standard, let's in learn how to make the HTTP requests directly.

7.1 Writing an ASP.NET web service

Our template's tuned more for web sites than web services. We'll start with this template and make the necessary adjustments to turn it into a web service-only project, but before we begin, let's find something interesting for our service to do.

7.1.1 Converting Markdown to HTML

Many implementations of Markdown exist and several are available in .NET Core or the .NET Standard. The library we'll be using is called MarkdownLite. Let's see how it works by creating an empty web application. Create a new folder called MarkdownLiteTest and run the `dotnet new console` command in it. Add a reference to "Microsoft.DocAsCode.MarkdownLite" to the project file as shown in Adding MarkdownLite as a package reference and run `dotnet restore`.

Listing 7.1 Adding MarkdownLite as a package reference

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.DocAsCode.MarkdownLite"
                      Version="2.13.1" />
  </ItemGroup>
</Project>
```

Or pick a later version from nuget.org

Now, let's try out some sample code. Test console application using MarkdownLite has a test to convert a simple Markdown text into HTML and write the HTML to the console.

Listing 7.2 Test console application using MarkdownLite

```
using System;
using Microsoft.DocAsCode.MarkdownLite;

namespace MarkdownLiteTest
{
  public class Program
  {
    public static void Main()
    {
      string source = @"
Building Your First .NET Core Applications
=======

In this chapter, we will learn how to setup our development environment,
create an application, and
";

      var builder = new GfmEngineBuilder(new Options());
      var engine = builder.CreateEngine(
          new HtmlRenderer());
      var result = engine.Markup(source);
      Console.WriteLine(result);
    }
  }
}
```

Render to HTML

Output to a string

The output should look like Output from MarkdownLite console test.

Listing 7.3 Output from MarkdownLite console test

```
<h1 id="building-your-first-net-core-applications">
Building Your First .NET Core Applications</h1>
<p>In this chapter, we will learn how to setup our development environment,
create an application, and</p>
```

MarkdownLite doesn't add the `<html>` or `<body>` tags, which is nice for inserting the generated HTML into a template. Now that we know how to use MarkdownLite, let's put it into a web service.

7.1.2 Creating an ASP.NET web service

We need to process the input coming in. ASP.NET has some built-in mechanisms to route requests based on URI and HTTP verb that we'll take advantage of in this chapter.

Start by creating a new folder called **MarkdownService** and running `dotnet new web`. Modify the project file as shown in Modify default web template project file.

Listing 7.4 Modify default web template project file

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>
```

wwwroot folder reference isn't needed

```
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore"
                      Version="1.1.1" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc"
                      Version="1.1.2" />
    <PackageReference Include="Microsoft.DocAsCode.MarkdownLite"
                      Version="2.13.1" />
  </ItemGroup>

</Project>
```

Add MVC reference

The **Program.cs** is responsible for starting the web server. The code can be simplified down to what's shown in Program.cs for the MarkdownLite service starts the Kestrel web server.

Listing 7.5 Program.cs for the MarkdownLite service starts the Kestrel web server

```
using Microsoft.AspNetCore.Hosting;

namespace MarkdownService
{
  public class Program
  {
    public static void Main()
    {
      var host = new WebHostBuilder()
          .UseKestrel()
          .UseStartup<Startup>()
          .Build();

      host.Run();
    }
  }
}
```

The **Startup** class is where we configure ASP.NET MVC. MVC handles the incoming request and routes depending on configuration and convention. Modify the **Startup.cs** file to look like the code from Startup.cs for the MarkdownLite service that sets up MVC.

MVC and Web API

MVC stands for Model View Controller, which is a pattern for building web applications. ASP.NET MVC was introduced as an alternative to the old WebForms approach for building web applications. Neither were intended for REST services so another product called Web API was introduced for which purpose.was intended for building HTTP REST services. In ASP.NET Core, Web API and MVC have been merged into one and WebForms no longer exists.

Listing 7.6 Startup.cs for the MarkdownLite service that sets up MVC

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.DocAsCode.MarkdownLite;

namespace MarkdownService
{
  public class Startup
  {
    public void Configure(IApplicationBuilder app)
    {
      app.UseMvc(); <-- Use ASP.NET MVC
    }

    public void ConfigureServices(IServiceCollection services)
    {
      services.AddMvc(); <-- ASP.NET MVC will handle the routing of requests

      var builder = new GfmEngineBuilder(new Options());
      var engine = builder.CreateEngine(new HtmlRenderer());
      services.AddSingleton<IMarkdownEngine>(engine); <-- ASP.NET Core has dependency injection built in
    }
  }
}
```

The `IMarkdownEngine` object's created at startup and added as a singleton to the dependency injection. If you're unfamiliar with dependency injection, see "A brief introduction to dependency injection" in chapter 6. ASP.NET Core uses the same **Microsoft.Extensions.DependencyInjection** library we used in chapter 6.

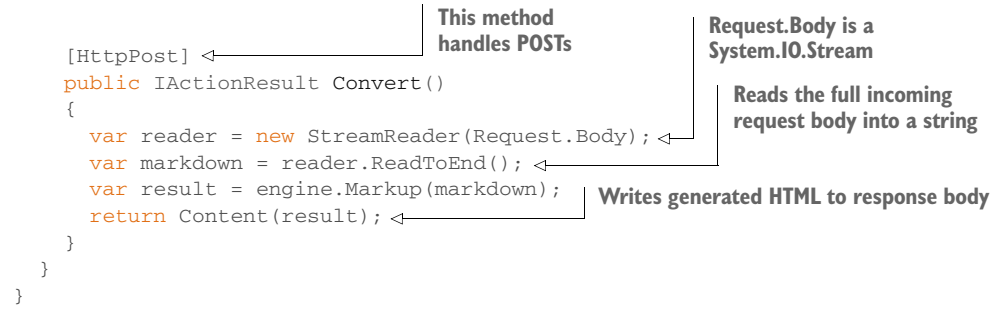
The next thing we need to do's create a controller. MVC uses reflection to find your controllers and route incoming requests to them. We need to follow the conventions. Create a new file called **MdlController.cs** and add the code from MdlController.cs is a controller for the MarkdownLite service that accepts Markdown text and returns HTML.

Listing 7.7 MdlController.cs is a controller for the MarkdownLite service that accepts Markdown text and returns HTML

```
using System.Collections.Generic;
using System.IO;
using Microsoft.AspNetCore.Mvc;
using Microsoft.DocAsCode.MarkdownLite;

namespace MarkdownService
{
  [Route("/")] <-- Indicates we want calls made to the root URL path
  public class MdlController : Controller
  {
    private readonly IMarkdownEngine engine;

    public MdlController(IMarkdownEngine engine) <-- IMarkdownEngine comes from dependency injection
    {
      this.engine = engine;
    }
```




```
    [HttpPost]
    public IActionResult Convert()
    {
      var reader = new StreamReader(Request.Body);
      var markdown = reader.ReadToEnd();
      var result = engine.Markup(markdown);
      return Content(result);
    }
  }
}
```

7.1.3 Testing the web service with Curl

After executing `dotnet restore` and `dotnet run`, you should have a web server running on http://localhost:5000, but if you navigate to this URL with a browser, you'll get a 404. The reason's because in MdlController.cs is a controller for the MarkdownLite service that accepts Markdown text and returns HTML we only created a `HttpPost` method. No `HttpGet` method exists. To test the service, we need to be able to send a POST with some Markdown text in it. The quickest way to do this is with Curl.



Curl is available on all platforms. Visit https://curl.haxx.se/download.html to download the version for your OS.

Curl's a command line tool that you'll find useful when developing web services and applications. It handles more protocols than HTTP or HTTPS. For our purposes, we can create an HTTP POST with the body contents taken from a file. First, create a file, such as **test.md**, with some Markdown text in it. Then execute a curl command like the one in Curl command to test the MarkdownLite service.

Listing 7.8 Curl command to test the MarkdownLite service

```
curl -X POST --data-binary @test.md http://localhost:5000
```



Use `--data-binary` instead of `-d` to preserve newlines.

If all goes correctly, the generated HTML should be printed on the command line. Curl made it possible to test our web service before writing client code. Because we now have a working service, let's learn how to make requests to web services as a client in .NET Core.

7.2 Making HTTP calls

We'll use the MarkdownLite service created in the previous section to test with. Leave it running and open another terminal to create this next project. Go to the **MarkdownLiteTest** folder created earlier. Copy the **test.md** file used in the previous section to the project folder and add it to the project file as shown in Copying test.md file to the project output and remove MarkdownLite package reference.

Listing 7.9 Copying test.md file to the project output and remove MarkdownLite package reference

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <None Include="test.md">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
  </ItemGroup>

</Project>
```

Recall from chapter 3

Next, let's write the code to POST data to an HTTP endpoint. The best option for this in .NET Core is `HttpClient`. Modify the **Program.cs** to add the code from Using HttpClient to call a web service, POST a file, and read the response.



.NET Framework veterans may remember `WebClient`, which wasn't originally included in .NET Core because `HttpClient` is a better option. But developers asked for `WebClient` to be included because some old `WebClient` code can't be ported to `HttpClient` easily. When writing new code, stick with `HttpClient`.

Listing 7.10 Using HttpClient to call a web service, POST a file, and read the response

```
using System;
using System.IO;
using System.Net.Http;

namespace MarkdownLiteTest
{
  public class Program
  {
    public static void Main(string[] args)
    {
      using (var client = new HttpClient())
      {
        var response = client.PostAsync(
          "http://localhost:5000",
          new StreamContent(
            new FileStream("test.md", FileMode.Open))
        ).Result;
        string markdown = response.Content.
          ReadAsStringAsync().Result;
        Console.WriteLine(markdown);
      }
    }
  }
}
```

Optional to set a base address in the HttpClient constructor

Read the test.md markdown file into an HttpContent object

Result blocks until the PostAsync operation's finished

ReadAsStringAsync returns a Task object, call Result to get the string

The `StreamContent` object inherits from `HttpContent`. You can provide any stream to `StreamContent`, which means you don't have to keep the full content of the POST in memory. The `PostAsync` method's also nice if you don't want to block the thread as you wait for the POST to complete. In this example, we didn't take advantage of the async features of .NET. But to build high performance microservices applications, we need to understand how to use these features.

7.3 Making the service asynchronous

In the code from Using HttpClient to call a web service, POST a file, and read the response, we explicitly call `.Result` on the returned values of two async methods: `PostAsync` and `ReadAsStringAsync`. These methods return `Task` objects. Our client doesn't need to be asynchronous because it's only doing one thing. It doesn't matter if we block the main thread, because there's nothing else that needs to happen.

Services can't afford to tie up threads waiting for something. Let's take a closer look at the service code the converts the posted Markdown to HTML. The method is shown in Synchronous Convert method blocks a thread waiting for the request content.

Listing 7.11 Synchronous Convert method blocks a thread waiting for the request content

```
[HttpPost]
public IActionResult Convert()
{
  var reader = new StreamReader(Request.Body);
  var markdown = reader.ReadToEnd();
  var result = engine.Markup(markdown);
  return Content(result);
}
```

This is the call that blocks the thread

The problem with blocking the thread to read the incoming HTTP request is when the client doesn't execute as fast as you think. If the client has a slow upload speed or is malicious, it could take minutes to upload all the data. Meanwhile, the service has a whole thread stuck on this client. Add enough of these clients and soon you'll run out of available threads and/or memory.

The answer's to rely on two powerful C# constructs called `async` and `await`. Asynchronous Convert method that does not block the thread waiting for the request content shows how we could rewrite the `Convert` method to be asynchronous.

Listing 7.12 Asynchronous Convert method that doesn't block the thread waiting for the request content

```
[HttpPost]
public async Task<IActionResult> Convert()
{
  using (var reader = new StreamReader(Request.Body))
  {
    var markdown = await reader.ReadToEndAsync();
    var result = engine.Markup(markdown);
    return Content(result);
  }
}
```

Mark the method async and return a Task or Task<T>

using block's to clean up reader, not necessary for async

await on the result of ReadToEndAsync()



The `async/await` constructs are a bit of compiler magic that make asynchronous code much easier to write. The `await` signals a point in the method where the code needs to wait for something. The C# compiler splits the Convert method into two methods, with the second being invoked when the awaited item's finished. This all happens behind the scenes, but if you're curious how it works, try viewing the IL generated for `async` methods in the ILDASM tool that comes with Visual Studio.

If the client's slowly uploading its request content, the only impact's the socket held open. The layers beneath your service code are responsible for gathering the network IO and buffering it until the request content length's reached. This means your service can handle more requests with fewer threads. Writing asynchronous code becomes more important as the service depends on other services, which limits operations to the speed of the network. We'll see an example of this in the next section.

7.4 Getting data from Azure blob storage

Now that we've figured out how to convert Markdown to HTML, let's incorporate storage of the posts in Azure blob storage. Instead of posting data to the Markdown service, I prefer to send it a blob name and have it return the converted HTML. We can do this by adding a GET method to our service. Before going into that, we've some values we need to pull from configuration.

7.4.1 Getting values from configuration

Our code uses the Microsoft.Extensions.Configuration library, which we learned about in chapter 6. Consult chapter 6 for instructions on adding a **config.json** file to your project, copying it to the build output, and adding the dependency on the Configuration library. The code for getting the config values is shown in Code to read the Azure storage account information from configuration.

Listing 7.13 Code to read the Azure storage account information from configuration

```
public class MdlController : Controller
{
  private readonly IMarkdownEngine engine;
  private readonly string AccountName;
  private readonly string AccountKey;
  private readonly string BlobEndpoint;
  private readonly string ServiceVersion;

  public MdlController(IMarkdownEngine engine)
  {
    this.engine = engine;
    var configBuilder = new ConfigurationBuilder();
    configBuilder.AddJsonFile("config.json", true);
    var configRoot = configBuilder.Build();
    AccountName = configRoot["AccountName"];
    AccountKey = configRoot["AccountKey"];
    BlobEndpoint = configRoot["BlobEndpoint"];
    ServiceVersion = configRoot["ServiceVersion"];
  }
```

Only using JSON config, no default config

Blob endpoint is determined differently in emulator than in Azure

The **config.json** has the four properties read in Code to read the Azure storage account information from configuration. An example config file's shown in Example config.json file for the Markdown service.

Listing 7.14 Example config.json file for the Markdown service

```
{
  "AccountName": "myaccount",
  "AccountKey": "<accountkey>",
  "BlobEndpoint": "https://myaccount.blob.core.windows.net/",
  "ServiceVersion": "2009-09-19"
}
```

If you're using the Azure emulator, often referred to as development storage, use the configuration settings from config.json when connecting to the Azure emulator.

Listing 7.15 config.json when connecting to the Azure emulator

```
{
  "AccountName": "devstoreaccount1",
  "AccountKey":

    "Eby8vdM02xNOcqFlqUwJPLlmEtlCDXJ1OUzFT50uSRZ6IFsuFq2UVErCz4I6tq/K1SZFPTO
    tr/KBHBeksoGMGw==",
  "BlobEndpoint": "http://127.0.0.1:10000/devstoreaccount1/",
  "ServiceVersion": "2009-09-19"
}
```

The account key's well known and can be found online

7.4.2 Creating the GetBlob method

In GetBlob method that gets Markdown content from Azure blob storage and returns the converted HTML, we expect the caller to pass in the container and blob names in the querystring. The method makes a request to Azure blob storage to retrieve the Markdown content. It converts the result to HTML and sends the response.

Listing 7.16 GetBlob method that gets Markdown content from Azure blob storage and returns the converted HTML

```
using System.Security.Cryptography;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;

[HttpGet]
public async Task<IActionResult> GetBlob(
  string container, string blob)
{
  var path = $"{container}/{blob}";
  var rfcDate = DateTime.UtcNow.ToString("R");
  var devStorage = BlobEndpoint.StartsWith("http://127.0.0.1:10000") ?
                   $"/{AccountName}" : "";
  var signme =  "GET\n\n\n\n\n\n\n\n\n\n\n\n" +
                "x-ms-blob-type:BlockBlob\n" +
               $"x-ms-date:{rfcDate}\n" +
               $"x-ms-version:{ServiceVersion}\n" +
               $"/{AccountName}/{path}";
```

Add these usings to the top of the file

HttpGet indicates this method's hit when using a GET verb

Parameters can be specified in querystring or request body

Storage emulator computes URI slightly differently

The empty lines are header properties we don't want to specify

ServiceVersion comes from config

```
  var uri = new Uri(BlobEndpoint + path);
  var request = new HttpRequestMessage(HttpMethod.Get, uri);
  request.Headers.Add("x-ms-blob-type", "BlockBlob");
  request.Headers.Add("x-ms-date", rfcDate);
  request.Headers.Add("x-ms-version", ServiceVersion);

  string signature = "";
  using (var sha = new HMACSHA256(
    System.Convert.FromBase64String(AccountKey)))
  {
    var data = Encoding.UTF8.GetBytes(headers);
    signature = System.Convert.ToBase64String(sha.ComputeHash(data));
  }

  var authHeader = $"SharedKey {AccountName}:{signature}";
  request.Headers.Add("Authorization", authHeader);

  using (var client = new HttpClient())
  {
    var response = await client.SendAsync(request);
    var markdown = await response.Content.ReadAsStringAsync();
    var result = engine.Markup(markdown);
    return Content(result);
  }
}
```

BlobEndpoint comes from config

Notice the same properties in headers string

AccountKey comes from config, used to created signature

Use SHA to create the signature in the authorization property

AccountName comes from config

Sending the request and receiving the response are both async methods

The code in GetBlob method that gets Markdown content from Azure blob storage and returns the converted HTML can seem overwhelming, and we'll break it down into manageable pieces. The first part's the method signature, shown in Signature for the GetBlob method.

Listing 7.17 Signature for the GetBlob method

```
[HttpGet]
public async Task<IActionResult> GetBlob(
  string container, string blob)
```

The `HttpGet` attribute tells ASP.NET MVC that `GetBlob` receives client HTTP requests using the GET verb. The parameters of the method, "container" and "blob" are expected to be passed from the client in the querystring. For example, the client could make a GET request to "http://localhost:5000?container=netcorebook&blob=test.md". MVC extracts the name/value pairs from the querystring and matches them to the method parameters.

As of the time this book was written, there's no .NET Core version of the Azure SDK. Luckily, it's still accessible as an HTTP REST service. Most of the code in `GetBlob` is creating an HTTP request to send to Azure blob storage. You'll need an Azure storage account to test it. Azure has a 30-day free trial if you don't already have a subscription. An Azure storage emulator is available, which is part of the Azure SDK, but it only works on Windows.

The GET blob request's encapsulated in an `HttpRequestMessage` object. Let's put the code that creates that object into it's own method, as shown in Code to create an HttpRequestMessage to send a GET blob request to Azure storage.

Listing 7.18 Code to create an HttpRequestMessage to send a GET blob request to Azure storage

```
private HttpRequestMessage CreateRequest(
  HttpMethod verb, string container, string blob)
{
  var path = $"{container}/{blob}";
  var rfcDate = DateTime.UtcNow.ToString("R");
  var uri = new Uri(BlobEndpoint + path);
  var request = new HttpRequestMessage(verb, uri);
  request.Headers.Add("x-ms-blob-type", "BlockBlob");
  request.Headers.Add("x-ms-date", rfcDate);
  request.Headers.Add("x-ms-version", ServiceVersion);

  var authHeader = GetAuthHeader(
    verb.ToString().ToUpper(), path, rfcDate);
  request.Headers.Add("Authorization", authHeader);

  return request;
}
```

The date and time of the request in RFC format

Constructs the URI

Indicates blob type—blocks are a good default

Azure storage version

Covered later in this section

This chapter focuses on making requests to Azure blob storage, which are the same techniques that apply to other HTTP services. Because we'll be writing several operations against Azure blob storage in this chapter, we'll be able to reuse `CreateRequest` in other operations.

Azure blob containers have different levels of exposure. It's possible to expose the contents publicly to prevent a request from needing authentication. In our case, the container's private. The only way to access it's to use a shared key to create an authentication header in the request. In Code to create an HttpRequestMessage to send a GET blob request to Azure storage, the code for creating the authentication header's split into a separate method called `GetAuthHeader`. The code for `GetAuthHeader` is shown in Creates the authentication header for accessing the Azure storage endpoint using the shared account key.

Listing 7.19 Creates the authentication header for accessing the Azure storage endpoint using the shared account key

```
private string GetAuthHeader(string verb, string path, string rfcDate)
{
  var devStorage = BlobEndpoint.StartsWith("http://127.0.0.1:10000") ?
                   $"/{AccountName}" : "";
  var signme = $"{verb}\n\n\n\n\n\n\n\n\n\n\n\n" +
               "x-ms-blob-type:BlockBlob\n" +
               $"x-ms-date:{rfcDate}\n" +
               $"x-ms-version:{ServiceVersion}\n" +
               $"/{AccountName}{devStorage}/{path}";

  string signature;
  using (var sha = new HMACSHA256(
    System.Convert.FromBase64String(AccountKey)))
  {
    var data = Encoding.UTF8.GetBytes(signme);
```

The newlines are fields we don't need to specify

The account key's available in the Azure portal

```
    signature = System.Convert.ToBase64String(
      sha.ComputeHash(data));
  }

  return $"SharedKey {AccountName}:{signature}";
}
```

Hashes the headers with the account key

SharedKeyLite has fewer newlines

The aim of this method's to produce a hashed version of the request header. The server performs the same hash and compares against the value you sent. If they don't match, it reports an error and tells you what content it hashed. This helps in case you've mistyped something.



Authentication for Azure storage's covered in depth here: https://docs.microsoft.com/en-us/rest/api/storageservices/fileservices/authentication-for-the-azure-storage-services

The helper methods above have made the `GetBlob` method much shorter. Updated GetBlob method that uses the helper methods for creating the HTTP request against Azure blob storage has the updated version.

Listing 7.20 Updated GetBlob method that uses the helper methods for creating the HTTP request against Azure blob storage

```
[HttpGet]
public async Task<IActionResult> GetBlob(string container, string blob)
{
  var request = CreateRequest(HttpMethod.Get, container, blob);

  using (var client = new HttpClient())
  {
    var response = await client.SendAsync(request);
    var markdown = await response.Content.ReadAsStringAsync();
    var result = engine.Markup(markdown);
    return Content(result);
  }
}
```

7.4.3 Testing the new Azure storage operation

The Markdown service now has a GET operation. The first step in testing it's to put a Markdown file in an Azure blob container. Many tools exist that do this, including the Azure portal. You'll also need to get the account name and key from the Azure portal to populate the values in the **config.json**.

