John Callaway, Clayton Hunt

Foreword by:
John Sonmez
Founder, *Simple Programmer*

Practical Test-Driven Development using C# 7

Unleash the power of TDD by implementing real world examples under .NET environment and JavaScript

Packt>

Practical Test-Driven Development using C# 7

Unleash the power of TDD by implementing real world examples under .NET environment and JavaScript

John Callaway
Clayton Hunt

BIRMINGHAM - MUMBAI

Practical Test-Driven Development using C# 7

Mapt

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals

- Improve your learning with Skill Plans built especially for you

- Get a free eBook or video every month

- Mapt is fully searchable

- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Foreword

I'll be honest, when I first looked at the title of this book, *Practical Test-Driven Development Using C# 7*, I thought "do we really need another book outlining the basics of unit testing and TDD?" I mean at this point, TDD isn't really that new, and plenty of authors have written about what TDD is and have given us countless examples of how to score a bowling or a tic-tac-toe game using TDD to drive the creation of the application.

The problems most developers run into when learning or trying to implement TDD is not understanding what TDD itself is and what "red, green, refactor" means, but how to practically implement it in real-world situations. Most real-world applications are messy and don't conform easily to the usual TDD shoehorn that many well-meaning TDD books and examples try to demonstrate. The real difficulty in successfully implementing TDD arises when you try to use it to develop a non-trivial application—an application with databases and multiple layers and external services you need to call.

As a consultant, I spent a large amount of time teaching teams who were supposedly doing TDD how to actually do TDD. I saw countless examples of teams who would hardcode tests with fake data or call out directly to a database, because they didn't really understand what a mock was and how to isolate their tests. Moreover, I found many development teams that understood the basics of TDD, but didn't understand how to take business requirements and user stories and convert them into working unit tests that they can actually use to develop the system they were trying to create.

I've never had a difficult time teaching software developers the basics of TDD. It's fairly easy to explain how TDD works and how to get started doing it—and there have always been plenty of resources available to teach all that. No, what companies paid me the big bucks for was to sit down with their teams and explain all the nuances of TDD. What do you do when you have to mock a class that directly uses the database? How do you handle 15 test cases that have different input values but are essentially testing the same thing? Where do we start with TDD—which tests do we write first?

That's where Practical Test-Driven Development comes into the picture. For the first time ever, all that "real tricky shit," that I thought only I knew about and was esoteric and complicated to explain in a book, well it's explained—with plenty of examples—right here in the one you are holding.

When I first cracked open Practical Test-Driven Development, I was amazed. Not only did it explain the right concept of what TDD was (a design activity, not a testing one), and give an absolute beginner a step-by-step approach to learning and understanding TDD, but it took it far, far from there and showed you how to practically and pragmatically apply the concepts to a real-world, non-trivial application and did it in a way that didn't try and gloss over the messy stuff. Instead, Practical Test-Driven Development plows right into all the nooks and crannies of TDD and tells you exactly the kinds of problem you will encounter and how to get past them.

Here's the best part—you don't even have to know a thing about TDD to get huge value out of this book and become a better TDD practitioner than 90% of software developers who claim they are already doing TDD. Practical Test-Driven Development is laid out in such a way that it assumes that you don't know anything about TDD and shows you step-by-step, example-by-example, everything you need to know to go from complete beginner to expert in a way that I honestly didn't think was possible, until I sat down and read the book myself.

All in all, I am extremely happy with this book and I have a feeling I'm going to be recommending it as the go-to resources for learning TDD for many years to come. Every developer should be doing TDD and doing it the right way. Therefore, every developer should read this book.

John Sonmez
Founder, Simple Programmer

Contributors

About the authors

John Callaway, a Microsoft MVP, has been a professional developer since 1999. He has focused primarily on web technologies and has experience with everything from PHP to C# to ReactJS to SignalR. Clean code and professionalism are particularly important to him, along with mentoring and teaching others what he has learned along the way.

Clayton Hunt has been programming professionally since 2005, doing mostly web development with an emphasis on JavaScript and C#. He has a focus on Software Craftsmanship and is a signatory of both the Agile Manifesto and the Software Craftsmanship manifesto. He believes that through short iterations and the careful gathering of requirements, we can deliver the highest quality and most value in the shortest time. He enjoys learning and encouraging others to continuously improve themselves.

About the reviewer

Tomi Juhola is a versatile software development professional from Finland. He has wide experience, from embedded systems through distributed enterprise systems to a reinvention of IT in manufacturing in various roles. The key thing for him is to be agile and to help others grow.

Currently, he works in an industry-leading manufacturing company and is responsible for all things software and electric. He likes to spend his free time with new interesting development languages, technologies, and frameworks as well as with novel thoughts on organizations and personal growth.

I would like to thank my dear wife, Jonna, and daughter, Isla, for letting daddy reserve some time for reviewing. Love you both.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

As software projects grow in size and complexity, it can often become more difficult, time-consuming, and expensive to maintain them. Through Test Driven Development (TDD), you can learn to develop testable, extensible, and maintainable software applications.