Once the Markdown files are in place, you can make a request to the Markdown service with a console application. Console application that calls Markdown service's Azure storage operation shows the contents of the **Program.cs** file in a console application that tests the new Azure storage operation.

```
using System;
using System.IO;
```

Listing 7.21 Console application that calls Markdown service's Azure storage operation

```
using System.Net.Http;

namespace ConsoleApplication
{
  public class Program
  {
    public static void Main(string[] args)
    {
      using (var client = new HttpClient())
      {
        var response = client.GetAsync(
          "http://localhost:5000?container=somecontainer&blob=test.md")
          .Result;
        string markdown = response.Content.
          ReadAsStringAsync().Result;
        Console.WriteLine(markdown);
      }
    }
  }
}
```

Conversely, you can use the curl command shown in Curl command to make get request against the Markdown service.

Listing 7.22 Curl command to make get request against the Markdown service

```
curl "http://localhost:5000?container=netcorebook&blob=test.md"
```



The quotations around the URL in Curl command to make get request against the Markdown service are necessary for Windows. The & symbol has a special meaning in Windows command line scripting.

7.5 Uploading and receiving uploaded data

Our Markdown service isn't technically a micro-service. One of the key principals of a micro-service is that it has its own isolated data source. In the previous section, we added blobs to the Azure storage account either through the Azure portal or an external tool. In order to isolate the data source for the Markdown service, we'll need to add methods to upload new blobs and change existing blobs. For this, we'll add a PUT operation, as in Operation to upload a blob to Markdown service's blob storage account.

Listing 7.23 Operation to upload a blob to Markdown service's blob storage account

```
[HttpPut("{container}/{blob}")]
public async Task<IActionResult> PutBlob(string container, string blob)
{
  var contentLen = this.Request.ContentLength;
  var request = CreateRequest(HttpMethod.Put,
    container, blob, contentLen);
```

Get content length from request

Content length needed to create request header

```
  request.Content = new StreamContent(
    this.Request.Body);
  request.Content.Headers.Add("Content-Length",
    contentLen.ToString());

  using (var client = new HttpClient())
  {
    var response = await client.SendAsync(request);
    if (response.StatusCode = HttpStatusCode.Created)
      return Created(
        $"{AccountName}/{container}/{blob}", null);
    else
      return Content(await
        response.Content.ReadAsStringAsync());
  }
}
```

Read the stream from the request into our request

Notice this header's on the request content

Respond with 201 status code and path

Send any errors back to client

In the `PutBlob` method, we're taking a PUT request and creating our own request with the right Authorization header for Azure blob storage. In a production service, you wouldn't expose a secure resource through an unsecure one. Securing services with ASP.NET Core's a deep subject that you can read about in ASP.NET Core in Action (also from Manning publications). The purpose of this example's to explore how PUT operations work.

An HTTP PUT operation's considered idempotent. This means that no matter how many times you call it, it'll result in the same outcome. If you PUT the same blob multiple times, each call returns a 201. Contrast this with POST, which isn't idempotent. If you perform a POST and it times out, the state of the resource's unknown and we need to make a GET call to verify the state of the resource before retrying the POST. But with a PUT, we retry knowing that a duplicate call won't result in adverse effects. In the Markdown service, we use POST only for an operation that doesn't save data.

The content of the Markdown file that the client's requesting to store in our service's in the body of the request. We can get a `Stream` with the content data directly from `this.Request.Body`. Rather than measure the length of the content ourselves, we get it from the incoming request using `this.Request.ContentLength`. The content length's a required header for PUT operations to Azure blob storage. But you'll notice it's added to `Request.Content.Headers` instead of `Request.Headers`. Content headers include things like length, type, and encoding. This probably has to do with the fact that these headers are special and are indicated by position rather than name. To see what I mean by that, look at how the authentication header's created in GetAuthHeader method modified to allow content length specification.

Listing 7.24 GetAuthHeader method modified to allow content length specification

```
private string GetAuthHeader(string verb, string path,
  string rfcDate, long? contentLen)
{
  var devStorage = BlobEndpoint.StartsWith("http://127.0.0.1:10000") ?
                   $"/{AccountName}" : "";
```

Optional value for content length

```
  var signme = $"{verb}\n\n\n" +
               $"{contentLen}\n" +
                "\n\n\n\n\n\n\n\n\n" +
                "x-ms-blob-type:BlockBlob\n" +
               $"x-ms-date:{rfcDate}\n" +
               $"x-ms-version:{ServiceVersion}\n" +
               $"/{AccountName}{devStorage}/{path}";

  string signature;
  using (var sha = new
     HMACSHA256(System.Convert.FromBase64String(AccountKey)))
  {
    var data = Encoding.UTF8.GetBytes(signme);
    signature = System.Convert.ToBase64String(sha.ComputeHash(data));
  }

  return $"SharedKey {AccountName}:{signature}";
}
```

Content length's three lines after the verb

Only the number's written, not "Content-Length: "

For a PUT operation against Azure blob storage, only the content length's required. It goes three lines after the verb. Because `contentLen` is a nullable long, nothing's written if it's null. This means we don't have to change the `GetBlob` method. But we do need to change the `CreateRequest` helper method, as shown in CreateRequest method changed to allow content length specification.

Listing 7.25 CreateRequest method changed to allow content length specification

```
private HttpRequestMessage CreateRequest(HttpMethod verb,
  string container, string blob,
  long? contentLen = default(long?))
{
  ...

  var authHeader = GetAuthHeader(verb.ToString().ToUpper(),
    path, rfcDate, contentLen);
  request.Headers.Add("Authorization", authHeader);

  return request;
}
```

Default parameter in case it's not specified

Default contentLen is null



Default parameters are a handy C# feature. They must go at the end of the parameter list and are specified by assigning a default value with `=`. The `default()` keyword creates a constant value. In the case of nullable types, like `long?`, the default is null.

To test this new method in the Markdown service, you can use the same code and Curl commands as used early in this chapter for the POST operation. Change POST to PUT and modify the URL to include the container and blob name. Curl command to test the PutBlob operation and C# client code to test the PutBlob operation show how to do this.

Listing 7.26 Curl command to test the PutBlob operation

```
curl -X PUT --data-binary @test.md http://localhost:5000/netcorebook/foo.md
```

Listing 7.27 C# client code to test the PutBlob operation

```
var response = client.PutAsync(
  "http://localhost:5000/netcorebook/foo.md",
  new StreamContent(
    new FileStream("test.md", FileMode.Open))
  ).Result;
```

7.6 Listing containers and blobs

Now that we've the ability to upload blobs to containers, we should expose a way for clients to get the list of containers and blobs in the containers. The most straightforward way's to modify the `HttpGet` operation to allow null values for blob or container. A null blob parameter indicates that the client wants a list of all blobs in the container. A null container parameter indicates that they want a list of all containers.

Azure blob storage supports list requests. It returns the list in an XML document. Up until now, we haven't specified a content type for the response. The default content type from ASP.NET is "text/html", which is perfect for a response which has been converted from Markdown to HTML. In this example, we'll return the result of the Azure storage call. HttpGet operation modified to support listing containers and blobs and return XML shows the modifications to support returning XML.

Listing 7.28 HttpGet operation modified to support listing containers and blobs and return XML

```
[HttpGet]
public async Task<IActionResult> GetBlob(string container, string blob)
{
  var request = CreateRequest(HttpMethod.Get, container, blob);
  var contentType = blob == null ? "text/xml" :
    "text/html";

  using (var client = new HttpClient())
  {
    var response = await client.SendAsync(request);
    var responseContent = await response.Content.ReadAsStringAsync();
    if (blob != null)
      responseContent = engine.Markup(responseContent);
    return Content(responseContent, contentType);
  }
}
```

Assuming the container is null, the blob is also

Only convert if it's Markdown

Overrides default content type of text/html

Making a GET request to the service with the blob or container parameter not specified results in null values passed into the `GetBlob` method. For example, to request a list of blobs in the "netcorebook" container, use URL http://localhost:5000?container=netcorebook. To get a list of all the containers, use URL http://localhost:5000.

The request to Azure blob storage's slightly different for list requests. Modifications to Azure blob storage helper methods to support listing blobs and containers shows the updates to the helper methods to list blobs and containers.

Listing 7.29 Modifications to Azure blob storage helper methods to support listing blobs and containers

```
private HttpRequestMessage CreateRequest(HttpMethod verb,
  string container, string blob, long? contentLen = default(long?))
{
  string path;
  Uri uri;
  if (blob != null)                                   <-- Gets blob content
  {
    path = $"{container}/{blob}";
    uri = new Uri(BlobEndpoint + path);
  }
  else if (container != null)                         <-- Lists blobs in a container
  {
    path = container;
    uri = new Uri($"{BlobEndpoint}{path}?restype=container&comp=list");
  }
  else                                                <-- Lists containers
  {
    path = "";
    uri = new Uri($"{BlobEndpoint}?comp=list");
  }

  var rfcDate = DateTime.UtcNow.ToString("R");
  var request = new HttpRequestMessage(verb, uri);
  if (blob != null)                                   <-- Don't write this header for list requests
    request.Headers.Add("x-ms-blob-type", "BlockBlob");
  request.Headers.Add("x-ms-date", rfcDate);
  request.Headers.Add("x-ms-version", ServiceVersion);

  var authHeader = GetAuthHeader(verb.ToString().ToUpper(), path, rfcDate,
    contentLen, blob == null, container == null);
  request.Headers.Add("Authorization", authHeader);

  return request;
}

private string GetAuthHeader(string verb, string path, string rfcDate,
  long? contentLen, bool listBlob, bool listContainer)
{
  var devStorage = BlobEndpoint.StartsWith("http://127.0.0.1:10000") ?
                   $"/{AccountName}" : "";
  var signme = $"{verb}\n\n\n" +
               $"{contentLen}\n" +
                "\n\n\n\n\n\n\n\n" +                  <-- Leave blob type out of auth header
                (listBlob ? "" : "x-ms-blob-type:BlockBlob\n") +
               $"x-ms-date:{rfcDate}\n" +
               $"x-ms-version:{ServiceVersion}\n" +
               $"/{AccountName}{devStorage}/{path}";
  if (listContainer)                                  <-- Add querystring parameters to auth header when listing
    signme +=   "\ncomp:list";
```

```
  else if (listBlob)
    signme +=   "\ncomp:list\nrestype:container";

  string signature;
  using (var sha = new
     HMACSHA256(System.Convert.FromBase64String(AccountKey)))
  {
    var data = Encoding.UTF8.GetBytes(signme);
    signature = System.Convert.ToBase64String(sha.ComputeHash(data));
  }

  return $"SharedKey {AccountName}:{signature}";
}
```

7.7 Deleting a blob

To round out the functionality of the Markdown service, we'll add the ability to delete a blob from a container. A request with a DELETE verb has a similar structure as a GET request. The only real consideration's what status code to return. Azure blob storage returns a 202 (Accepted) status code when issuing a delete blob command. This is because the blob immediately becomes unavailable but isn't deleted until a garbage collection happens. This is inline with RFC 2616 of the HTTP specification.

> *A successful response SHOULD be 200 (OK) if the response includes an entity describing the status, 202 (Accepted) if the action hasn't yet been enacted, or 204 (No Content) if the action has been enacted but the response doesn't include an entity.*
>
> — RFC 2616 (https://tools.ietf.org/html/rfc2616#section-9.7)

For the Markdown service, the blob's deleted. We won't return the value of the blob in the response, and a 204 (No Content) seems more appropriate. DeleteBlob operation to delete a blob from container shows how to write the delete operation.

Listing 7.30 DeleteBlob operation to delete a blob from a container

```
[HttpDelete]
public async Task<IActionResult> DeleteBlob(string container, string blob)
{
  var request = CreateRequest(HttpMethod.Delete,
    container, blob);

  using (var client = new HttpClient())
  {
    var response = await client.SendAsync(request);
    if (response.StatusCode==HttpStatusCode.Accepted)
      return NoContent();
    else
      return Content(await response.Content.ReadAsStringAsync());
  }
}
```

Don't forget to use the right verb here

If Azure returns 202, we'll return 204

7.8 Summary

In this chapter, we learned how to write a microservice and communicate with other HTTP services as a client. The key concepts covered were:

- Use `HttpClient` to make requests
- ASP.NET Core routes messages based on the `HttpGet`, `HttpPost`, etc. attributes
- ASP.NET Core automatically populates method parameters and allows access to the raw stream from the request
- Microservices control their own data store

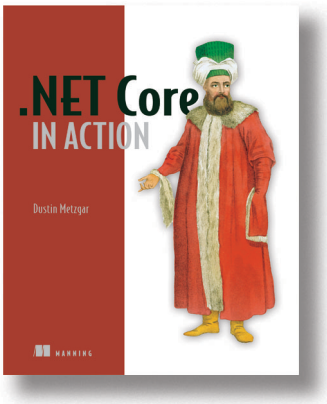
Some important techniques to remember from this chapter:

- A library called MarkdownLite's available for quick and easy conversion of Markdown to HTML
- Async programming leaves threads unblocked, which improves the performance of your application
- Curl's a powerful and simple tool for quickly testing your services

Much of modern programming involves writing and communicating with HTTP services. ASP.NET Core makes writing HTTP REST services quick and intuitive by using a convention-based approach. Methods like `Content`, `Created`, `Accepted`, etc. match the HTTP specifications. Routing requests to the right methods are handled via the `Http*` attributes and accessing parameters from the URI or querystring doesn't require manual parsing.

Making HTTP requests from .NET Core code's straightforward. The `HttpClient` class offers useful helper methods. In this chapter, we used `HttpClient` to communicate with Azure storage. For .NET Framework developers used to having the Azure SDK, contacting the HTTP services directly can seem daunting. But once you understand how to authenticate, it's easy.

Now that we've an idea of how to build services, we need to find out how to add logging to those services. This way, we can see what's going on with our services in production. In the next chapter, we'll look at the logging mechanisms built into the .NET Standard.



.NET Core is what it sounds like. It's a subset of the .NET framework with libraries and runtimes that drastically reduce its footprint, so you can write and run .NET applications more efficiently. In addition to Windows, .NET Core includes runtimes for Mac and Linux, making it a high-productivity cross-platform option for web, cloud, and server-based applications. It's open source and backed by Microsoft, so supported operating systems, CPUs, and application scenarios will continue to grow over time.

.NET Core in Action shows .NET developers how to build professional software applications with .NET Core. You'll start by getting the big picture of how to build .NET Core applications and use the tools. Then you'll learn unit testing, debugging, and logging. You'll also discover simple data access and networking. The last part of the book goes into more advanced topics, like performance profiling, localization, and signing assemblies, that you need to know so you can release your library or application to the world. By the end of this book, you'll be able to convert existing .NET code to work on multiple platforms or start new projects with knowledge of the tools and capabilities of .NET Core.

What's inside:

- Debugging .NET Core applications
- Using PerfView to investigate performance issues
- Enabling localization in a library
- Creating unit tests with XUnit
- Converting existing .NET projects to Core
- Working with relational data stores
- Interacting with web services
- Tools for writing .NET Core apps
- All examples are in C#

This book is for developers who are familiar with a C-like language.

Creating Web Pages with MVC Controllers

The previous selection introduced ASP.NET Core for writing simple services. Andrew Lock takes you into the realm of writing web sites with ASP.NET Core Model-View-Controller in an excerpt from his book, *ASP.NET Core in Action*. You'll learn how to separate data, business logic, and the display of the data into well-defined sections. This chapter also covers the conventions used by ASP.NET Core MVC that save you time and make code easier to read.

Chapter 4 from *ASP.NET Core in Action* by Andrew Lock.

Creating web pages with MVC Controllers

This chapter covers

- What the MVC design pattern is
- How MVC is used in ASP.NET Core
- Creating MVC controllers for serving web pages

In the previous chapter, you learned about the middleware pipeline, which defines how an ASP.NET Core application responds to a request. Each piece of middleware has an opportunity to either modify or handle an incoming request, before passing the request to the next middleware in the pipeline.

In ASP.NET Core web applications, the final piece of middleware in the pipeline will normally be the `MvcMiddleware`. This is typically where you write the bulk of your application logic, by calling out to various other classes in your app. It also serves as the main entry-point for users to interact with your app. It typically takes one of two forms:

- An HTML web application, designed for direct use by users. If the application's consumed directly by users, as in a traditional web application, then

the `MvcMiddleware` is responsible for generating the "web pages" that the user interacts with. It handles requests for URLs, it receives data posted using forms, and it generates the HTML that users use to view and navigate your app.

- An API designed for consumption by another machine or in code. The other main possibility for a web application's to serve as an API, either to back-end server processes, to a mobile app, or to a client framework for building single page applications (SPAs). The same `MvcMiddleware` can fulfill this role by serving data in machine-readable formats such as JSON or XML, instead of the human-focused HTML output.

In this chapter, you'll learn how ASP.NET Core uses the `MvcMiddleware` to serve these two requirements. You'll start by looking at the Model-View-Controller (MVC) design pattern to see the benefits that can be achieved through its use, and learn why it's been adopted by many web frameworks as a model for building maintainable applications.

Next you'll learn how the MVC design pattern applies specifically to ASP.NET Core. The MVC pattern's a broad concept that can be applied in a variety of situations, but the use case in ASP.NET Core's specifically as a user interface (UI) abstraction. You'll see how to add the `MvcMiddleware` to your application, as well as how to customize it for your needs.

Once you've installed the middleware in your app, I'll show how to create your first MVC controllers. You'll learn how to define action methods to execute when your application receives a request, and how to generate a result that can be used to create an HTTP response to return. For traditional MVC web applications, this'll be a `ViewResult` that can generate HTML.

I won't cover how to create Web APIs in this chapter. Web APIs still use the `MvcMiddleware`, but they're used slightly differently. Instead of returning web pages that are directly displayed on a user's browser, they return data formatted for consumption in code. They're often used for providing data to mobile and web applications, or to other server applications, but they still follow the same general MVC pattern. You'll see how to create a Web API more generally in chapter eight.

> **NOTE** This chapter's the first of several on MVC in ASP.NET Core and the `MvcMiddleware`. As I've already mentioned, this middleware's often responsible for handling all the business logic and UI code for your application, and it's, perhaps unsurprisingly, large and somewhat complicated. The next five chapters all deal with a different aspect of the MVC pattern that makes up the MVC middleware.

In this chapter I'll try and prepare you for each of the upcoming topics, but you may find that some of the behavior feels a bit like magic at this stage. Try not to become too concerned with exactly how all the pieces tie together at this stage; focus on the specific concepts being addressed. It should all become clear as we cover the associated details in the remainder of this first part of the book.

4.1 An introduction to MVC

Depending on your background in software development, you may've previously come across the MVC pattern in some form. In web development, MVC is a common paradigm and is used in various frameworks such as Django, Rails, and Spring MVC. But as it is such a broad concept, you can find MVC in everything from mobile apps to rich client desktop applications. Hopefully that is indicative of the benefits the pattern can bring if used correctly!

In this section, I'll look at the MVC pattern in general, how it applies to ASP.NET Core, and how to add the `MvcMiddleware` to your application. By the end of this section you should have a good understanding of the benefits of this approach and how to get started.

4.1.1 The MVC design pattern

The MVC design pattern's a common pattern for designing apps that have user interfaces. Many different interpretations of the original MVC pattern can be found, each of which focuses on a slightly different aspect of the pattern. For example, the original MVC design pattern was specified with rich-client graphical user interface (GUIs) apps in mind, rather than web applications, and it uses terminology and paradigms associated with a GUI environment. Fundamentally though, the pattern aims to separate the management and manipulation of data from its visual representation.

Before I dive too far into the design pattern itself, let's consider a typical request. Imagine a user of your application requests a page that displays a ToDo list. What happens when the `MvcMiddleware` gets this request? Figure 4.1 shows how the MVC pattern's used to handle different aspects of that single page request, all of which combine to generate the final response.

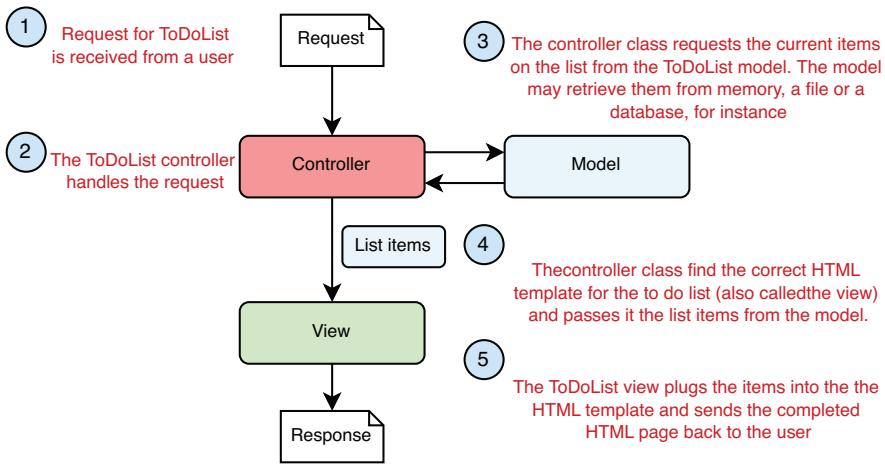


Figure 4.1 Requesting a ToDo list page for an MVC application. A different component handles each aspect of the request.

In general, three components make up the MVC design pattern:

- *Model*—The data that needs to be displayed
- *View*—The template that displays the data provided by the model
- *Controller*—Updates the model and selects the appropriate View.

Each of the components in an MVC application is responsible for a single aspect of the overall system that, when combined, can be used to generate a user interface. The ToDo list example considers MVC in terms of a web application, but a request could also be equivalent to the click of a button in a desktop GUI application for example.

In general, the order of events when an application responds to a user interaction or request is as follows:

1. The controller receives the request.
2. Depending on the request, the controller either fetches the requested data from the application model, or it updates the data that makes up the model.
3. The controller selects a view to display and passes the model to it.
4. The view uses the data contained in the model to generate the user interface.

When we describe MVC in this format, the controller serves as the entry point for the interaction. The user communicates with the controller to instigate an interaction. In web applications, this interaction takes the form of an HTTP request, and when a request to a URL is received, the controller handles it.

Depending on the nature of the request, the controller may take a variety of actions, but the key point's that the actions are undertaken using the model. The model here contains the business logic for the application, and it's able to provide requested data or perform actions.

NOTE In this description of MVC, the model is a complex beast, containing all the logic for how to perform an action, as well as any internal state.

For example, consider a request to view a product page for an ecommerce application. The controller would receive the request, and would know how to contact some product service which is part of the application model. This might fetch the details of the requested product from a database and return them back to the controller.

Alternatively, imagine the controller receives a request to add a product to the user's shopping cart. The controller would receive the request, and most likely invoke a method on the model to request that the product be added. The model would then update its internal representation of the user's cart, for example by adding a new row to a database table holding the user's data.

After the model's been updated, the controller needs to select a way to display the data. One of the advantages of using the MVC design pattern is that the model representing the data's decoupled from the final representation of that data, called the view.

This separation creates the possibility for the controller to choose to display the model using a different view, based on where the original request originated, as shown in figure 4.2. If the request came from a standard web application, then the controller can display an HTML view. If the request came from another application, then the controller can choose to display the model in a format the application understands, such as JSON or XML.

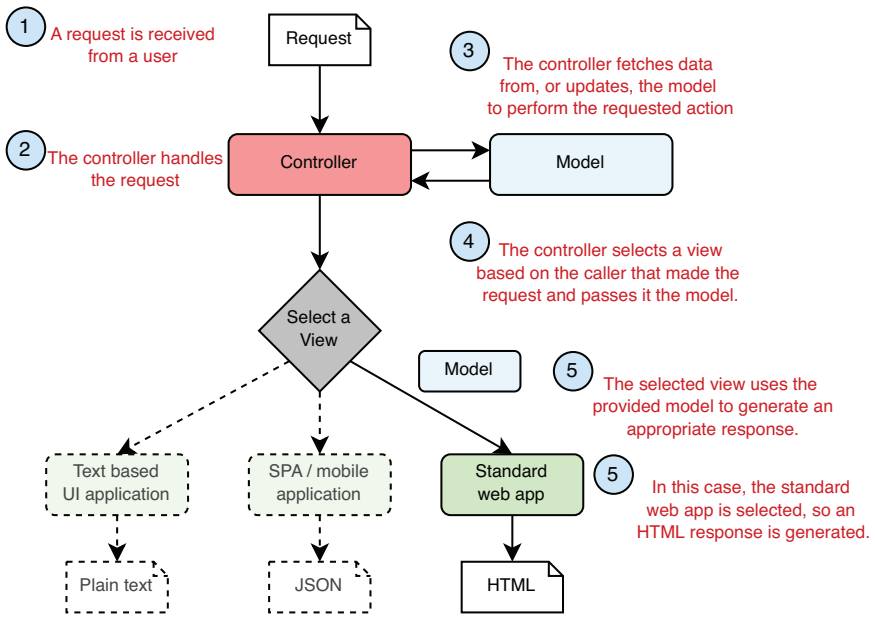


Figure 4.2 Selecting a different view using MVC depending on the caller. The final representation of the model, created by the view, is independent of the controller and business logic.

The other advantage of the model being independent of the view's that it improves testability. User interface code's classically hard to test, as it's dependent on the environment—anyone who's written UI tests simulating a user clicking on buttons and typing in forms knows that it's fragile. By keeping the model independent of the view, you can ensure the model stays easily testable, without any dependencies on UI constructs. As the model often contains your application's business logic, this is clearly a good thing!

Once the controller has selected a view, it passes the model to it. The view can use the data passed to it to generate an appropriate user interface, for example an HTML web page or a simple JSON object. The view's only responsible for the generating of the final representation.

This is all there is to the MVC design pattern as applied to web applications. Much of the confusion related to MVC seems to stem from slightly different usages of the term for slightly different frameworks and types of application. In the next section, I'll show how the ASP.NET Core framework uses the MVC pattern, along with some more examples of the pattern in action.

4.1.2 MVC in ASP.NET Core

As you've seen in previous chapters, ASP.NET Core implements MVC using a single piece of middleware, which is normally placed at the end of the middleware pipeline, as shown in figure 4.3. Once a request has been processed by each middleware (and assuming none of them handle the request and short-circuit the pipeline), it'll be received by the MVC middleware.

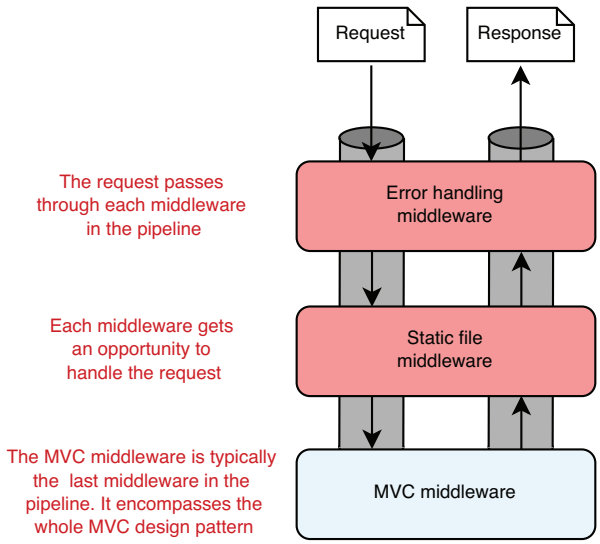


Figure 4.3 The middleware pipeline. The MVC middleware's typically configured as the last middleware in the pipeline.

Middleware often handles cross cutting concerns or narrowly defined requests, such as requests for files. For requirements that fall outside of these functions, or that have many external dependencies, a more robust framework's required. The `MvcMiddleware` in ASP.NET Core can provide this framework, allowing interaction with your application's core business logic, and generation of a user interface. It handles everything from mapping the request to an appropriate controller, to generating the HTML or API response.

In the traditional description of the MVC design pattern, there's only a single type of model, which holds all the non-UI data and behavior. The controller updates this model as appropriate and then passes it to the view, which uses it to generate a UI. This simple, three component pattern may be sufficient for some basic applications, but for more complex applications it often doesn't scale.

One of the problems when discussing MVC is the vague and overloaded terms that it uses, such as "controller" and "model." Model is such an overloaded term that it's often difficult to be sure exactly what it refers to—is it an actual object, a collection of objects, or an abstract concept? Even ASP.NET Core uses the word "model" to describe several related, but different, components, as you'll see shortly.

DIRECTING A REQUEST TO A CONTROLLER AND BUILDING A BINDING MODEL

The first step when the `MvcMiddleware` receives a request is the routing of the request to an appropriate controller. Let's think about another page in our ToDo application. On this page, you're displaying a list of items marked with a given category, assigned to a particular user. If you're looking at the list of items assigned to the user "Andrew" with a category of "Simple", you'd make a request to the URL `/todo/list/Simple/Andrew`.

Routing takes the path of the request, `/todo/list/Simple/Andrew`, and maps it against a preregistered list of patterns. These patterns match a path to a single controller class and action method. You'll learn more about routing in the next chapter.

DEFINITION An *action* (or *action method*) is a method that runs in response to a request. A *controller* is a class that contains several logically grouped action methods.

Once an action method's selected, the *binding model* (if applicable) is generated, based on the incoming request and the method parameters required by the action method, as shown in figure 4.4. A binding model's normally a standard class, with properties that map to the request data. We'll look in detail at binding models in chapter six.

DEFINITION A binding model's an object that acts a "container" for the data provided in a request which is required by an action method.

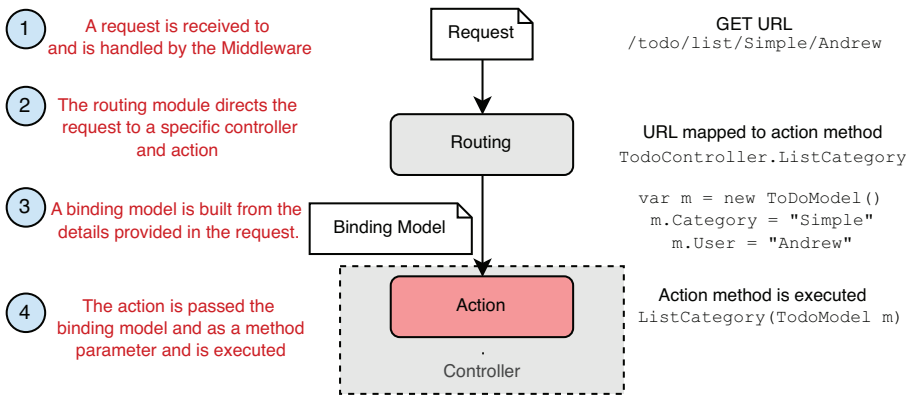


Figure 4.4 Routing a request to a controller, and building a binding model. A request to the URL `/todo/list/Simple/Andrew` results in the `ListCategory` action being executed, passing in a populated binding model.

In this case, the binding model contains two properties: `Category`, which is "bound" to the value `"Simple"`; and the property `User`, which is bound to the value `"Andrew."` These values are provided in the request URL's path and are used to populate a binding model of type `TodoModel`.

This binding model corresponds to the method parameter of the `ListCategory` action method. This binding model's passed to the action method when it executes, and it can be used to decide how to respond. For this example, the action method uses it to decide which ToDo items to display on the page.

EXECUTING AN ACTION USING THE APPLICATION MODEL

The role of an action method in the controller's to *coordinate* the generation of a response to the request it is handling. That means it should only perform a limited number of actions. It should

- Validate the binding model provided for the request
- Invoke the appropriate actions on the application model
- Select an appropriate response to generate, based on the response from the application model

Figure 4.5 shows the action model invoking an appropriate method on the application model. Here you can see that the "application model" is a somewhat abstract concept, which encapsulates the remaining non-UI part of your application. It contains the domain model, numerous services, database interaction, and so on.

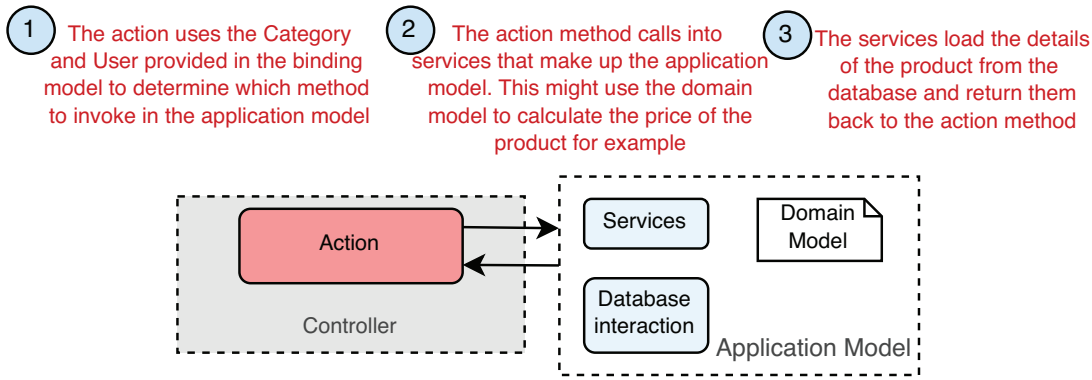


Figure 4.5 When executed, an action invokes the appropriate methods in the application model.

DEFINITION The domain model encapsulates complex business logic in a series of classes that don't depend on any infrastructure and can be easily tested

The action method typically calls into a single point in the application model. In our example of viewing a product page, the application model might use a variety of different services to check whether the user is allowed to view the product, to calculate the display price for the product, to load the details from the database, or to load a picture of the product from a file.

Assuming the request's valid, the application model returns the required details back to the action method. It's then up to the action method to choose a response to generate.

GENERATING A RESPONSE USING A VIEW MODEL

Once the action method's called out to the application model that contains the application business logic, it's time to generate a response. A *view model* captures the details necessary for the view to generate a response.

> **DEFINITION** A view model is a simple object that contains data required by the view to render a UI.

The action method selects an appropriate view template and passes the view model to it. Each view is designed to work with a specific view model, which it uses to generate the final HTML response. Finally, this is sent back through the middleware pipeline and out to the user's browser, as shown in figure 4.6.

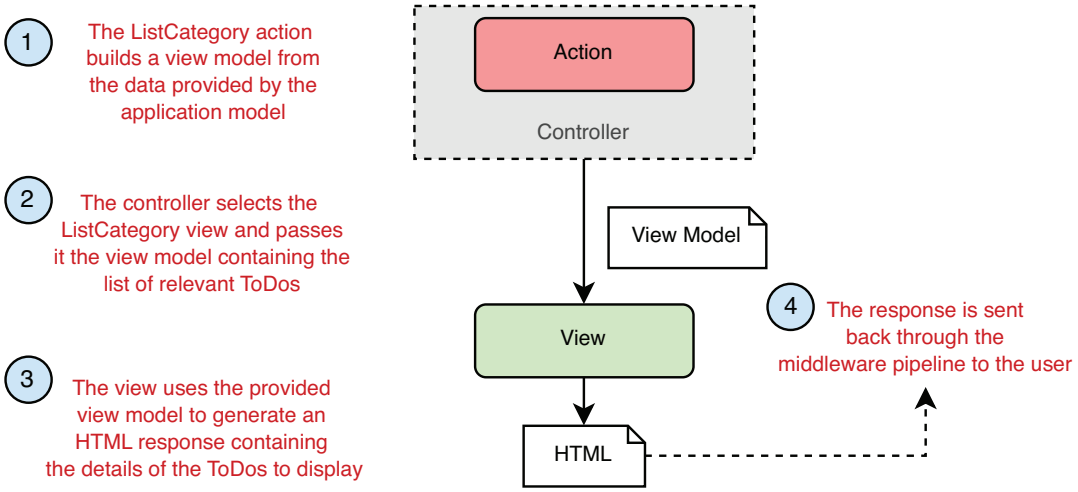


Figure 4.6 The action method builds a view model, selects which view to use to generate the response, and passes it the view model. It's the view which generates the response itself.

It's important to note that, although the action method selects which view to display, it doesn't select what's generated. It's the view that decides what the actual content of the response will be.

PUTTING IT ALL TOGETHER: A COMPLETE MVC REQUEST

Now you've seen each of the steps that go into handling a request in ASP.NET Core using MVC, let's put it all together from request to response. Figure 4.7 shows how each of the steps combine to handle the request to display the list of ToDos for user "Andrew" and category "Simple." The traditional MVC pattern's still visible in ASP.NET Core, made up of the action/controller, the view, and the application model.

By now, you might be thinking this whole process seems rather convoluted –many steps to display some HTML! Why not allow the application model to create the view

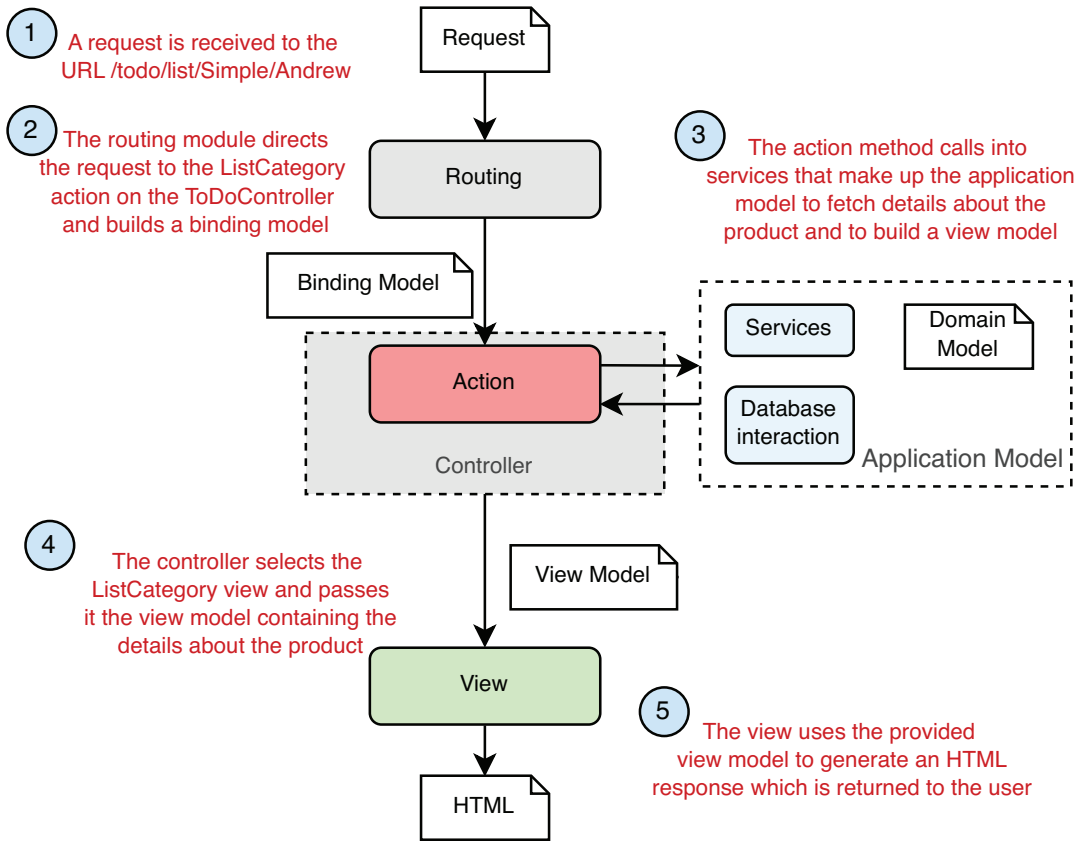


Figure 4.7 A complete MVC request for the list of ToDos in the "Simple" category for user "Andrew".

directly, rather than having to go on a dance back and forth with the controller/action method?

The key benefit throughout this process is *the separation of concerns*.

- The view's responsible only for taking some data and generating HTML.
- The application model's responsible only for executing the required business logic.
- The controller's responsible only for validating the incoming request and selecting the appropriate view to display, based on the output of the application model.

By having clearly defined boundaries it's easier to update and test each of the components without depending on any of the others. If your UI logic changes, you don't have to modify any of your business logic classes, and you're less likely to introduce errors in unexpected places.

The dangers of tight coupling

It's often a good idea to reduce coupling between logically separate parts of your application as much as possible. This makes it easier to update your application without causing adverse effects or requiring modifications in seemingly unrelated areas. Applying the MVC pattern's one way to help with this goal.

As an example of when coupling rears its head, I remember a case a few years ago when working on a small web app. In our haste, we hadn't properly decoupled our business logic from our HTML generation code, but initially there were no obvious problems—the code worked, ship it!

A few months later, someone new started working on the app, and immediately "helped" by renaming an innocuous spelling error in a class in the business layer. Unfortunately, the names of those classes had been used to generate our HTML code, and renaming the class caused the whole website to break in user's browsers! Suffice to say, we made a concerted effort to apply the MVC pattern after that, and ensure we'd a proper separation of concerns.

The examples shown here demonstrate the majority of the MVC middleware functionality. Some additional features exist, such as the filter pipeline that I'll cover later (chapter thirteen), and I'll discuss binding models in greater depth in chapter six, but the overall behavior of the system's the same.

Similarly, I'll discuss how the MVC design pattern applies when you're generating machine-readable responses using Web API controllers in chapter nine. The process is, for all intents and purposes, identical, with one exception; the result generated.

In the next section, you'll see how to add the MVC middleware to your application.Most templates in Visual Studio and the .NET CLI includes the MVC middleware by default, but you'll see how to add it to an existing application, and explore the various options available.

4.1.3 Adding the MvcMiddleware to your application

The MVC middleware's a foundational aspect of all but the simplest ASP.NET Core applications, and virtually all templates include it configured by default. To make sure you're comfortable with adding MVC to an existing project, I'll show how to start with a basic empty application and add the MVC middleware to it from scratch.

The result of your efforts won't be exciting, yet we'll display "Hello World" on a web page, but it'll show how simple it is to convert an ASP.NET Core application to use MVC. It also emphasizes the pluggable nature of ASP.NET Core—if you don't need the functionality provided by the MVC middleware, then you don't have to include it.

1. In Visual Studio 2017, choose File > New > Project
2. From the New Project dialog, choose .NET Core, and then select ASP.NET Core Web Application (.NET Core).
3. Enter a Name, Location, and optionally a solution name and click OK.

4 Create a basic template without MVC, for example by selecting the Empty Project template in Visual Studio, as shown in figure 4.8.

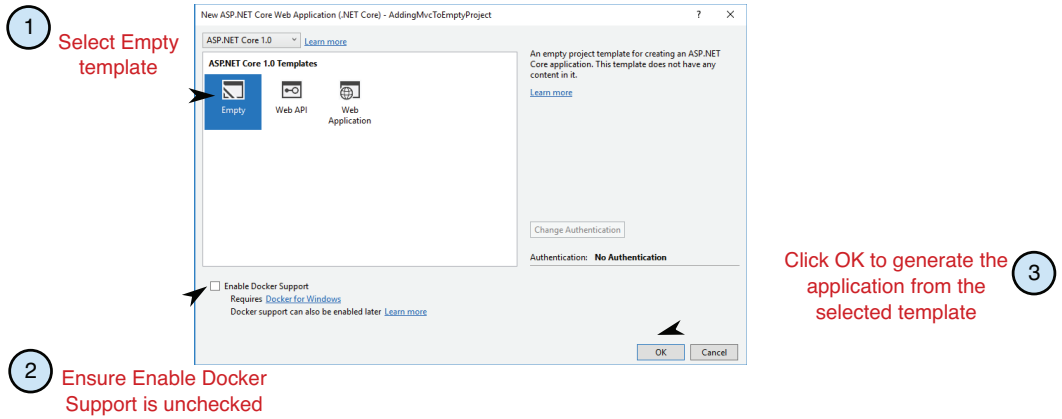


Figure 4.8 Creating an empty ASP.NET Core template. The empty template creates a simple ASP.NET Core application that contains a small middleware pipeline, but not the `MvcMiddleware`.

5 Edit your project file by Right Clicking on the project, and selecting Edit *Project*.csproj, where *Project* is the name of your project, as show in figure 4.9.

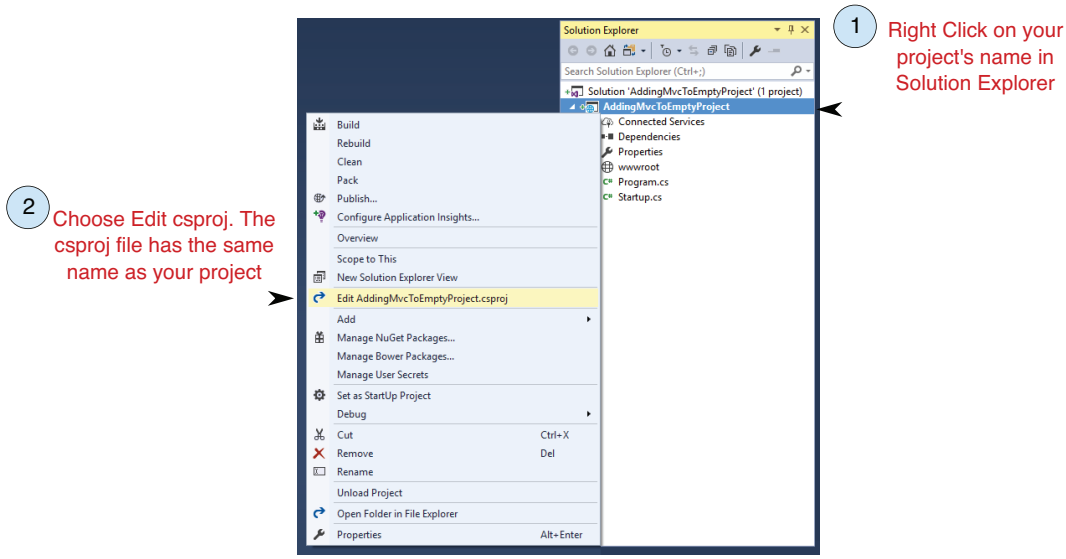


Figure 4.9 You can edit the csproj file in Visual Studio while the project is open. Alternatively, edit the csproj file directly in a text editor

6 Add the Microsoft.AspNetCore.Mvc package reference to your project's dependencies in the csproj file:

```
<!--Other configuration -->
<ItemGroup>
  <PackageReference
          Include="Microsoft.ApplicationInsights.AspNetCore"
          Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore" Version="1.0.4" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc"
          Version="1.0.3" />
</ItemGroup>
<!--Other configuration -->
```

7 Add the necessary MVC services in your Startup.cs file's `ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
}
```

8 Add the `MvcMiddleware` to the end of your middleware pipeline with the `UseMvc` extension method. For simplicity, remove any other middleware from the `Configure` method of Startup.cs for now:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
            ILoggerFactory loggerFactory)
{
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

9 Right-click on your project in solution explorer and choose Add > Class, as shown in figure 4.10.

10 In the dialog, name your class `HomeController` and click OK as in figure 4.11

11 Add an action called `Index` to the generated class:

```
public class HomeController
{
    public string Index()
    {
        return "Hello world!";
    }
}
```

Once you've completed these steps, you should be able to restore, build, and run your application.

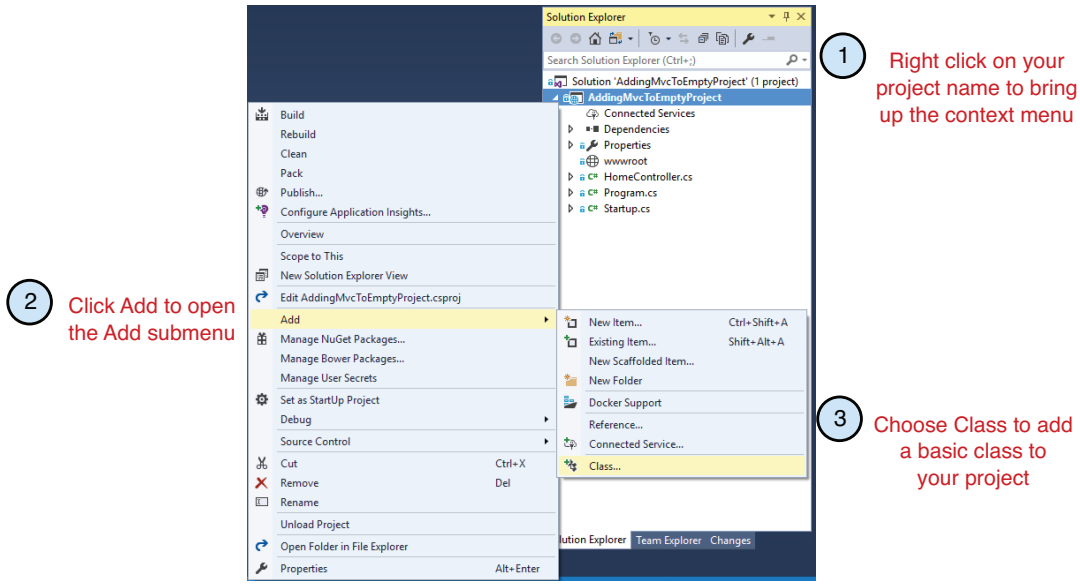


Figure 4.10 Adding a new class to your project

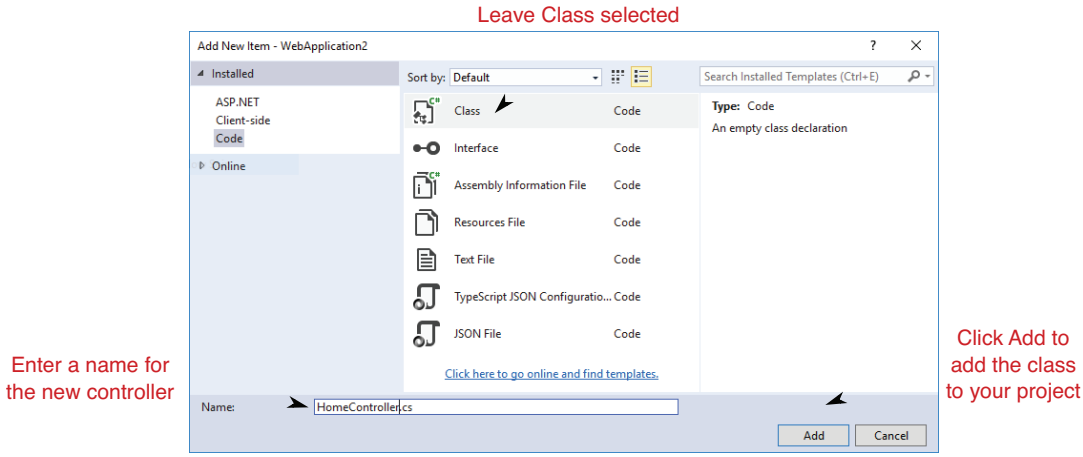


Figure 4.11 Creating a new MVC controller class using the Add New Item dialog

NOTE Remember that Visual Studio automatically restores your project and installs the referenced NuGet packages. You can run your project by pressing F5 from within Visual Studio (or by calling `dotnet run` at the command line). This'll build your project and start your application, opening a browser window to access your application's home page.

When you make a request to the path `"/"`, the application invokes the method `Index` on `HomeController`, due to the way we configured routing in the call to `UseMvc`. Don't worry about this for now; we'll go into it in detail in the next chapter.

This returns the string value `"Hello world!"`, which is rendered in the browser as plain text. You're returning data rather than a view here, and it's more of a Web API controller, but you could've created a `ViewResult` to render HTML instead.

You access the MVC functionality by adding the Microsoft.AspNetCore.Mvc package to your project. The `MvcMiddleware` relies on several internal services to perform its function, which must be registered during application startup. This is achieved with the call to `AddMvc` in the `ConfigureServices` method of Startup.cs. Without this, you'll get exceptions at runtime when the `MvcMiddleware` is invoked, reminding you that the call's required.

The call to `UseMvc` in `Configure` registers the `MvcMiddleware` in the middleware pipeline. As part of this call, the routes used to map URL paths to controllers and actions are registered. We used the default convention here, but you can easily customize these to match your requirements.

NOTE I'll cover routing in detail in the next chapter.

As you might expect, the `MvcMiddleware` comes with many options for configuring how it behaves in your application. This can be useful when the default conventions and configuration don't meet your requirements. You can modify these options by passing a configuration function to the `AddMvc` call that adds the MVC services. As an example, the following listing shows how you could use this method to customize the maximum number of validation errors that the MVC middleware can handle.

Listing 4.1 Configuring MVC options in Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.MaxModelValidationErrors = 100;
    });
}
```

AddMvc has an overload that takes a lambda function

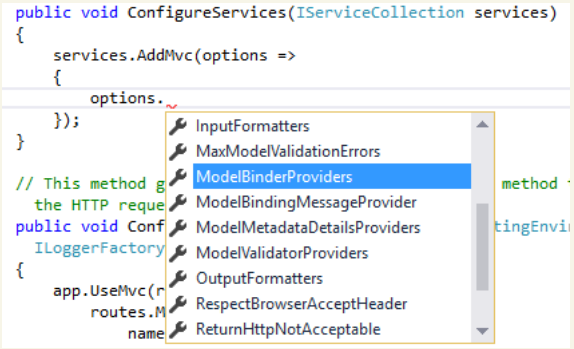
Many properties are available to customize the MvcMiddleware behavior

You can completely replace many parts of the MVC middleware internals this way thanks to the extensible design of the middleware. You won't often need to touch the MVC options, but it's nice to be able to customize them when the need arises!

Customizing the MVC middleware internals

As I've hinted, the `MvcMiddleware` exposes a large amount of its internal configuration through the `AddMvc` method, as shown in the figure below. The `options` object contains many different properties that you can use to extend and modify the default behavior of the middleware.

(continued)



Some of the customizations options available when configuring the `MvcMiddleware`

By manipulating these options, you can control things such as how data's read from the request, how the data should be validated, and how the output data's formatted. You can even modify the way actions function across your whole application. For details on the options available, see the API reference on https://docs.microsoft.com.

The final part of adding MVC to your application's creating the controllers that are invoked when a request arises. But what makes a class act as a controller?

4.1.4 What makes a controller a controller?

Controllers in ASP.NET Core are classes that contain a logical grouping of action methods. How you define them is largely up to you, but there are several conventions used by the runtime to identify controllers.

MVC or Web API controllers are discovered and used by the `MvcMiddleware` if they

- Are instantiable (they've a public constructor, aren't static, and aren't abstract); and either:
- Have a name ending in "Controller", for example `HomeController`; or
- Inherit from the `Controller` or `ControllerBase` class (or a class that inherits from these).

The `MvcMiddleware` identifies any class that meets these requirements at runtime, and make it available to handle requests as required.

Although not required, the Microsoft.AspNetCore.Mvc package provides a base class, `Controller`, which your controllers can inherit from. It's often a good idea to use this class as it contains several helper methods for returning results, as you'll see later in this chapter.

TIP If you're building a WebApi, you can also inherit from `ControllerBase`. This includes many of the same helper methods, but no helpers for creating Views.

Convention over configuration

Convention over configuration's a sometimes-controversial approach to building applications, in which a framework makes certain assumptions about the structure or naming of your code. Conforming to these assumptions reduces the amount of boilerplate code a developer must write to configure a project—you typically only need to specify the cases where your requirements don't match the assumptions.

For example, imagine you've several provider classes that can load different types of files. At runtime, you want to be able to let the user select a file and you'd automatically select the correct provider. To do this, you could explicitly register the providers in some sort of central configuration service, or you could use a convention to rely on the runtime to "find" your classes and do the wiring up for you. Typically, in .NET, this is achieved by using reflection to search through all the types in an assembly and find those with a specific name, that derive from a base class, or that contain specific named methods.

ASP.NET Core takes this approach in many cases, perhaps most notably with the Startup class, whose configuration methods are "discovered" at runtime, rather than by implementing an interface explicitly.

This approach can sometimes result in an almost magical effect of things working for no apparent reason, which some people find confusing. It can occasionally make debugging problems tricky due to the additional level of indirection at play. On the other hand, using conventions can result in terser code, as there's no explicit wiring-up necessary, and can provide additional flexibility, for example by allowing the signature of the methods in your Startup class to vary.

Another common convention's to place your controller files in a "Controllers" sub folder in your project, as shown in figure 4.12. This can be useful for organizing some projects, but it isn't required for the `MvcMiddleware` to discover them. You're free to place your controller files anywhere you like in your project folder.

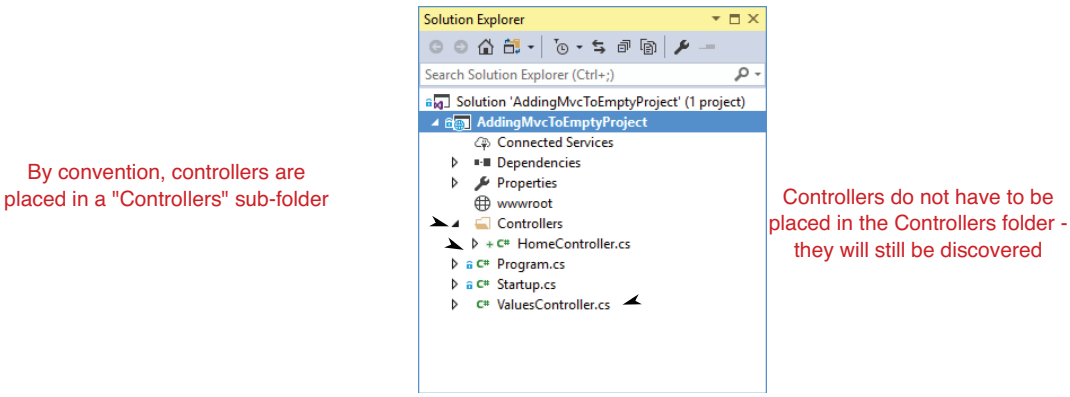


Figure 4.12 Controller location conventions in ASP.NET Core. MVC applications often place Controllers in a Controllers sub folder, but they can be located anywhere in your project—the `MvcMiddleware` still identifies and makes them available to handle requests.

Based on these requirements, it's possible to come up with a variety of naming conventions and hierarchies for your controllers, all of which will be discovered and found at runtime. In general, though, it's far better to stick to the common convention of naming your controllers by ending them with "Controller", and optionally inheriting from the `Controller` base class, as shown in the following listing.

Listing 4.2 Common conventions for defining controllers

```
public class HomeController: Controller
{
    public ViewResult Index()
    {
        return View();
    }
}

public class ValuesController
{
    public string Get()
    {
        return "Hello world!";
    }
}
```

Suffix your controller names with "Controller"

Inheriting from the Controller base class allows access to utility methods like View()

If not using the utility methods in your controller, you don't need to inherit from Controller

It's worth noting that although these examples have (implicit) parameterless constructors, it's perfectly acceptable to have dependencies in your constructor. In fact, this is one of the preferred mechanisms of accessing other classes and services from your controllers. By requiring them to be passed during construction of the controller, you explicitly define the dependencies of your controller, which, among other things, makes testing easier. The dependency injection container automatically populates any required dependencies when the controller's created.

NOTE See chapter gten for details about configuring and using dependency injection.

The controllers you've seen this far contain a single action method, which is invoked when handling a request. In the next section, I'll look at action methods, how to define them, how to invoke them, and how to use them to return views.

4.2 MVC Controllers and action methods

In the first section of this chapter I described the MVC design pattern and how it relates to ASP.NET Core. In the design pattern, the controller receives a request and is the entry point for the UI generation. In ASP.NET Core, the entry point's an action method that resides in a controller. *An action, or action method, is a method that runs in response to a request.*

MVC controllers can contain any number of action methods. Controllers provide a mechanism to logically group actions together and apply a common set of rules to

them. For example, it's simple to require a user to be logged in when accessing any action method on a given controller by applying an attribute to the controller; you don't need to apply the attribute to every individual action method.

NOTE You'll see how to apply authorization requirements to your actions and controllers in chapter fourteen.

Any public method on a controller acts as an action method, and can be invoked by a client (assuming the routing configuration allows for it). The responsibility of an action method's generally threefold:

1. Confirm the incoming request's valid.
2. Invoke the appropriate business logic corresponding to the incoming request.
3. Choose the appropriate *kind* of response to return.

An action doesn't need to perform every one of these actions, but it must choose the kind of response to return. For a traditional MVC application returning HTML to a browser, action methods typically return either a `ViewResult` that the `MvcMiddleware` uses to generate an HTML response, or a `RedirectResult`, which indicates the user should be redirected to a different page in your application. In Web API applications, action methods often return a variety of different results, as you'll see in chapter nine.

It's important to realize that an action method doesn't generate a response directly; it selects the *type* of response and prepares the data. For example, returning a `ViewResult` doesn't generate any HTML, it indicates which view template to use and the view model it has access to. This is in keeping with the MVC design pattern in which it's the *view* that generates the response, not the *controller*.

TIP The action method's responsible for choosing what sort of response to send; the *view engine* in the `MvcMiddleware` uses the action result to generate the response.

It's also worth bearing in mind that action methods should generally not be performing business logic directly. Instead, they should call appropriate services in the application model to handle requests. For example, if an action method receives a request to add a product to a user's cart, it shouldn't directly manipulate the database or recalculate cart totals. Instead, it should make a call to another class to handle the details. This approach of separating concerns ensures that your code stays testable and manageable as it grows.

4.2.1 Accepting parameters to action methods

Some requests made to action methods require additional values with details about the request. For example, if the request's for a search page, the request might contain details of the search term and the page number they're looking at. If the request's posting a form to your application, for example a user logging in with their username and password, then those values must be contained in the request. In other cases, there'll be no such values, such as when a user requests the home page for your application.

The request may contain additional values from a variety of different sources. They could be part of the URL, the query string, headers, or in the body of the request itself. The middleware extracts values from each of these sources, and converts them into .NET types.

If an action method definition has method arguments, the additional values in the request are used to create the required parameters. If the action has no arguments, then the additional values goes unused. The method arguments can be simple types, such as strings and integers, or they can be a complex type, as shown in the following listing.

Listing 4.3 Example action methods

```
public class HomeController: Controller
{
    private SearchService _searchService;
    public HomeController(SearchService searchService)
    {
        _searchService = searchService;
    }

    public ViewResult Index()
    {
        return View();
    }

    public IActionResult Search(SearchModel searchModel)
    {
        if(ModelState.IsValid)
        {
            var viewModel = _searchService.Search(searchModel);
            return View(viewModel);
        }
        return Redirect("/")
    }
}
```

The SearchService is provided to the HomeController for use in action methods

An action without parameters requires no additional values in the request

The action method requires the request to have values for the properties in SearchModel

The method doesn't need to check if the model's valid, it only returns a response.

If the model's valid, a view model's created and passed to the view

If the model isn't valid, the method indicates the user should be redirected to the path "/"

In this example, the `Index` action method doesn't require any parameters, and the method's simple—it returns a view to the user. The `Search` action method, on the other hand, accepts a `SearchModel` object. This could contain multiple different properties that are obtained from the request and are set on the model in a process called *model binding*. The `SearchModel` object's often described as a *binding model*.

NOTE I'll discuss model binding in detail in chapter six.

When an action method accepts parameters, it should always check that the model provided's valid using `ModelState.IsValid`. The `ModelState` property's exposed when you inherit from the base `Controller` class, and can be used to check the

method parameters are valid. You'll see how the process works in chapter six when you learn about validation.

Once an action establishes that the method parameters provided to an action are valid, it can execute the appropriate business logic and handle the request. In the case of the `Search` action, this involves calling the provided `SearchService`, to obtain a view model. This view model's returned in a `ViewResult` by calling the base method `return View(viewModel);`

If the model wasn't valid, then we don't have any results to display! In this example, the action returns a `RedirectResult` using the `Redirect` helper method. When executed, this result sends a 302 redirect response to the user, which causes their browser to navigate to the home page.

Note that the `Index` method returns a `ViewResult` in the method signature, but the `Search` method returns an `IActionResult`. This is required in the `Search` method to allow the C# to compile (as the `View` and `Redirect` helper methods return different types of values), but it doesn't change the final behavior of the methods. You could've returned an `IActionResult` in the Index method and the behavior would be identical.

TIP If you're returning more than one type of result from an action method, you'll need to ensure your method returns an `IActionResult`.

4.2.2 Using ActionResults

In the previous section I emphasized that action methods only decide what to generate, and don't perform the actual generation of the response. It's the `IActionResult` returned by an action method which, when executed by the `MvcMiddleware`, generates the response. The MvcMiddleware uses the view engine to execute.

This approach is key to following the MVC design pattern. It separates the decision of what sort of response to send from the actual generation of the response. This allows you to test your action method logic to confirm the right sort of response is sent for a given output. You can separately test that a given `IActionResult` generates the expected HTML.

Many different types of `IActionResult` exist in ASP.NET Core, such as:

- `ViewResult`—Generates an HTML view.
- `RedirectResult`—Sends a 302 HTTP redirect response to automatically send a user to a specified URL.
- `RedirectToRouteResult`—Sends a 302 HTTP redirect response to automatically send a user to another page, where the URL is defined using routing.
- `FileResult`—Returns a file as the response.
- `ContentResult`—Returns a provided string as the response.
- `StatusCodeResult`—Sends a raw HTTP status code as the response, optionally with associated response body content.
- `NotFoundResult`—Sends a raw 404 HTTP status code as the response.

Each of these, when executed by the `MvcMiddleware`, generates a response to send back through the middleware pipeline and out to the user.

ViewResult and RedirectResult

When you're building a traditional web application and generating HTML, most of the time you'll use the `ViewResult`, which generates an HTML response using Razor (by default). We'll look in detail as to how this happens in chapter seven.

You'll also commonly use the various redirect-based results to send the user to a new web page. For example, when you place an order on an ecommerce website you typically navigate through multiple pages, as shown in figure 4.13. The web applica-

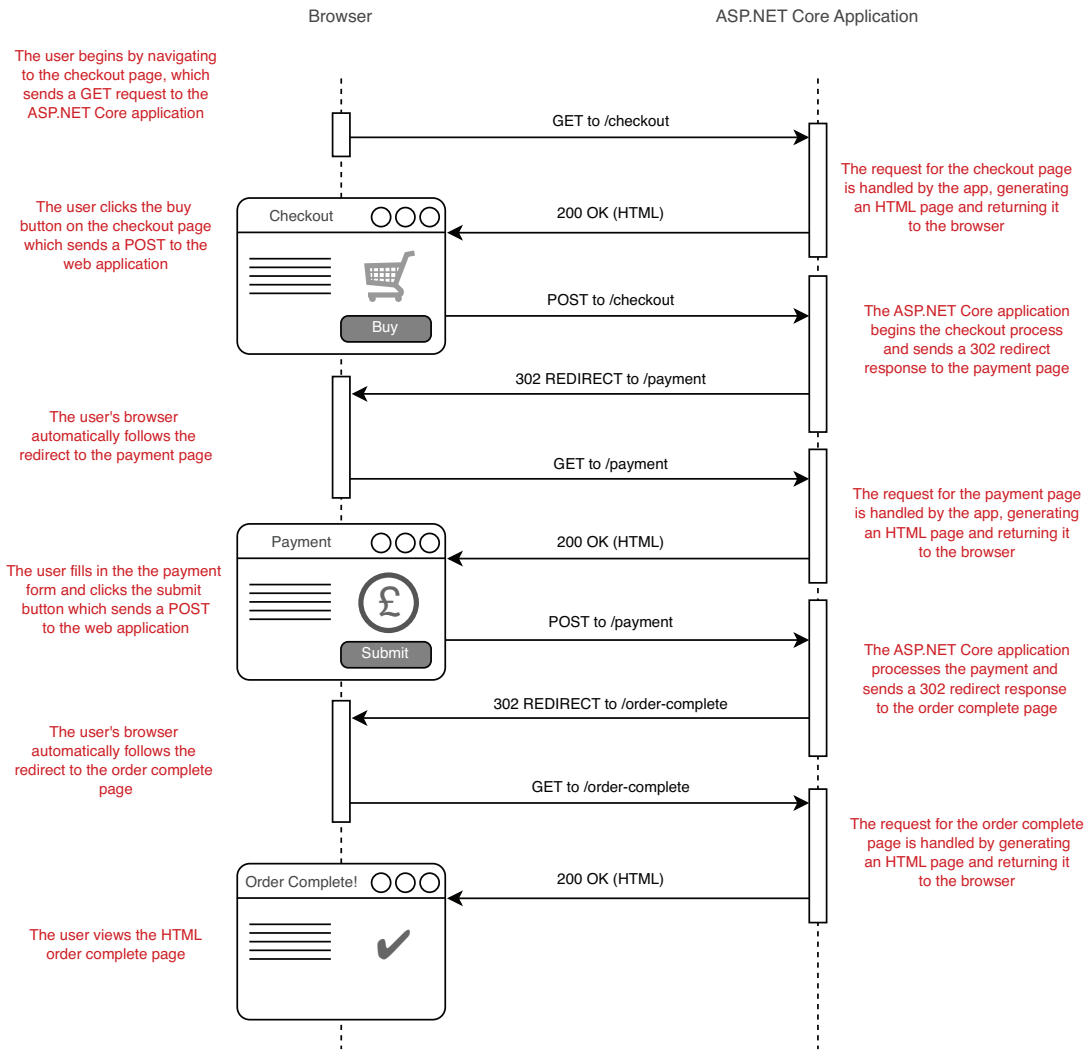


Figure 4.13 A typical POST, REDIRECT, GET flow through a website. A user sends their shopping basket to a checkout page which validates its contents and redirects to a payment page without the user having to manually change the URL.

tion sends HTTP redirects whenever it needs you to move to a different page, such as when a user submits a form. Your browser automatically follows the redirect requests giving a seamless flow through the checkout process.

NotFoundResult and StatusCodeResult

As well as HTML and redirect responses, you'll occasionally need to send specific HTTP status codes. For example, if you request a page for viewing a product on an ecommerce application, and that product doesn't exist, a 404 HTTP status code is returned to the browser and you'll typically see a "Not found" web page. The `MvcMiddleware` can achieve this behavior by returning a `NotFoundResult`, which returns a raw 404 HTTP status code. You could achieve a similar result using the `StatusCodeResult` and setting the status code returned explicitly to 404.

Note that the `NotFoundResult` doesn't generate any HTML; it only generates a raw 404 status code and returns it back through the middleware pipeline. But, as discussed in the previous chapter, you can use the `StatusCodePagesMiddleware` to intercept this raw 404 status code after it's been generated, and provide a user friendly HTML response for it.

Creating ActionResults using helper methods

`ActionResults` can be created and returned using the normal `new` syntax of C#

```
return new ViewResult()
```

but if your controller inherits from the base `Controller` class, then you can also use one of the helper methods for generating an appropriate response. It's common to use the `View` method to generate an appropriate `ViewResult`, the `Redirect` method to generate a `RedirectResponse`, or the `NotFound` method to generate a `NotFoundResult`.

> **TIP** Most `ActionResults` have a helper method on the base `Controller` class. They're typically named `Type` where the result generated's called `TypeResult`. For example, the `Content` method returns a `ContentResult` instance.

As discussed, the act of *returning* an `IActionResult` doesn't immediately generate the response—it's the *execution* of an `IActionResult` by the `MvcMiddleware`, which occurs outside the action method. After producing the response, the `MvcMiddleware` returns it back to the middleware pipeline. From there, it passes through the registered middleware in the pipeline, before the ASP.NET Core web serve finally sends it to the user.

By now you should have an overall understanding of the MVC design pattern and how it relates to ASP.NET Core. The action methods on a controller are invoked in response to given requests and are used to select the type of response to generate by returning an `IActionResult`.

In traditional web apps, the `MvcMiddleware` generates HTML web pages. These can be served to a user who is browsing your app with a web browser, as you'd see with a traditional website. It's also possible to use the `MvcMiddleware` to send data in a machine-readable format such as JSON, by returning data directly from action meth-

ods, as you'll see in chapter nine. Controllers handle these use cases, the only tangible difference being the data they return. These are typically known as MVC and Web API controllers respectively.

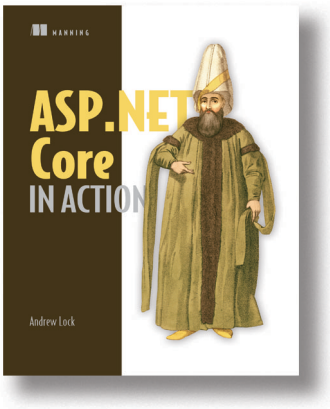
It's important to remember that the whole MVC infrastructure in ASP.NET Core's a piece of middleware that runs as part of the middleware pipeline, as you saw in the previous chapter. Any response generated, whether a `ViewResult` or a `RedirectResult`, will pass back through the middleware pipeline, giving a potential opportunity for middleware to modify the response before the web server sends it to the user.

An aspect I've only vaguely touched on is how the `MvcMiddleware` decides which action method to invoke for a given request. This process is handled by the routing infrastructure, and is a key part of MVC in ASP.NET Core. In the next chapter, you'll see how to define routes, how to add constraints to your routes, and how they deconstruct URLs to match a single action and controller.

4.3 Summary

In this chapter, you learned that

- MVC allows for a separation of concerns between the business logic of your application, the data passed around, and the display of data in a response.
- Controllers contain a logical grouping of action methods.
- ASP.NET Core Controllers inherit from the `Controller` base class, or have a name that ends in "Controller".
- Action *methods* decide what sort of response to generate; action *results* handle the actual generation.
- Action methods should generally delegate to services to handle the business logic required by a request, instead of performing the changes themselves. This ensures a clean separation of concerns that aids testing and improves application structure.
- Action methods can have parameters whose values are taken from properties of the incoming request.
- When building a traditional web application, you'll generally use a `ViewResult` to generate an HTML response.
- You can send users to a new URL using a `RedirectResult`.
- The Controller base class exposes many helper methods for creating an `ActionResult`.
- The MVC and Web API infrastructure's unified in ASP.NET Core. The only thing that differentiates a traditional MVC controller from a Web API controller's the data it returns. MVC controllers normally return a `ViewResult`, whereas Web API controllers typically return data or a `StatusCodeResult`.



ASP.NET Core is a re-imagining of the .NET Framework that frees developers from Visual Studio and Windows. You can now build and run cross-platform .NET applications on any OS, with any IDE, and using the tools that you choose. The entire framework is open-source, and has been developed with many contributions from the community. While ASP.NET Core is relatively new, Microsoft is heavily investing in it, promoting ASP.NET Core as their web framework of choice for the foreseeable future. Whether you are building traditional web applications or highly performant APIs for client side or mobile applications, ASP.NET Core could be the framework for you.

ASP.NET Core in Action is for C# developers without any web development experience who want to get started and productive using ASP.NET Core to build web applications. In the first half of the book, you will work through the basics of a typical ASP.NET Core application, focusing on how to create basic web pages and APIs using MVC controllers and the Razor templating engine. In the second half, you will build on this core knowledge looking at more advanced requirements and how to add extra features to your application. You will learn how to secure your application behind a login screen, how to handle configuration and dependency injection, and how to deploy your application to production. In the last part of the book you will look in depth at further bending the framework to your will by creating custom components and using more advanced features of the framework.

What's inside:

- Using MVC to deliver dynamically generated web pages
- Securing applications with login requirements
- Interacting with a RDMS using Entity Framework Core
- Publishing an ASP.NET Core application to a server
- Unit and integration testing
- Creating custom middleware and filters

Readers should have experience with C#. No web development experience needed.

Querying the Database

Many applications access relational databases. This chapter from Jon P. Smith's *Entity Framework Core in Action* introduces how to use the completely rewritten Entity Framework Core for data access. Despite being rebuilt from the ground up, migrating from Entity Framework to Entity Framework Core is fairly straightforward. If you're new to Entity Framework, Jon will introduce you to this powerful library and its many benefits.

Chapter 2 from *Entity Framework Core in Action* by Jon Smith.

Querying the database

This chapter covers

- The three main types of database relationship and how they're modelled in EF Core
- How to create and change a database using EF Core's migration feature
- How to define and create an application DbContext
- The three ways of loading related data
- How EF Core's Client vs. Server feature works
- A technique for splitting complex queries down into smaller sub-queries to make them easier to write, with no inherent loss in database performance

This chapter's all about reading, called *querying*, the database using EF Core. To help explain this I create a database which contains the three main types of database relationship that you'll come across in EF Core. On the way, I show you how to create and change a database's structure from EF Core.

Once I've set you up with that, I then explain how to access a database via EF Core—reading data from the database tables. I start by describing the basic format of EF Core queries before looking at the different approaches to loading related data with the main data, for instance loading the Author with the book in chapter one.

Once I've explained the different ways to load related data, I start to build the more complex queries needed to make the book-selling site work. This covers sorting, filtering, and paging, plus some approaches to combine each of these separate query commands to create one composite database query.

2.1 Setting the scene—my book-selling site example

In this chapter, I start building the book-selling site. This example application provides a good vehicle for looking at relationships in queries. In this section, I introduce the database, the various classes, and EF Core parts that the book-selling site application needs to access the database.

NOTE You can see a live site of the book-selling site at http://efcoreinaction.com/ .

2.1.1 Our book-selling site relational database

Although I could've created a database with all the data about a book, its author(s), and its reviews in one table, that wouldn't have worked well in a relational database because the reviews are variable in length. The norm for relational databases is to split out any repeated data (in this case the Authors).

I could've arranged the various parts of the book data in the database in several different ways, but for this example I designed the database to have one of each of the main types of relationships you can have in EF Core. The three types are:

- One-to-One relationship: PriceOffer to a Book
- One-to-Many relationship: Reviews to a Book
- Many-to-Many relationship: Books to Authors

ONE-TO-ONE RELATIONSHIP: PRICEOFFER TO A BOOK

A book can have a promotional price applied to it. This is done with an optional `PriceOffer`, which is an example of a One-to-One (technically it's a One-to-ZeroOrOne relationship, but EF Core handles this the same way)—see figure 2.1.

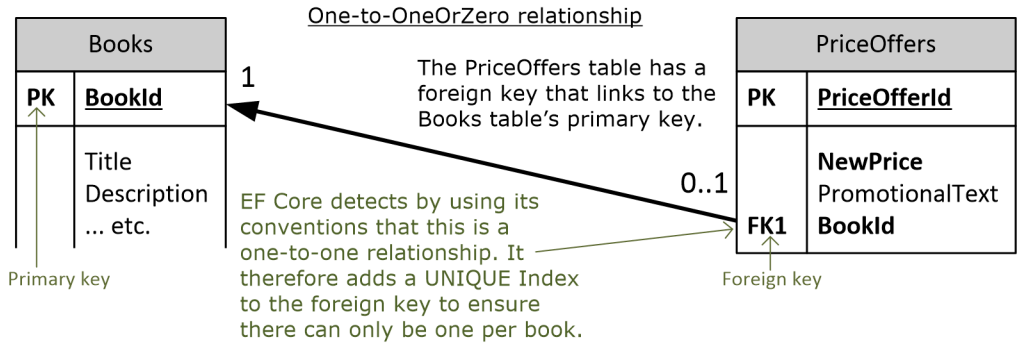


Figure 2.1 This shows the one-to-one relationship between a book and an optional PriceOffer relationship.

To calculate the final price of the book I need to check for the `PriceOffer`. If it's found then the NewPrice would supersede the price in the original book and the PromotionalText will be shown on-screen, for instance:

~~$40~~ $30 Our summer-time price special, for this week only!

ONE-TO-MANY RELATIONSHIP: REVIEWS TO A BOOK

I want to allow customers to review a book; they can give a book a star rating and have the option to leave a comment. Because a books' reviews may range from no reviews to many (unlimited) reviews, it's best to create its own table, which I've called `Review`. The `Books` table has a One-to-Many relationship to the `Review` table—see figure 2.2.

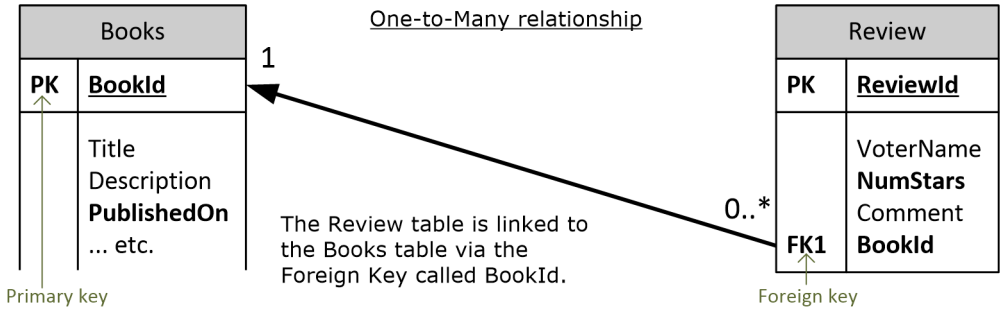


Figure 2.2 This shows the one-to-many relationship between a book and its zero-to-many reviews.

In the summary display I need to do a bit of math on the reviews to show a summary, for instance a typical on-screen display might produce from this one-to-many relationship:

Votes 4.5 by 2 customers

MANY-TO-MANY RELATIONSHIP: BOOKS TO AUTHORS

Books can be written by one or more authors, and an author may be involved in writing one or more books. I need a table called `Books` holding the books data and another table called `Authors` holding the authors. The link between the `Books` and `Authors` tables is called a Many-to-Many relationship, which needs a linking table (see figure 2.3).

The typical on-screen display from this relationship would look like is:

by Dino Esposito, Andrea Saltarello

EF6 An EF6.x user can define a Many-to-Many relationship without needing to define a linking class. For instance, the `BookAuthor` class in listing 2.2. EF 6.x creates a hidden linking table for you. EF Core doesn't create that linking table—you must do that.

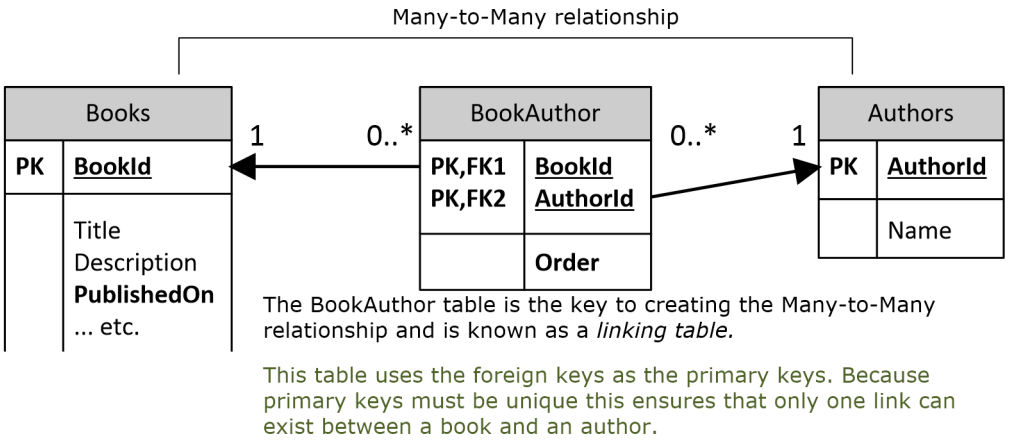


Figure 2.3 This shows the three tables involved in creating the many-to-many relationship between the Books table and the Authors table.

2.1.2 Other relationship types not covered in this chapter

In EF Core, you can include a class in the application's DbContext that inherits from another class in the application's DbContext. For instance, I could've defined the `PriceOffer` class as inheriting the `Book` class. That would've achieved a similar result to the One-to-One relationship I showed earlier. This type of inheritance's supported by EF Core as a table-per-hierarchy (TPH) configuration. I cover this in chapter seven.

Another relationship type's hierarchical- a set of data items that are related to each other by hierarchical relationships. A typical example's an `Employee` class that has a relationship that points to the employee's Manager, who in turn's an Employee. This type of relationship's handled by EF Core using the same approaches as One-to-One and One-to-Many, and I won't provide an example in part 1. I talk more about hierarchical relationships in chapter seven, where I explain how to configure them.

2.1.3 The final database showing all the tables

.Figure 2.4 shows you the complete database for the book-selling site example that I'll be using for the examples. It contains all the tables I've already described, including the full definition of all the columns in the `Books` table.

> **NOTE** The database diagram uses the same layout and terms as in the first chapter, where PK means Primary Key and FK means Foreign Key.

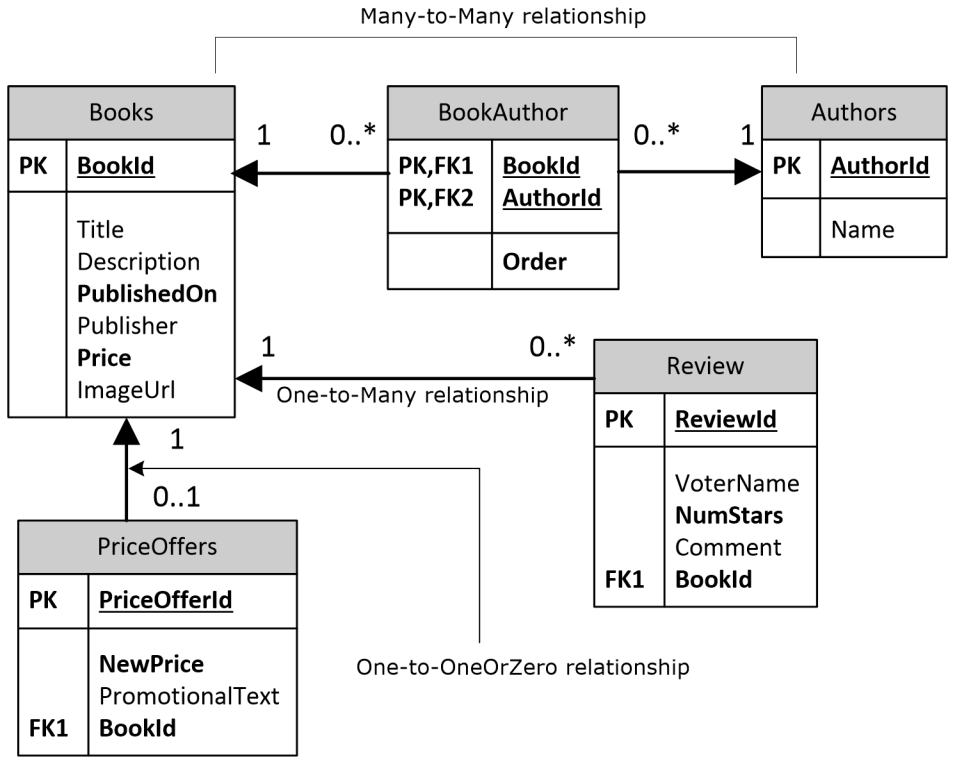


Figure 2.4 The complete relational database schema for the book selling site showing all the tables and their columns.

To help you make sense of this database, figure 2.5 shows the on-screen output of the list of books, but focuses on one book. As you can see, the book-selling site application needs to access every table in the database to build the book list. Later on, I show you this same book display, but with the query that supplies each element.



Figure 2.5 A listing of a single book showing which database table provides each part of the information

TIP You can see a live site running the example book site code at http://efc-oreinaction.com/. This might help you understand the rest of this chapter.

2.1.4 The classes that EF Core maps to the database

I've created five .NET classes to map onto the five tables in my database. They're called `Book`, `PriceOffer`, `Review`, `Author`, and finally `BookAuthor` for the Many-to-Many-linking table.

These classes are referred to as *entity classes* to show that they're mapped by EF Core to the database. From the software point of view there's nothing special about e*ntity classes*—they're normal .NET classes, sometimes referred to as POCO (Plain Old CLR Objects). The term *entity class* identify the class as one that EF Core has mapped to the database.

The primary *entity class* is the `Book` class, shown in listing 2.1. You can see it refers to a single `PriceOffer` class, a collection of `Review` classes, and finally a collection of `BookAuthor` classes, which links the book data to one or more `Author` classes that contain the author's name.

Listing 2.1 The Book class which is mapped to the Books table in the database

```
public class Book       <-- The Book class contains the main book information
{
    public int BookId { get; set; }          <-- I use EF Core's 'by convention' approach to defining the primary key of this entity class. In this case, I use <ClassName>Id, and because the property if of type int EF Core assumes that the database will use the SQL IDENTITY command to create a unique key when a new row is added
    public string Title { get; set; }
    public string Description { get; set; }
    public DateTime PublishedOn { get; set; }
    public string Publisher { get; set; }
    public decimal Price { get; set; }
    /// <summary>
    /// Holds the url to get the image of the book
    /// </summary>
    public string ImageUrl { get; set; }

    //-----------------------------------------------
    //relationships

    public PriceOffer Promotion { get; set; }          <-- This is the link to the optional PriceOffer
    public ICollection<Review> Reviews { get; set; }   <-- There can be zero to many Reviews of the book
    public ICollection<BookAuthor>
        AuthorsLink { get; set; }                      <-- This provides a link to the Many-to-Many linking table that links to the
}
```

For simplicity, I've used EF Core's *by convention* modelling of the database.I use specific names defined by EF Core for the class properties that hold the primary key and foreign keys plus the relationship are defined by the type of the relationships and the type of the foreign key.

In chapters six and seven I describe the other approaches for configuring the EF Core database model.

2.2 Creating the application's DbContext

To access the database, I need to:

1. Define my application's DbContext, which I do by creating a class and inheriting from EF Core's `DbContext` class.
2. Create an instance of that class every time I want to access the database.

All the database queries you'll see later in this chapter use these steps, which I now describe in more detail.

2.2.1 Defining my application's DbContext: EfCoreContext

The key class you need to use EF Core is the application's DbContext. This is a class you define by inheriting EF Core's DbContext and adding various properties to allow your software to access the database tables. It also contains methods you can override to access other features in EF Core, such as configuring the database modelling.

> **NOTE** I'm going skip over configuring the database modelling, done in the `OnModelCreating` method in my application's DbContext. I cover how to model the database in detail in chapters six and seven.

Listing 2.2 shows you the application's DbContext, called `EfCoreContext`. You'll see that I've only provided properties to access the `Books`, `Authors` and `PriceOffers` tables—the other two tables, `Review` and the `BookAuthor` linking table, are accessed via the `Book` class, as you'll see later.

Listing 2.2 The example book setting site's DbContext, called EfCoreContext

```
public class EfCoreContext : DbContext
{
    public DbSet<Book> Books { get; set; }
    public DbSet<Author> Authors { get; set; }
    public DbSet<PriceOffer> PriceOffers { get; set; }

    public EfCoreContext(
        DbContextOptions<EfCoreContext> options)
        : base(options) {}

    protected override void
        OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<BookAuthor>()
            .HasKey(x => new {x.BookId, x.AuthorId});
    }
}
```

The three properties link to the database tables with the same name

This constructor's how the ASP.NET creates an instance of EfCoreContext

I need to tell EF Core about a few things to make the database modelling work.

2.2.2 Creating an instance of my application's DbContext

In chapter one I showed you one way set up the application's DbContext by overriding its `OnConfiguring` method. The downside is that the connection string is fixed. In this chapter I use another approach.I want to use a different database for development and unit testing, and the application's DbContext constructor provides that.

Listing 2.3 shows me providing the options for the database at the time I create my application DbContext called `EfCoreContext`. This listing's based on what I use in my unit testing, as it has the benefit of showing you the component parts. In chapter five, which is about using EF Core in an ASP.NET Core application, I cover a more powerful way to create the application's DbContext using a feature called *dependency injection*.

Listing 2.3 Creating an instance of the applications DbContext to access the database

```
const string connection =
     "Data Source=(localdb)\\mssqllocaldb;"+
     "Database=EfCoreInActionDb.Chapter02;"+
     "Integrated Security=True;";
var optionsBuilder =
    new DbContextOptionsBuilder
           <EfCoreContext>();

optionsBuilder.UseSqlServer(connection);
var options = optionsBuilder.Options;

using (var context = new EfCoreContext(options))
{
    var bookCount = context.Books.Count();
    //... etc.
```

This is the "connection string". Its format's dictated by the sort of database provider and hosting you're using

I need a EF Core DbContextOptionsBuilder<> instance to be able to set the options we need.

I'm accessing a SQL Server database and use the UseSqlServer method from the Microsoft.EntityFrameworkCore.SqlServer library, need the connection string.

This creates the all-important EfCoreContext using the options we've set up. Note that I use a 'using' statement, as the DbContext is disposable, i.e. it should be 'disposed' once you've finished your data access

This code uses the DbContext to find out how many books are in the database.

At the end of listing 2.3 I create an instance of the `EfCoreContext` inside a `using` statement. This is because DbContext has an IDisposable interface and should be disposed after you've used it. From now on, if you see a variable called `context` it was created using the code in listing 2.3, or a similar approach.

2.2.3 Creating a database for your own application

> **TIP** If you're running my example application downloaded from the Git repo that goes with this book, you don't need to use the `Migrate` commands that follows. If run in development mode, it uses the command `EnsureCreated` to create the database. This is less flexible than the `Migrate`, but it doesn't need you to type any commands.

A few ways to create a database using EF Core work, but the normal way's to use EF Core's Migrations feature. This uses your application's DbContext and the *entity classes*, like the ones I've described, as the model for the database structure. The `Add-`

`Migration` command first models your database and then, using that model, builds commands to create a database that fits that model.

Besides creating the database, the great thing about Migrations is that it can update the database with any changes you make in the software. If you change your *entity classes* or any of your application's DbContext configuration, the `Add-Migration` command will build a set of commands to update the existing database.

To use the migration feature you need to install one extra EF Core NuGet libraries in your application called `Microsoft.EntityFrameworkCore.Tools` to your application startup project. This allows you to use the `Migrate` commands in the Visual Studio Package Manager Console (PMC). The ones we need are:

1. `Add-Migration MyMigrationName`
   This creates a set of commands that migrate the database from its current state to a state that matches your application's DbContext and the *entity classes* when you run your command. The `MyMigrationName` shown in the command's the name used for the migration.
2. `Update-Database`
   This will apply the commands created by the Add-Migration command to your database. If there's no database, it creates one. If there's a database that migrate created last time, it will update it.

NOTE You can also use EF Core's command line Interface (CLI) to run these commands—see https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/dotnet. In this book, I use Visual Studio 2017 based commands.

An alternative to using the `Update-Database` command's to call the `context.Database.Migrate()` method in the startup code of your application. This is particularly useful for a ASP.NET Core web application that's hosted—I cover this in chapter five, including some of its limitations.

NOTE Although EF Core's Migrate feature's useful, it doesn't cover all types of database structure changes. Also, for some projects the database's defined and managed outside of EF Core, which means you can't use EF Core's Migrate feature. I explore the various options available for database migration, with their pros and cons, in chapter eleven.

WHAT TO DO IF YOUR APPLICATION USES MULTIPLE PROJECTS?

If your application has a separate project for the application's DbContext from the main, startup application (the book-selling site example application does), then the Add-Migration command's more complex.

In the example book-selling site my application's DbContext is in a project called DataLayer, and my ASP.NET Core application's in a project called EfCoreInAction (I describe why this is later in this chapter). To add an EF Core migration the `Add-Migration` commands would be:

```
Add-Migration Chapter02 -Project DataLayer -StartupProject
[CA] EfCoreInAction
```

You need to provide a way for the Migrations to create an instance of your application's DbContext. In the book-selling site application, the DbContext, `EfCoreContext` has no parameter-less constructor and the `Add-Migration command` will fail. Various ways can get around this. Have a look at the class `ContextFactoryNeededForMigrations` in the DataLayer in the Git repo for one solution.

2.3 Anatomy of a database query

Now we can start looking at how to query a database using EF Core. Figure 2.6 shows an example EF Core database query, with the three main parts of the query highlighted:

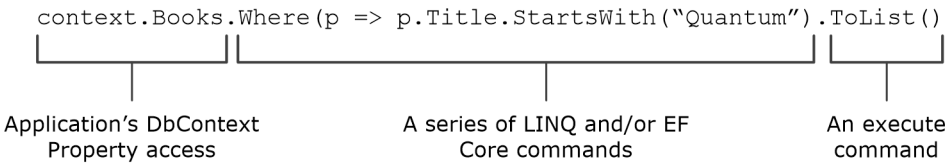


Figure 2.6 This shows the three parts of an EF Core database query, with some example code

TIME SAVER If you're familiar with EF and/or LINQ you could skip this section.

The command shown in 2.6 consists of several methods, one after the other. This is known as a *fluent* interface. Fluent interfaces like this flow logically and intuitively, making them easy to read. I've highlighted the three parts of this overall command and I describe each part in turn:

2.3.1 Application's DbContext Property access

The first part's something which is connected via EF Core to the database. The most common way to refer to a database table's via a `DbSet<T>` property in the application's DbContext, which I've shown in figure 2.6.

I use this DbContext property access throughout this chapter, but in later chapters I introduce other ways to get to a class or property. But the basic idea's the same—you need to start with something that's connected to the database via EF Core.

2.3.2 A series of LINQ/EF Core command

The major part of the command's a set of LINQ and/or EF Core methods that creates the type of query you need—this can range from nothing to complicated. This chapter starts with simple examples of queries, but by the end of it you'll know how to build some complex queries.

NOTE If you aren't familiar with LINQ, then you'll be at a disadvantage in reading this book. I've provided an appendix at the end of this book which gives you a brief overview of LINQ. You can also find plenty of online resources; see https://msdn.microsoft.com/en-us/library/bb308959.aspx.

2.3.3 The execute command

The last part of the command reveals something about LINQ. Until a final execute command at the end's used, the LINQ is held as a series of commands; it hasn't been executed on the data yet. This means that EF Core can 'translate' each command in the LINQ query into the correct commands to use for the database you're using. In EF Core a query's executed against the database when

- It's enumerated by a `foreach` statement.
- It's enumerated by a collection operation such as `ToArray`, `ToDictionary`, `ToList`, or `ToListAsync`.
- LINQ operators, such as `First` or `Any`, are specified in the outermost part of the query.
- When you use certain EF Core commands, like `Load()` which I use in the *explicit loading* of relationship later in this chapter.

2.4 The three ways of loading related data

I've shown you the `Book` *entity class*, which has links to three other *entity classes*: `PriceOffer`, `Review` and `BookAuthor`. I now want to deal with how you as a developer can access the data behind these relationships. There are three ways, known as: *eager loading*, *explicit loading* and *select loading*, with a subsection dedicated to each type.

Before I cover these approaches, you need to be aware that EF Core won't load any relationships in an *entity class* unless you ask it to. This means if I load a `Book` class then, by default, each of the relationship properties in the `Book` *entity class*, `Promotion`, `Reviews` and `AuthorsLink` will be null.

This default behavior of not loading relationships is correct, as it means that EF Core minimizes the database accesses. If you want to load a relationship, then you need to add some code to tell EF Core to do that. The next three sections describe the three different approaches, with their pros and cons, to get EF Core to load a relationship.

2.4.1 Eager loading: loading relationships with the primary entity class

The first approach to loading related data's *eager loading*. *Eager loading* entails telling EF Core to load the relationship in the same query that loads the primary *entity class*. *Eager loading* is specified via two fluent methods, `Include()` and `ThenInclude()`. Listing 2.4 shows the loading of the first row of the `Books` table as an instance of the `Book` *entity class*, and the *eager loading* of the single relationship, `Reviews`.

Listing 2.4 Eager loading of first book with the corresponding Reviews relationship

```
var book = context.Books
    .Include(r => r.Reviews)
    .First();
```

The Include() gets a collection of Reviews, which may be an empty collection

This takes the first book

If you look at the SQL command this EF Core query creates, which is shown in the following snippet, you'll see two SQL commands. The first loads the first row in the `Books` table and the second loads the reviews where the foreign key, `BookId`, has the same value as the first `Books` row primary key.

```
-- First SQL command to get the first row in the Books table
SELECT TOP(1)
    [r].[BookId], [r].[Description], [r].[ImageUrl],
    [r].[Price], [r].[PublishedOn], [r].[Publisher],
    [r].[Title]
FROM [Books] AS [r]
ORDER BY [r].[BookId]
-- Second SQL command to get the reviews for this book
SELECT [r0].[ReviewId], [r0].[BookId],
       [r0].[Comment], [r0].[NumStars], [r0].[VoterName]
FROM [Review] AS [r0]
INNER JOIN (
    SELECT DISTINCT TOP(1) [r].[BookId]
    FROM [Books] AS [r]
    ORDER BY [r].[BookId]
) AS [r1] ON [r0].[BookId] = [r1].[BookId]
ORDER BY [r1].[BookId]
```

EF6 *Eager loading* is like EF Core as EF6.x, but with improved syntax and a different SQL implementation. First syntax: In EF6.x there isn't a `ThenInclude()` method, and you must use Select(), for example `Books.Include(p => p.AuthorLink.Select(q => q.Author)`. Second SQL implementation: In EF6.x it'd try to load all the data in one query, including collections. This can be inefficient and EF Core loads collections in a separate query—you can see this in the SQL snippet produced by the code in listing 2.4

Now let's look at a more complex example in listing 2.5 which shows a query to get the first book, with *eager loading* of all its relationships—in this case `AuthorsLink` and the second-level `Author` table, the `Reviews` and the optional `Promotion`.

Listing 2.5 Eager loading of the Book class and all of the related data

```
var book = context.Books
    .Include(r => r.AuthorsLink)
        .ThenInclude(r => r.Author)

    .Include(r => r.Reviews)
    .Include(r => r.Promotion)
    .First(); //
```

The first Include() gets a collection of BookAuthor

The ThenInclude() gets the next link, in this case the link to the Author

The Include() gets a collection of Reviews, which may be an empty collection

This loads any optional PriceOffer class, if one's assigned

This takes the first book

Listing 2.5 shows the use of *eager loading* method `Include()` to get `AuthorsLink` relationship. This is a *first level* relationships, which are relationships referred to directly from the *entity class* you're loading. That `Include()` is followed by a `ThenInclude()` to load the *second level* relationship, in this case the `Author` table at the other end of the linking table `BookAuthor`. This `Include()` followed by a `ThenInclude()` pattern's the standard way of accessing relationships that go deeper than a first level relationship. You can go to any depth with multiple `ThenInclude()`s one after the other.

If the relationship doesn't exist, like the optional PriceOffer class pointed to by the `Promotion` property in the `Book` class, then `Include()` doesn't fail—nor does it load anything, or in the case of collections, it returns an empty collection; a valid collection but with zero entries. This applies to `ThenInclude()` as well, and if the previous `Include()` or `ThenInclude()` was empty, then subsequent `ThenInclude()`s are ignored.

Eager loading has the advantage that EF Core will load all the data referred to by the `Include()` and `ThenInclude()` in an efficient manner, using the minimum of database accesses, called *database round trips*. I find this type of loading useful in relational updates where I need to update an existing relationship—chapter three cover this. I also find *eager loading* useful in business logic—chapter four cover this in much more detail.

The downside is that it loads *all* the data, when sometimes you don't need a part of that data, for instance the book list display doesn't need the book description, which could be quite large.

2.4.2 Explicit loading: loading relationships after the primary entity class

The second approach to loading data is *explicit loading*, where, after you've loaded the primary *entity class*, you can explicitly load any other relationships you want. Listing 2.6 shows a series of commands that first loads the book and then uses *explicit loading* commands to read all the relationships.

Listing 2.6 Explicit loading of the Book class and some related data

```
var book = context.Books.First();
context.Entry(book)
    .Collection(c => c.AuthorsLink).Load();
foreach (var authorLink in book.AuthorsLink)
{
    context.Entry(authorLink)
        .Reference(r => r.Author).Load();
}
context.Entry(book)
    .Collection(c => c.Reviews).Load();
context.Entry(book)
    .Reference(r => r.Promotion).Load();
```

This reads in the first book on its own

This explicitly loads the linking table, BookAuthor

To load all the possible Authors it loops through all the BookAuthor entries and loads each linked Author class

This loads all the Reviews

This loads the optional PriceOffer class

Explicit loading has an extra command that allows a query to be applied to the relationship, rather than only loading it. The example in listing 2.7 shows use of the *explicit loading* method `Query()` to obtain the count of the number of reviews and also load all

the star ratings of each review. You can use any standard LINQ command after the `Query()` method, for instance `Where` or `OrderBy`.

Listing 2.7 Explicit loading of the Book class with refined set of related data

```
var book = context.Books.First();//
var numReviews = context.Entry(book)//
    .Collection(c => c.Reviews)     //
    .Query().Count();               //
var starRatings = context.Entry(book)//
    .Collection(c => c.Reviews)      //
    .Query().Select(x => x.NumStars) //
    .ToList();                       //
```

This reads in the first book on its own

This executes a query to count how many reviews there are for this book

This executes a query to get all the star ratings for the book

The advantage of *explicit loading* is that you can load a relationship of an *entity class* later. I've found this useful using a library that only loads the primary *entity class* when I need one of its relationships. *Explicit loading* can also be useful if you only need that related data in some circumstances. You might also find *explicit loading* useful in complex business logic, as you can leave the job of loading the specific relationships to the parts of the business logic that needs it.

The downside of *explicit loading* is that there are more *database round trips*, which can be inefficient. If you know up front what data you need, it's normally more efficient to eager load the data to take the minimum database round trips.

2.4.3 Select loading: loading the specific parts of primary entity class and any relationships

The third approach to loading data's to use the LINQ `Select` method to specifically pick out the data you want, which I call *select loading*. Listing 2.8 shows the use of the Select method to select a few standard properties from the Book class and execute specific code inside the query to get the count of customer reviews for this book.

Listing 2.8 Select of the Book class picking specific properties and one calculation

```
var result = context.Books
    .Select(p => new
        {
            p.Title,
            p.Price,
            NumReviews
               = p.Reviews.Count,
        }
    ).First();
```

This uses the LINQ select keyword and creates an anonymous type to hold the results

These are simple copies of a couple of properties

This runs a query that counts the number of reviews

The advantage of the select query approach is that only the data you need is loaded, which can be more efficient if you don't need all the data. In the case shown in listing 2.8 it only takes one SQL SELECT command to get all that data, which is also efficient in terms of database round-trips. In fact EF Core turns the `p.Reviews.Count` part of the query into a SQL command, and that count's done inside the database, as you can see in the following listing of the SQL created by EF Core.

```
SELECT TOP(1) [p].[Title], [p].[Price], (
    SELECT COUNT(*)
    FROM [Review] AS [r0]
    WHERE [p].[BookId] = [r0].[BookId]
)
FROM [Books] AS [p]
```

The downside to the *select loading* approach is that you need to write code for each property/calculation you want. But, later in this book, I show a way you can automate this.

> **NOTE** You'll see a much more complex select loading example later in this chapter, as I use this type of loading to build the book list query for the book-selling site.

Lazy loading: loading relationships when you need them—coming sometime?

I can't write this section without mentioning lazy loading. This is a feature in EF6.x which allows you to mark a property as virtual, and the database access occurs only when you read that property. Lazy loading is, at the time of writing this book, not in EF Core, but there've been numerous requests to put it back—see https://github.com/aspnet/EntityFramework/issues/3797.

The proponents of lazy loading say that it's easy to use, because you don't need the application's DbContext when you read the property. The downside is that you get a database access for every property you access. I should say that I don't use lazy loading because it can be extremely inefficient.

I believe lazy loading may be added to EF Core because of the requests from developers, but I've no idea in what form. I'd recommend you look at the approaches I use in this book, which should allow you to work round any issues you have with the loss of lazy loading.

2.5 Client vs. Server evaluation: moving part of your query into software

All the queries you've seen are ones that EF Core can convert to commands that can be run on the database server. But EF Core has a feature called *Client vs. Server evaluation* which allows you to include methods in your query that can't be run on the data-

base, for example on relational databases methods that EF Core can't convert to SQL commands. EF Core runs these non-server runnable commands after the data returns from the database. Let me show you an example and then provide you with a diagram to show you what's happening inside EF Core to make *Client vs. Server evaluation* work.

EF6 *Client vs. Server evaluation* is a new feature in EF Core, and a useful one too.

2.5.1 An example of using Client vs. Server evaluation to create the display string of s book's authors

For the list display of the books on the book-selling site web site I need to a) extract all the author's names in order from the `Authors` table and b) turn them into one string with commas between each name. Listing 2.9 shows an example which loads two properties, `BookId` and `Title`, in the normal manner and a third property, `AuthorsString`, which uses *client vs. server evaluation.*

Listing 2.9 Showing a Select query that includes a non-SQL command, string.Join

```
var book = context.Books
    .Select(p => new
    {
        p.BookId,
        p.Title,
        AuthorsString = string.Join(", ",
            p.AuthorsLink
            .OrderBy(q => q.Order)
            .Select(q => q.Author.Name)),
    }
    ).First();
```

These parts of the select can be converted to SQL and run on the server

The String.Join is executed on the client in software

The result of running this on a book that had two authors, Jack and Jill, would cause the `AuthorsString` to contain "Jack, Jill", and the `BookId` and `Title` would be set to the value of the corresponding columns in the `Books` Table.

Figure 2.7 gives you a view of how listing 2.9 is processed through four stages. The one I want to focus on is stage ❸, where EF Core runs the client-side code that it couldn't convert into SQL.

The Client vs. Server evaluation feature gives you, as a developer, the ability to create complex queries, and EF Core optimizes the query to run as much as it can on the database Server. But if there's some method in your query that can't be run on the database server then the query won't fail, but EF Core will apply that method after SQL Server has done its part.

The example shown in listing 2.9 is simple, but the possibilities are endless. You should watch out for a few things.

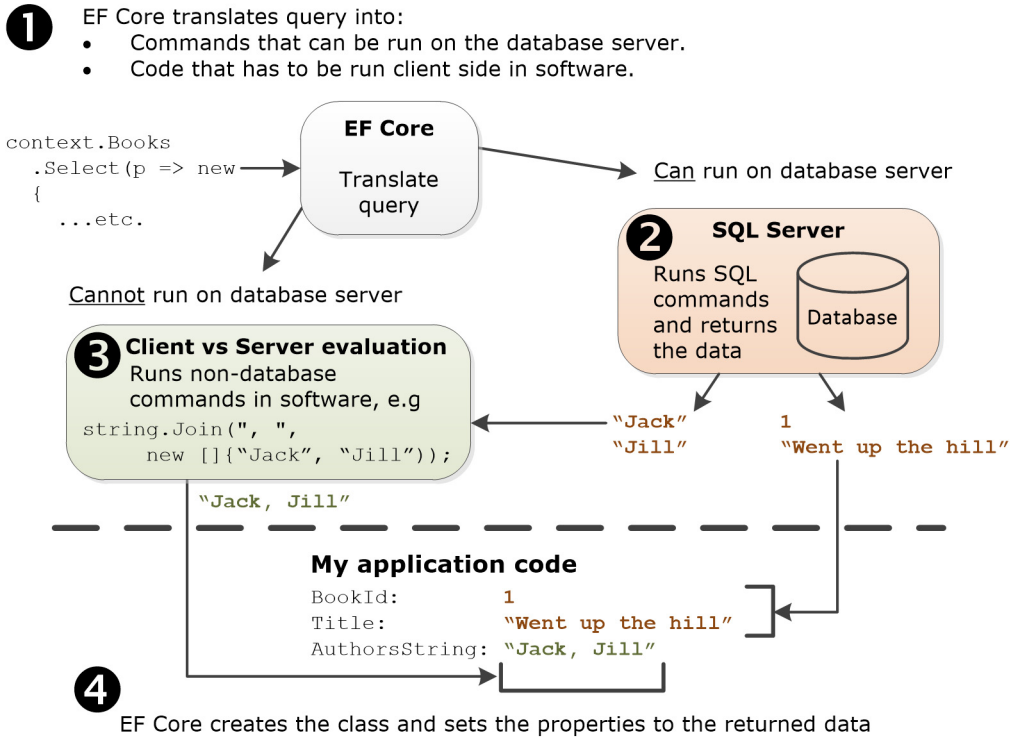


Figure 2.7 This shows what parts of the query are converted to SQL and run in the SQL server ❷, and the part, in this case the `string.Join`, that had to be done client-side, ❸, by EF Core before the combined result, ❹, is handed back to my application code.

2.5.2 Understanding the limitations of Client vs. Server evaluation

I think the *Client vs. Server evaluation* feature is a useful addition to EF, but, like all powerful features, it's best to understand what's going on to use it in the right way.

Firstly, the obvious thing's the method you provide is run on every entry (row) you read from the database. If you've 10,000 rows in the database and don't filter/limit what's loaded then, as well as a SQL command that takes a long time, your processor will spend a long time running your method 10,000 times.

The second point's subtler: The *Client vs. Server evaluation* feature blurs the lines between what's run in the database and what's run in the client. It's possible to create a query that works, but which is slower than it could be because it had to use client-side evaluation. To give you some context, in EF6.x this form of mixed client/server query would've failed because it didn't support that. That meant in EF6.x you had to do something about it—often by changing the query to better suit the database. Now your query may work, but perform worse than one you write such that EF Core can convert it directly to SQL commands.

One extreme example of the problem's that *Client vs. Server evaluation* allows you to sort on a client-side evaluated property, which means the sorting is done in the client rather than in the database server. I tried this by replacing the `.First()` command with `.Sort(p => p. AuthorsString)` in listing 2.9 and returning a list of books. In that case EF Core produces SQL code that reads all the books, then reads each row individually, twice, which isn't optimal.

My experiments with *Client vs. Server evaluation* showed that EF Core's quite intelligent and builds an optimal SQL query for all the sensible cases I gave it; maybe this isn't such a big worry. I suggest you use it and performance tune later (see chapter twelve on finding and improving database performance).

TIP You can use EF Core's logging to identify possible bad performing *Client vs. Server* queries. EF Core logs a warning on the first use of a *Client vs. Server* query that can adversely effect the query. Also, you can configure logging to throw an exception on a *Client vs. Server* query warnings—for more information see https://docs.microsoft.com/en-us/ef/core/querying/client-eval#disabling-client-evaluation.

2.6 Building complex queries—the book-selling site

Having covered the basics of querying the database, let's look at some examples that are more common in real applications. I'm going to build a query to list all the books on the book-selling site with a range of features like sorting, filtering, and paging.

2.6.1 Building the book list query using select loading

We could build the book display by using *eager loading*; we could load all the data and combine the authors, calculate the actual price, or calculate the average votes with our code. The problem with that approach is that the book list query includes various sorting options, such as on price, and filtering options, for instance only showing books with four or more customer star ratings.

With *eager loading*, I could load ALL the books and then, in memory, sort or filter the books. For our chapter two book-selling site, which has fifty books, that'd work, but I don't think that approach would work for Amazon! The better solution's for the values to be calculated inside SQL Server where sorting and filtering can be done before the data's returned to the application.

Although I could add sorting and filtering methods in front of *eager loading* (or *explicit loading*), I've chosen to use a *select loading* approach, where I combine all the individual queries into one big select query. This select precedes the sorting, filtering, and paging parts of the query. That way EF Core knows, via the select query, how to load each part of the query and can therefore use any property in the LINQ select in a SQL `ORDER BY` (sort) or SQL `WHERE` (filter) clause as it needs to.

NOTE I use a Client vs. Server evaluation to get the string containing the author(s) of the book. That excludes, for performance reasons, that property from being used in a SQL sort or filter command.

Before I show you the select query that loads the book data, let's go back to the book list display of the book "Quantum Networking" that I showed near the beginning of this chapter, but this time figure 2.8 shows each individual LINQ query that's needed to get each piece of data.

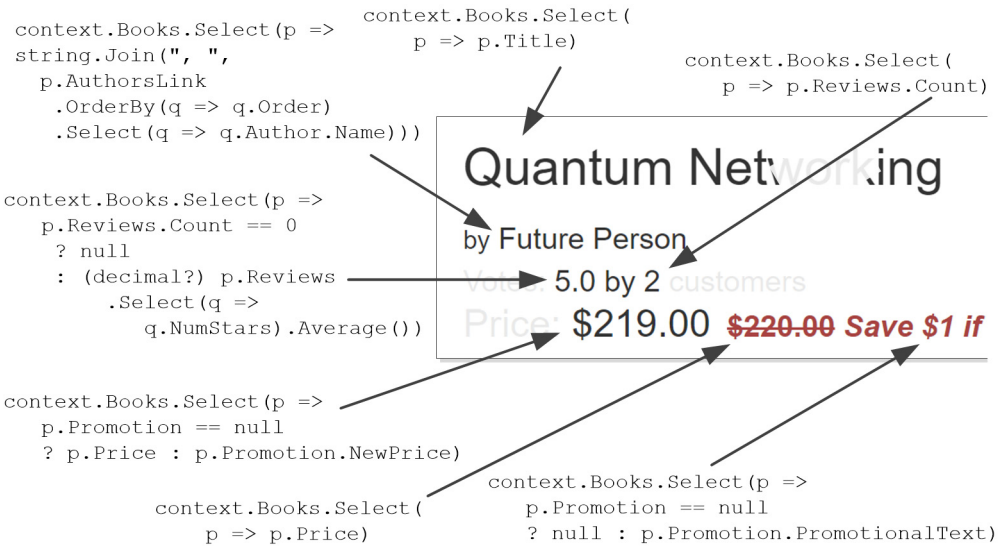


Figure 2.8 Showing each individual query needed to build the book list display

This diagram, shown in figure 2.8, is complicated because the queries needed to get all the data are complicated. But with this diagram in mind, let's look at how we build the book select query.

I start with the class we're going to put the data in. This type of class, which only exist to bring together the exact data we want, are referred to in many ways. In ASP.NET they're referred to as a ViewModel, but the term ViewModel has other connotations and uses. I therefore refer to this type of class as a Data Transfer Object (DTO). Listing 2.10 shows you the DTO class, `BookListDto`.

DEFINITION The term Data Transfer Object (DTO) is used to describe "an object that carries data between processes" (Wikipedia) or "object that is used to encapsulate data, and send it from one subsystem of an application to another" (stackoverflow answer). The usage of the DTO class in this book-selling site example's closer to the stackoverflow answer.

Listing 2.10 The DTO (Date Transfer Object) BookListDto

```
public class BookListDto
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public DateTime PublishedOn { get; set; }
    public decimal Price { get; set; }
    public decimal
        ActualPrice { get; set; }
    public string
        PromotionPromotionalText { get; set; }
    public string AuthorsOrdered { get; set; }

    public int ReviewsCount { get; set; }
    public decimal?
        ReviewsAverageVotes { get; set; }
}
```

I need the Primary Key if the customer clicks the entry to buy the book

Although the publish date isn't shown, we'll want to sort by it, which is why we must include it

This is the normal Price

This is the selling price—either the normal price, or the promotional.NewPrice if present

The promotional text to show if there's a new price

An array of the authors' names in the right order

The number of people who reviewed the book

The average of all the Votes—null if no votes

To work with EF Core's s*elect loading*, the class that's going to receive the data must have a default constructor (it can be created without needing to provide any properties to the constructor), the class mustn't be static and the properties must have public setters.

Next, we build a select query that fills in every property in the BoolListDto. Because I want to use this with other query parts, like sort, filter, and paging, I use the `IQueryable<T>` type to create a method called `MapBookToDto` that takes in `IQueryable<Book>` and returns `IQueryable<BookListDto>`. Listing 2.11 shows this method and, as you can see, the LINQ Select pulls together all the individual queries you saw in figure 2.8.

Listing 2.11 The Select query to fill the BookListDto

```
public static IQueryable<BookListDto>
    MapBookToDto(this IQueryable<Book> books)
{
    return books.Select(p => new BookListDto
        {
            BookId = p.BookId,
            Title = p.Title,
            Price = p.Price,
            PublishedOn = p.PublishedOn,
            ActualPrice = p.Promotion == null
                ? p.Price
                : p.Promotion.NewPrice,
            PromotionPromotionalText =
                p.Promotion == null
                  ? null
                  : p.Promotion.PromotionalText,
            AuthorsOrdered = string.Join(", ",
                p.AuthorsLink
                .OrderBy(q => q.Order)
                .Select(q => q.Author.Name)),
```

This method takes in IQueryable<Book> and returns IQueryable<BookListDto>

These are simple copies of existing columns in the Books table

This calculates the selling price, which is the normal price, or the promotion price if that relationship exists

The PromotionalText depends on whether a PriceOffer exists for this book

This obtains an array of Authors' names, in the right order. We're using a Client vs. Server evaluation as we want the author's names combined into one string

```
            ReviewsCount = p.Reviews.Count,
            ReviewsAverageVotes =
                p.Reviews.Count == 0
                ? null
                : (decimal?) p.Reviews
                    .Select(q => q.NumStars).Average()
    });
}
```

We need to calculate how many reviews there are

We can't calculate the average of zero reviews, and we need to check the count first

NOTE The individual parts of the Select query in listing 2.11 are the repetitive code I mention in "my lightbulb moment" section of chapter one. In chapter ten I introduce Mappers to automate much of this coding, but in part one I'm going to list all the code in full to show the whole picture. But be assured—there's a way to automate the *select loading* approach of querying which improves productivity.

The `MapBookToDto` method's using a pattern known as a *query object*. This pattern's all about encapsulating a query, or part of a query, in a method. That way the query's isolated in one place, which makes it easier find, debug and performance tune. I'll use the *query objects* pattern for the sort, filter, and paging parts of the query too.

NOTE I think *query objects* are a useful pattern for building queries like the book list in this example, but there are alternative approaches, such as the repository pattern. I cover this topic in more detail in chapter ten, which is all about the different patterns that can be used with EF Core.

The `MapBookToDto` method's also what .NET calls an extension method. Extension methods allow you to chain *query objects* together. You'll see this chaining used later when I combine each individual part of the book list query to create the final, composite query.

NOTE A method can become an extension method if a) it is declared in a static class, b) the method is static, and c) the first parameter has the keyword `this` in front of it.

Because the `MapBookToDto` method uses `IQueryable<T>` for both input and output the LINQ commands inside the method doesn't get executed. This means the input can be the `DbSet<Books>` property in the application's DbContext, or another source of type `IQueryable<Book>`. Also, the `MapBookToDto` method's output can either be fed into a method that takes `IQueryable<BookListDto>` and returns `IQueryable<BookListDto>`, in which case the LINQ commands are still un-executed.

EF Core turns this into a reasonable query, but certainly not the best possible performing SQL query, but for now it's sufficient. In chapter thirteen on performance tuning I use this query in a worked example of the techniques you can use to improve a query.

NOTE You can see the results of this query by cloning the code from the Git repo, selecting the Chapter02 branch, and then running the EfCoreInAction web application locally. I provide a 'Logs' menu feature which shows you the SQL used to load the book list with the specific sorting, filtering, and paging setting you've selected.

2.6.2 Introducing the architecture of the book-selling site application

I've waited until this point to talk about the design of the book-selling site application, because it should make more sense now that we've created the `BookListDto` class. At this stage, we've the *entity classes*, for example `Book`, `Author`, which maps to the database via EF Core. We also have a `BookListDto` class, which holds the data in the form that the presentation side needs, in this case a ASP.NET Core web server.

In a simple example application, we might put the *entity classes* in one folder and the DTOs in another. But even with a small application like this book-selling site, this can be quite confusing as the approach you use with the database is different to the approach you use when displaying data to the customer. It's all about what's called separation of concerns—see https://en.wikipedia.org/wiki/Separation_of_concerns.

I could've split up the parts of the book selling site application in many ways, but I used a common design called *layered architecture*. The *layered architecture* approach works well for small-to-medium .NET web applications. Figure 2.9 shows you the architecture of the book-selling site for this chapter.

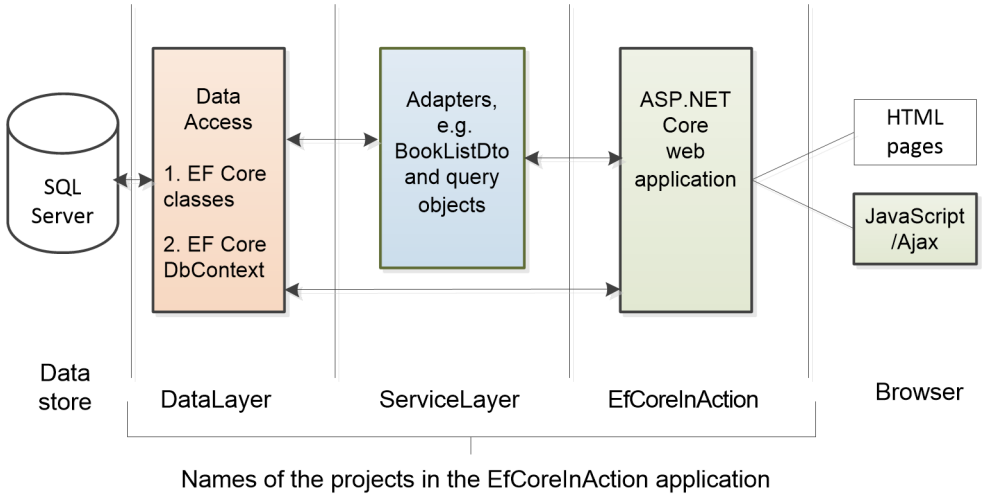


Figure 2.9 Diagram of the layered architectural approach for the example book-selling web site.

Each of the three large rectangles are .NET projects, with their names at the bottom of the figure. I've split the classes and code between these three projects in the following way:

- **DataLayer:**
  This layer's focus is the database access. The *entity classes* and the application's DbContext are in this project. It doesn't know anything about the layers above.
- **ServiceLayer:**
  This is the layer that acts as an adapter between the DataLayer and the ASP.NET Core web application. It does this using DTOs, *query objects* and various classes to run the commands. The idea is that the front-end ASP.NET Core layer has enough to do that the ServiceLayer hands it pre-made data for display.
- **EfCoreInAction:**
  The focus of this layer's on presenting data in a way that's convenient and applicable to the user. That is a challenge, which is why I move as much of the database and data adapting to the ServiceLayer. In the example book-selling site, I'm using an ASP.NET Core web application mainly serving html pages, with a small amount of JavaScript running in the browser.

Using a layered architecture makes the book-selling site example more complex to understand, but it's how real applications are built. It also means you can more easily know what each bit of the code's supposed to be doing in the associated Git repo as the code isn't all tangled up together.

2.7 Adding Sorting, Filtering, and Paging to the book-selling site

With the project structure out of the way, we can now push on more quickly and build the remaining *query objects* to create the final book list display. Let me start by showing you a screen shot in figure 2.10 of the book-selling site's sort, filter, and page controls to give you an idea of what we're implementing.

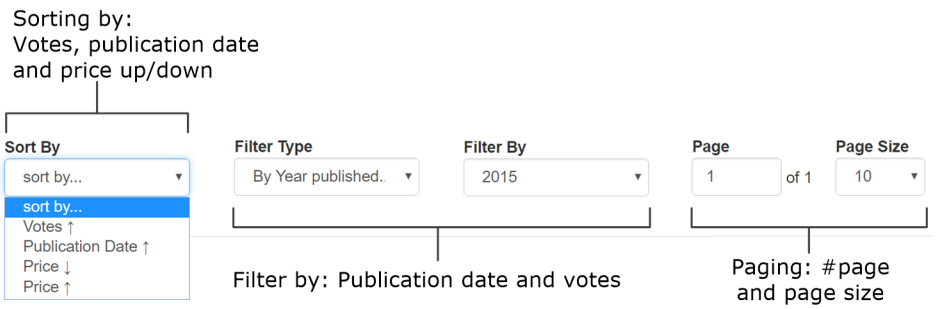


Figure 2.10 The three commands, sorting, filtering, and paging, as shown on the example book-selling site.

2.7.1 The sorting of books by price, publication date and customer ratings

Sorting in LINQ is done by the two methods: `OrderBy` and `OrderByDescending`. I created a query object called `OrderBooksBy` as an extension method– see listing 2.12.

You'll see that with the `IQueryable<BookListDto>` parameter it also takes in an enum parameter. This enum defines the type of sort the user wants.

Listing 2.12 The OrderBooksBy query object method

```
public static IQueryable<BookListDto> OrderBooksBy
    (this IQueryable<BookListDto> books,
     OrderByOptions orderByOptions)
{
    switch (orderByOptions)
    {
        case OrderByOptions.SimpleOrder:
            return books.OrderByDescending(
                x => x.BookId);
        case OrderByOptions.ByVotes:
            return books.OrderByDescending(x =>
                x.ReviewsCount > 0
                      ? x.ReviewsAverageVotes : 0);
        case OrderByOptions.ByPublicationDate:
            return books.OrderByDescending(
                x => x.PublishedOn);
        case OrderByOptions.ByPriceLowestFirst:
            return books.OrderBy(x => x.ActualPrice);
        case OrderByOptions.ByPriceHigestFirst:
            return books.OrderByDescending(
                x => x.ActualPrice);
        default:
            throw new ArgumentOutOfRangeException(
                nameof(orderByOptions), orderByOptions, null);
    }
}
```

Because of paging we always need to sort. I default to showing latest entries first

This orders the book by votes. Books without any votes go at the bottom

Order by publication date—latest books at the

Order by actual price, which considers promotional price—both lowest first and highest first

Calling the `OrderBooksBy` method returns the original query with the appropriate LINQ sort command added to the end. We then pass this onto the next query object, or, if we're finished, we call a command to execute the code, like ToList.

> **NOTE** You'll see that even if the user doesn't select a sort, I still sort (see `SimpleOrder` switch statement). This is because I'll be using paging, which provides a page at a time rather than all the data, and SQL requires the data to be sorted to handle paging. The most efficient sort's on the primary key, and I sort on that.

2.7.2 The filtering of books by publication year and customer ratings

The filtering I created for the book-selling site's a bit more complex than the sorting we covered. That's because I get the customer to first select the type of filter they want and then select the actual filter value. The filter value for votes is easy, it's a set of fixed values—4 or above, 3 or above, and so on. But for the filter by date I need to find the dates of the publications to put into the dropdown list.

It's instructive to look at the code for working out the years that have books, as it's a nice example of combining several LINQ commands to create the final dropdown

list. Listing 2.13 shows a snippet of code taken from the `GetFilterDropDownValues` method.

Listing 2.13 The code to produce a list of the years that we've published books

```
var comingSoon = _db.Books.
    Any(x => x.PublishedOn > DateTime.UtcNow);
var nextYear = DateTime.UtcNow.AddYears(1).Year;
var result = _db.Books
    .Select(x => x.PublishedOn.Year)
    .Distinct()
    .Where(x => x < nextYear)
    .OrderByDescending(x => x)
    .Select(x => new DropdownTuple
    {
        Value = x.ToString(),
        Text = x.ToString()
    }).ToList();
if (comingSoon)
    result.Insert(0, new DropdownTuple
    {
        Value = BookListDtoFilter.AllBooksNotPublishedString,
        Text = BookListDtoFilter.AllBooksNotPublishedString
    });

return result;
```

This returns true if there's a book in the list which isn't yet published

This gets next year and we can filter out all future publications

This long command gets the year of publication, uses distinct to only have one of each year, filters out the future books and orders with newest year at the top

I finally use two client/server evaluations to turn the values into strings

Finally I add a "coming soon" filter for all the future books

The result of the code in listing 2.13 is a list of Value/Text pairs holding each year that we've published books, plus a "coming soon" section for books yet to be published. This is turned into a HTML dropdown list by ASP.NET Core and sent to the browser.

I've listed the filter query object called `FilterBooksBy` in listing 2.14. This takes as an input the 'Value' part of the dropdown list you saw created in listing 2.13, plus whatever type of filtering the customer has asked for.

Listing 2.14 The FilterBooksBy query object method

```
public static IQueryable<BookListDto> FilterBooksBy(
    this IQueryable<BookListDto> books,
    BooksFilterBy filterBy, string filterValue)
{
    if (string.IsNullOrEmpty(filterValue))
        return books;

    switch (filterBy)
    {
        case BooksFilterBy.NoFilter:
            return books;
        case BooksFilterBy.ByVotes:
            var filterVote = int.Parse(filterValue);
            return books.Where(
                x => x.ReviewsCount > 0
                  && x.ReviewsAverageVotes > filterVote);
```

The method's given both the type of filter and the user selected filter value

If the filter value isn't set then it returns the IQueryable with no change

Same for no filter selected—it returns the IQueryable with no change

The filter by votes is a value and above, for example 3 and above. We also ignore books with no reviews

```
            case BooksFilterBy.ByPublicationYear:
                if (filterValue == AllBooksNotPublishedString)
                    return books.Where(
                        x => x.PublishedOn > DateTime.UtcNow);

                var filterYear = int.Parse(filterValue);
                return books.Where(
                    x => x.PublishedOn.Year == filterYear
                      && x.PublishedOn <= DateTime.UtcNow);
            default:
                throw new ArgumentOutOfRangeException
                    (nameof(filterBy), filterBy, null);
        }
    }
```

If the "coming soon" was picked then we only return books not yet published

If we've a specific year we filter on that. Note that we also remove future books (in case the user chose this year's date)

I could've created loads of other types of filters/search of books– search by title, or books between $20-$40, but this gives you a feel for how filtering works. It's all about finding the right value to filter/search on and then using the right combination of LINQ commands to achieve the result you want.

2.7.3 The paging of the books in the list

If you used Google search, then you've used paging—Google presents the first dozen results and you can 'page' through the rest. The book-selling site uses paging, which is simple to implement use the LINQ commands `Skip` and `Take` methods.

In fact, although the other *query objects* were tied to the BookListDto class because the LINQ paging commands are simple, I can create a generic paging *query object* which works with any `IQueryable<T>` query. This *query object* is shown in listing 2.15, but it does rely on getting a page number in the right range, but another part of my application does that anyway to show the correct paging information on screen.

Listing 2.15 A generic Page query object method

```
public static IQueryable<T> Page<T>(
    this IQueryable<T> query,
    int pageNumZeroStart, int pageSize)
{
    if (pageSize == 0)
        throw new ArgumentOutOfRangeException
            (nameof(pageSize), "pageSize cannot be zero.");

    if (pageNumZeroStart != 0)
        query = query
            .Skip(pageNumZeroStart * pageSize);

    return query.Take(pageSize);
}
```

It skips the correct number of pages

It then takes the number for this page size

As I said earlier, paging only works if the data's ordered, otherwise SQL Server throws an exception. This is because relational databases don't guarantee the order in which data's handed back, and there's no default row order in a relational database.

2.8 Putting it all together: how to combine query objects

I've covered each *query object* we need to build book list for the book-selling site. Now it's time to see how we combine each of these *query objects* to create a composite query to work with the web site. The benefit of building a complex query as separate parts is that it makes writing and testing the overall query simpler, as you can test each part on its own.

Listing 2.16 shows a class called ListBooksService, which has one method, SortFilterPage, which uses all the *query objects*, select, sort, filter, and page, to build the composite query. It needs the application's `DbContext` to access the `Books` property, which we provide via the constructor.

Listing 2.16 The ListBookService class which provides a sorted, filtered, and paged list

```
public class ListBooksService
{
    private readonly EfCoreContext _context;

    public ListBooksService(EfCoreContext context)
    {
        _context = context;
    }

    public IQueryable<BookListDto> SortFilterPage
        (SortFilterPageOptions options)
    {
        var booksQuery = _context.Books
            .AsNoTracking()
            .MapBookToDto()
            .OrderBooksBy(options.OrderByOptions)
            .FilterBooksBy(options.FilterBy,
                           options.FilterValue);

        options.SetupRestOfDto(booksQuery);

        return booksQuery.Page(options.PageNum-1,
                               options.PageSize);
    }
}
```

Because this is a read-only query, I add .AsNoTracking(). It makes the query faster

This starts by selecting the Books property in the Application's DbContext

It then uses the Select query object which will pick out/calculate the data it needs

It then adds the commands to order the data using the given options

Then it adds the commands to filter the data

This stage sets up the number of pages and makes sure PageNum is in the right range

Finally it applies the paging commands

At you can see the four *query objects*, select, sort, filter, and page, are added in turn (called chaining) to form the final composite query. Note that the `options.SetupRestOfDto(booksQuery)` code before the Page query object sorts things out such as how many pages there are, ensures that the PageNum is in the right range, and a few other housekeeping items.

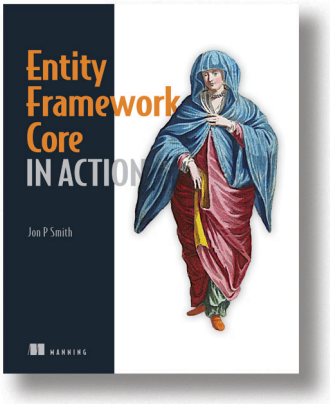
2.9 Summary

I started this chapter by describing the various types of relationships in the example book-selling site example application, and showed you how to set up an application's DbContext to access and create a database. From that base, I could start talking about querying a database, with progressively more complex versions of queries. Takeaways from this chapter are:

- To access a database in any way via EF Core, you need to define an application DbContext.
- An EF Core query consists of three parts: The application's DbContext property, a series of LINQ/EF Core commands and finally a command to execute the query.
- Using EF Core you can model three primary database relationships: One-to-One, One-to-Many, and Many-to-Many. Another is hierarchical, which I cover in chapter seven.
- The classes that EF Core maps to the database are referred to as *entity classes*. I use this term to highlight that the class I'm referring to is mapped by EF Core to database.
- If you load an *entity class*, by default, it won't load any of its relationships. For example, querying the `Book` entity class won't load its relationship properties, `Reviews`, `AuthorsLink`, and `Promotion`, but leave them as null.
- Here's three ways you can load related data that's attached to an *entity class*: *eager loading*, *explicit loading*, and *select loading*. It's likely that another technique called *lazy loading* will be added to EF Core at some point.
- EF Core has a feature called *Client vs. Server evaluation* which allows you to include commands that can't be converted to SQL commands in your database query. EF Core extracts these non-SQL commands and executes them after the database access has finished.
- I've used the term *query object* to refer to an encapsulated query, or section of a query. These *query objects* are often built as .NET extension methods which means they can easily be chained together, in a similar style to how LINQ is written.

For readers who are familiar with EF6.x, the extra takeaways are:

- Many of the concepts in this chapter are the same as in EF6.x, but in some cases, for instance *eager loading*, the EF Core commands have changed slightly, but often for the better.
- Some features in EF6.x, such as automatic Many-to-Many relationship setup and lazy loading are, at the time of writing this chapter, missing from EF Core. Alternatives exist, but it changes the way you use EF.
- EF Core's *Client vs. Server evaluation* feature's new and allows you to write queries that would've previously thrown an exception in EF6.x.



Entity Framework Core in Action teaches developers how to add database functionality to .NET applications with EF Core. Part 1 starts with a clear introduction to what EF Core is and how it fits into your applications. Next, you'll get hands-on quickly by building a .NET application that uses a relational database with EF Core's default configuration. By the end of part 1 you will be able to build a well-structured application that uses EF Core for database access.

The second part of the book dives deeper and shows you how to change default settings as well as teaching you many EF Core commands. You'll learn how to create a database exactly the way you want it, as well as how to link to an existing database, and how to change the way database data is exposed inside your .NET application.

The last part is all about improving your skills and making you a better developer and debugger of EF Core applications. You'll learn from real-world applications of EF Core starting with a range of known patterns and practices that you can use, extending ER Core, and how to find and fix EF Core performance issues. Software developers who have never used Entity Framework and seasoned EF6.x developers, as well as anyone who wants to know what EF Core is capable of will find this book great at deepening their knowledge and making them more productive with EF Core.

What's inside:

- Querying a database
- Creating, updating and deleting data
- Using EF Core in business logic
- Building a .NET web application EF Core

This book assumes readers are familiar with .NET development and some understanding of what relational databases are. No experience with SQL needed.

index

Symbols

.NET Core
  and writing blog posts 64
.NET web applications, layered architecture and 133
@CheckForNull annotation 17–18
@Nonnull annotation 17–18
@Nullable annotation 17–18
& symbol 77
#equals(Object) method 4
#getCacheExpirySeconds() method 4
#hashCode() method 4–5

Numerics

202 (Accepted) status code 82
404 status code, raw 108

A

A class 34
action method 94
  and threefold responsibility of 104
  defined 92, 103
  example 105
  method arguments 105
  MVC controllers and 103–109
  returning more than one type of result 106
  view engine and generating a response 104
action model, and invoking an appropriate action 93
action *See* action method
Add New Item dialog 99
Add-Migration, Migrate command 120
  and multiple projects used by an application 120
AddMvc call 100
anti-corruption layer 49
application model, executing an action using 93
armorStrength variable 11–12
ASP.NET Core
  and convention over configuration 102
  and different types of IActionResult 106
  and MVC implementation 91
  and writing HTTP REST services 83
  dependency injection 68
  empty template 97
  pluggable nature of 96
  securing services, reference literature 78
ASP.NET Core application
  and a request response 86
  middleware and 86
ASP.NET MVC 67
ASP.NET web service, creation of 66–69
ASP.NET, and built-in mechanisms to route requests 66
async/await constructs 72
authentication header 75
Azure blob container 75
  Markdown file in 76
Azure blob storage 65
  and creating the GetBlob method 73
  authorization header 78
  incorporating post storage in 72
  requests and 80
Azure emulator 73
Azure portal 76
Azure storage account 74

B

B class 34
BannerAdChooser class 22–25
BannerAdChooserImpl class 24
binding model 105
  building 92
  defined 92
blob storage *See* Azure blob storage
blob(s)
  deleting 82
  listing 80–81
bounded context 42
BoundedQueue class 14–15
Brolund, Daniel 10
business capabilities 40–47, 59
  identifying 42–43
  individually deployable 59
  overview 41–42
  point-of-sale system example 43–47
    identifying business capabilities in point-of-sale domain 44–46
    Special Offers microservice 46–47
  replaceable and maintainable by small team 59
  responsible for single capability 59
business logic, action methods and 104
business logic, complex, explicit loading in 125

C

C class 34
chaining 138
characteristics of microservices
  individually deployable 59–60
  maintainable by small team 59–60
  replaceable 59–60
  responsible for single capability 59–60
CharacterProfile object 26–27
CharacterProfileViewAdapter class 27
CLI *See* command line interface (CLI)
Client vs. Server evaluation 126, 139
  as a new feature in EF Core 127
  benefits 127
  example of using 127
  understanding the limitations 128–129
client, slow upload speed 71
code communication 56
collection operations 122
com.google.common.base.Optional class 17
command line Interface (CLI) 120
commented-out code 11
composite query, query objects for building 138
config file, an example 73
config.json file
  Azure emulator 73
config.json file
  adding to a project 72
configuration, getting values from 72–73
Configure method 98
ConfigureServices method 98, 100
connection string 119
console application, testing the new Azure storage operation 76
container(s)
  deleting a blob from 82
  listing 80–81
content header 78
content length 79
ContentResult, a type of IActionResult 106
context.Database.Migrate() method, startup code and 120
ContextFactoryNeededForMigrations class 121
controller 101–103
  action methods and 103
  and using a different view for model display 90
  as a MVC design pattern component 89
  as interaction entry point 89
  common conventions for defining 103
  data display and 90
  defined 92
  incoming request and 95
  instantiable 101
  location conventions 102
  the role of an action method in 93
Controller class 101
controller, creating 68–69
Controllers sub folder 102
convention over configuration 102
Convert method, rewriting to be asynchronous 71
Conway's law 42
coupling, reduction of 96
Coupons microservice 58
CreateRequest helper method 79
CreateRequest, reusing 75
csproj file 97
Curl command line tool
  and web service testing 69
  described 69
Curl command, testing the new Azure storage operation 77

D

data display, and advantage of using MVC design pattern 89
Data Transfer Object (DTO) 130
database
  by convention modelling of 117
  creating for an application 119–121

EF Core and accessing 112
EF Core and mapping .NET classes onto 117
example of complete 115
database diagram 115
database query, anatomy of *See* also query 121–122
database relationship
and default behavior of not loading 122
EF Core and modelling 112, 139
database round trips 124
explicit loading and 125
database, blog post storage 64
DataLayer 134
data-model duplication 54
DbContext
creating an instance of the application's 119
defining the application's 118
IDisposable interface 119
DbContext property access 121
DbSet property 121
dead code 11–12
default constructor 131
default web template project file, modified 66
default() keyword 79
dependency injection 119
dependency injection, IMarkdownEngine object added to 68
dependency, adding on the Configuration library 72
Design Patterns: Elements of Reusable Object-Oriented Software (Kerievsky) 29
development mode 119
development storage *See* Azure emulator
Django, MVC design pattern and 88
DoBy Scala library 13
domain model, defined 93
domain-driven design 41
dotnet new console command, converting Markdown to HTML 65
dotnet restore, converting Markdown to HTML 65
dotnet run, and running a project from the command line 99
drivers 58–59
DTO *See* Data Transfer Object (DTO)

E

eager loading
and loading relationships with the primary entity class 122
benefits 124
described 123
downside 124
example of a complex 123
EF Core database model 117
EF Core, database querying and 112
EF6.x user, Many-to-Many relationship and 114
EfCoreContext, defining the application's DbContext 118
EfCoreInAction 134
Ellnestam, Ola 10
else block 11–12
Empty Project template 97
EnsureCreated command, database creation and 119
entity class 117
and simple example application 133
as database structure model 119
entry point, ASP.NET Core 103
enum parameter 135
ERP (Enterprise Resource Planning) system 48, 50
execute command 122
execute() method 9
expired code 13
explicit loading
and loading relationships after the primary entity class 124
benefits 125
downside 125
example of 124
Load() command and 122
exposure, level of, Azure blob containers and 75
extension method 132, 134
external product catalog system, integrating with 48–50

F

Feathers, Michael 33
FileResult, a type of IActionResult 106
filtering 135–137
final class 4
FindBugs 12
first level relationship 124
Fixme Scala library 13
FK abbreviation 115
fluent interface 121
FooView Pattern 27
foreach statement, execute command and 122
foreign keys, used as primary keys 115
Fowler, Martin 29

G

Gamma, Erich 29
generic paging query object 137
GET method, Azure blob storage and 72
GET request 82
  GetBlob method 74
GetAuthHeader method 75
GetBlob method
  creation of 73–76
  updated 76
GetFilterDropDownValues method 136
gradual rollouts 36
GUI environment, MVC design pattern and 88

H

Halladay, Steve 29
Helm, Richard 29
helper methods
  and creating ActionResults 108–109
  returning results 101
hierarchical relationship, EF Core and 115
Hoare, Tony 16
HTML view, model display and 90
HTML web application 86
HTTP calls, making 69–71
HTTP PUT operation, idempotency of 78
HTTP REST service, .NET Core version and 74
HTTP REST services 67
HttpClient vs. WebClient 70
HttpGet attribute, GetBlob method 74
HttpGet operation, listing containers and blobs 80
HttpPost method 69
HttpRequestMessage object 74

I

IActionResult 106–109
ID/key-generating methods 4
IDE
  refactoring and 5–9
if statement 11
ILSpy tool, Visual Studio 72
IMarkdownEngine object 68
Include() method, eager loading and 122
Index method 106
IntelliJ IDEA 6
IQueryable method 131
IQueryable query 137
isStable flag 4

J

java.util.Optional class 17
Johnson, Ralph 29
JSON, as a machine-readable format 87

K

Kerievsky, Joshua 29

L

layered architecture approach 133
  diagram of 133
lazy loading 126
linking table 114
LINQ
  execute command and 122
  online resources for 121
LINQ/EF Core command 121

M

Macbeth Syndrome 3
maintainable applications, MVC design pattern as a model for building 87
Many-to-Many relationship 113–114
  automatic 139
Mappers 132
mapping capabilities 41
Markdown
  and writing blog posts 64
  availability in .NET Core or .NET Standard 65
Markdown file, content of 78
Markdown service, GET operation 76
Markdown text, conversion into HTML, sample code 65–66
MarkdownLite library 65
  and and tags 66
method signature, GetBlob method 74
microservice
  and building high performance microservice applications 71
  and its own isolated data source 77, 83
  described 65
  individually deployable 65
  replaceability 65
Microservices in .NET Core 65
Microsoft.AspNetCore.Mvc package 101
Microsoft.AspNetCore.Mvc package reference 98
Microsoft.EntityFrameworkCore.Tools 120
Microsoft.Extensions.Configuration library 72

middleware pipeline 91
  small 97
Migrate commands 119
migration feature 120
Mikado Method 10
model
  as a MVC design pattern component 89
  view independent 90
Model View Controller *See* MVC
ModelState property 105
Model-View-Controller (MVC) design pattern *See* MVC design pattern
Model-View-Controller pattern 25
mutable state 18–20
MVC 68
  and Web API 67
  HttpGet attribute 74
MVC application
  responsibilities of each component of 89
  the order of events in responding to a request 89
  traditional, action methods for 104
MVC design pattern 88–90
  and handling different aspects of a single page request 88
  and traditional description 91
  different interpretations of the original 88
  terminology issues 91
MVC functionality, accessing 100
MVC middleware internals
  customization of 100–101
  replacing parts of 100
MVC options, configurations of 100
MVC request, complete 94–96
MvcMiddleware 91
  adding to an application 96–101
  and handling UI code for an application 87
  view engine in 104
Mvcmiddleware
  and generating web pages 87
  as a final piece of middleware in the pipeline 86, 91

N

new syntax, ActionResults and 108
nil object 18
non-server runnable commands, EF Core and 127
NotFoundResult, a type of IActionResult 106, 108
Notifications microservice 51
notifications, sending to customers 50–52
null blob parameter 80
null container parameter 80
null references 16–18
NullPointerException 16–17

O

OnConfiguring method 119
One-to-Many relationship 113
  example of 114
  hierarchical relationship and 115
One-to-One relationship 113
  hierarchical relationship and 115
One-to-ZeroOrOne relationship 113
OnModelCreating method 118
Option type 16–17
OrderBy method 134
OrderByDescending, method 134

P

paging 137
parameter-less constructor 121
parameters, action methods and 104–106
PK abbreviation 115
Plain Old CLR Objects 117
Player object 31
PlayerUpdater 32
POCO *See* Plain Old CLR Objects
point-of-sale system example 43–47
  identifying business capabilities in point-of-sale domain 44–46
  Special Offers microservice 46–47
POST vs. HTTP PUT 78
POST, REDIRECT, GET flow 107
PostAsync method 71
presentation model 26
Principle-Based Refactoring 29
Product Catalog microservice 44
Program.cs 67
project
  adding a new class to 99
  restoring and running 99
public API 58
public constructor 32
public method *See* action method
PUT operation 77
PutBlob operation
  C# client code and testing 80
  Curl command and testing 79
PutBlob method 78

Q

query object 132, 134
  encapsulated query 139
query, building complex *See* also database query 129–134
Query() method, explicit loading and 124
querying 112

R

Rails, MVC design pattern and 88
readonly keyword 20
Redirect helper method, and model validity 106
RedirectResult, a type of IActionResult 107
RedirectResult, action methods and returning a response 104
RedirectToRouteResult, a type of IActionResult 106
refactoring 2–10
  common legacy code traits and 10–29
    business logic 20–25
    complexity in view layer 25–29
    needlessly mutable state 18–20
    null references 16–18
    stale code 11–13
    toxic tests 14–16
  disciplined
    avoiding Macbeth Syndrome 3
    IDE and 5–9
    Mikado Method 10
    separate refactoring from other work 3–5
    VCS (version control system) and 9
  testing legacy code 29–36
    regression testing without unit tests 33–36
    untestable code 29–33
    using user data 36
Refactoring to Patterns (Kerievsky) 29
Refactoring: Improving the Design of Existing Code (Fowler et. al) 29
related data
  different approaches to loading 112
  loading 122
  three ways of loading 122–126
relational database schema, example of complete 115–116
relational database, splitting out any repeated data and 113
request
  creating authentication header in 75
  handling more requests with fewer threads 72
  HttpClient and making 70–71, 83
request header, hashed version of 76
request, routing to an appropriate controller 92
response, action method and appropriate kind of 104
Rule class 22–23
RuneQuest 11
runtime, exceptions at 100

S

scoping microservices
  business capabilities 41–47
    identifying 42–43
    overview 41–42
    point-of-sale system example 43–47
  microservice characteristics and 58–60
    primarily scoping to business capabilities leads to good microservices 59
    secondarily scoping to supporting technical capabilities leads to good microservices 59–60
  technical capabilities 48–52
    identifying 52
    overview 48
    supporting, examples of 48–52
  unclear scope, moving forward despite 52–58
    carving out new microservices from existing 56–57
    planning to carve out new microservices later 58
    starting bigger 53–56
Search action method 105
SearchModel object 105
second level relationship 124
secure resource, production service 78
select loading 125–126, 129–133
  and combining individual queries 129
Select method, example of use 125
separation of concerns 95, 133
service code, responsibility of the layers beneath 72
ServiceLayer 134
sorting 134–135
Special Offers microservice 44, 46–47, 53, 58
Spell object 19–20
SpellWithUsageCount class 19–20
Spring MVC, MVC design pattern and 88
SQL commands, loading related data 123
SQL IDENTITY command 117
stale code 11–13
  commented-out code 11
  dead code 11–12
  expired code 13
  zombie code 12–13

standard web application, model display and 90
Startup class 67
Startup.cs file 98
Startup.cs file, modified 67
StatusCodeResult, a type of IActionResult 106, 108
stealth releases of versions 36
StreamContent object, making HTTP calls and 71

T

table-per-hierarchy (TPH) configuration 115
Task objects, async methods and 71
technical capabilities 40, 48–52, 59–60
  identifying 52
  individually deployable 60
  overview 48
  replaceable and maintainable by small team 60
  responsible for single capability 60
  supporting, examples of 48–52
    integrating with external product catalog system 48–50
    sending notifications to customers 50–52
test.md file, and making HTTP calls 69
testability, view independent model and 90
testing
  legacy code 29–36
    regression testing without unit tests 33–36
    untestable code 29–33
    using user data 36
ThenInclude() method, eager loading and 122–123
this, keyword, extension method and 132
thread, problem with blocking 71
tight coupling 96
timesUsed field 19
TPH configuration *See* table-per-hierarchy (TPH) configuration
Tweet code 8
TweetBuilder class 8

U

UNIQUE index 113
Update-Database, Migrate command 120
updatePlayer() method 32
uploaded data, receiving 77–79
usage-tracking process 46
UseMvc extension method 98
user data, using with legacy code testing 36
using statement 119
Util class 30–31, 33, 37
Util.updatePlayer method 31

V

VCS (version control system) 9
view
  as a final data representation 89
  as a MVC design pattern component 89
  generating of the final representation and 90
  model independent of 90
view layer, complexity in 25–29
view model
  defined 94
  generating a response using 94
ViewModel 26
ViewModel, ASP.NET 130
ViewResult, a type of IActionResult 107
ViewResult, action methods and returning a response 104
Visual Studio Package Manager Console (PMC) 120
Vlissides, John 29

W

Web API, building HTTP REST services and 67
web application, MarkdownLite library and example of creating 65
web server, Program.cs and starting 67
web service, testing with Curl 69
WebClient 70
Working Effectively with Legacy Code (Feathers) 33

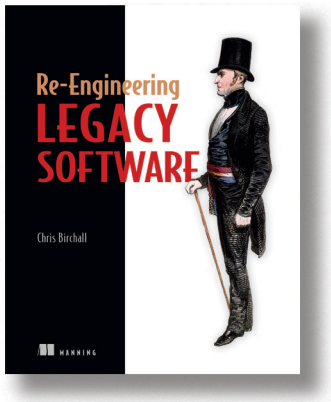
X

XML, as a machine-readable format 87

Z

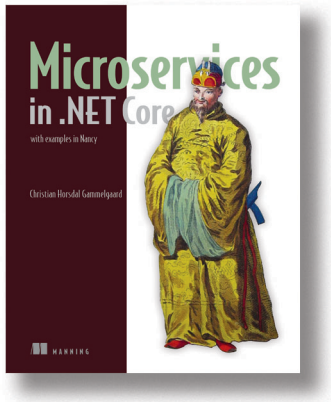
zombie code 12–13

Save 50% on these selected books—eBook, pBook, and MEAP. Just enter ***feecore50*** *in the Promotional Code box when you check out. Only at manning.com.*



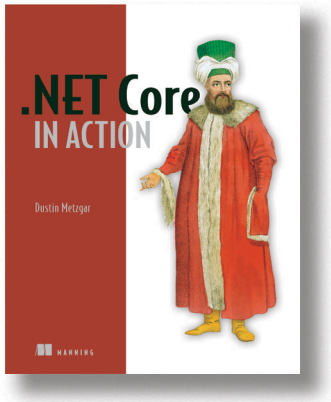
Re-Engineering Legacy Software
by Chris Birchall

ISBN: 9781617292507
232 pages
$64.99
April 2016



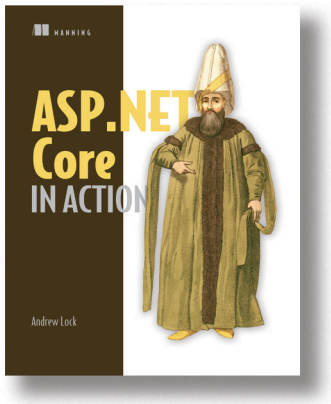
Microservices in .NET Core with examples in Nancy
by Christian Horsdal Gammelgaard

ISBN: 9781617293375
344 pages
$49.99
January 2017



.NET Core in Action
by Dustin Metzgar

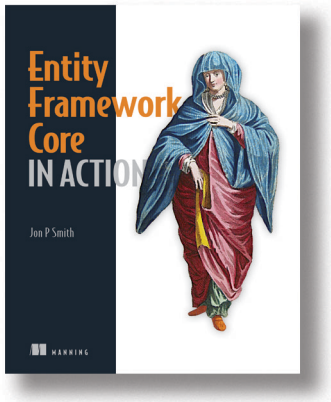
ISBN: 9781617294273
350 pages
$44.99
Early 2018



ASP.NET Core in Action

by Andrew Lock

ISBN: 9781617294617
500 pages
$49.99
Spring 2018



Entity Framework Core in Action

by Jon Smith

ISBN: 9781617294563
500 pages
$49.99
Spring 2018