Who this book is for

This book is for software developers who have cursory knowledge of TDD and are looking to gain a thorough understanding of how TDD can benefit them and the applications they produce. Software developers with an intermediate understanding of C# and the .NET Framework and/or a thorough understanding of JavaScript and React will likely be able to follow along with all the code examples used throughout the book.

What this book covers

The book covers everything from why TDD is important to setting up testing environments, and how to get started testing a green-field application. As the reader grows more comfortable, they will be exposed to more advanced TDD topics such as abstracting away third-party code, approaching a problem from a TDD perspective, and how to deal with legacy code that wasn't written with testability in mind.

Chapter 1, *Why TDD is Important*, asks what is TDD and why should you care? In this chapter, you will learn what TDD is and why it matters. A compelling argument for TDD will be made and the benefits, and more importantly, the execution will be shown.

Chapter 2, *Setting Up the .NET Test Environment*, explains how to set up your IDE and configure the testing framework so that you can easily run your tests in C# and .NET, with more detail and many more examples of growing complexity in the Speaker Meet API.

Chapter 3, *Setting Up a JavaScript Environment*, configures the JavaScript testing framework so that you can easily run your tests in your IDE. It provides more detail and many more examples of growing complexity in the Speaker Meet React application.

Chapter 4, *What to Know Before Getting Started*, dives deeper into the why and how of TDD. you will learn the importance of defining and testing boundaries and abstracting away third-party code (including the .NET Framework), and you'll discover more advanced concepts such as spies, mocks, and fakes, and how to avoid pitfalls along the way.

Chapter 5, *Tabula Rasa - Approaching an Application with TDD in Mind*, explains how to get started with a new application. You'll apply what you've learned in the previous chapters and take the same approach with a full-sized application using Speaker Meet as an example.

Chapter 6, *Approaching the Problem*, takes the broader problem of the overall application and breaks it into meaningful chunks that can be developed independently. You'll learn different approaches to developing an application, such as front to back, back to front, and inside out.

Chapter 7, *Test-Driving C# Applications*, takes requirements and assembled user stories and turns them into working software using TDD. It explains how to utilize all the skills you've assembled so far to test the boundaries, testing small, individual units.

Chapter 8, *Abstract Away Problems*, explores abstracting away third-party libraries, including the .NET Framework. It covers removing dependencies on things such as DateTime and Entity Framework. It explains how to decouple their application from specific implementations to not only allow your application to be testable but much more flexible and easy to modify in the future.

Chapter 9, *Testing JavaScript Applications*, now that you have a working API, focuses on creating a Single Page Application in JavaScript using React. It focuses on test-driven actions and

reducers and any functionality within the application.

Chapter 10, *Exploring Integrations*, explains how to write integration tests to ensure that your application is functioning properly.

Chapter 11, *Changes in Requirements*, focuses on what happens when the requirements change. What happens if a bug is discovered? No problem, change a test or write a new one to cover the new requirement or to defend against the discovered bug. Now, write some new code or change some existing code to make all of the new/modified tests pass. If you do everything correctly, you should feel safe to make these changes as your existing test suite will prevent you from introducing new bugs.

Chapter 12, *The Legacy Problem*, explains that there are a lot of applications out there without sufficient (any?) test coverage, and even fewer were written test-first. You'll discover some of the major problems with legacy applications that weren't written with testability in mind; they will be identified, and also how best to recover will be covered.

Chapter 13, *Unraveling a Mess*, dives into how to go about safely modifying a legacy application that wasn't written with testing in mind. How can you add tests to minimize the potential for introducing new bugs when modifying the existing code? An extreme example will be used to explore these topics and more.

Chapter 14, *A Better Foot Forward*, emphasizes that TDD is a personal choice. You don't need anyone's permission to do good work. Advice on how to continue a successful journey of TDD, how to introduce TDD to your team, and how to rejoin the world as a TDD expert will be covered in this chapter.

To get the most out of this book

Readers wanting to follow along with the examples in the book should have the following:

- An intermediate understanding of C# and/or JavaScript
- Prior exposure to React will be beneficial though not required
- Familiarity with N-tier architecture

Download the example code files

You can download the example code files for this book from your account at www.packtpub.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Practical-Test-Driven-Development. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: http://www.packtpub.com/sites/default/files/downloads/PracticalTestDrivenDevelopment_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
"babel": {
    "presets": [
      "react-app"
    ]
 },
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

Any command-line input or output is written as follows:

```
>npm install mocha chai sinon enzyme
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select System info from the Administration panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

Why TDD is Important

You've picked up this book because you want to learn more about **Test-Driven Development** (**TDD**). Maybe you've heard the term before. Perhaps you've known software developers who write unit tests and want to learn more. We'll introduce you to the terms, the structure, and the ideology around TDD. By the end of this book, you'll have sufficient knowledge to re-enter the world as a Test-Driven Developer and feel confident about using your skills throughout your long and prosperous career.

Why this book? Certainly, there are many other books on the topic of TDD. We have written this book with the hope that it provides you, the reader, with low-level insight into the mindset we use when doing TDD. We also hope that this book provides an updated view of some of the concepts and lessons we have learned while doing TDD over the last 10 years.

So, why is TDD so important? As more businesses and industries rely on software solutions, it's increasingly important that those solutions be robust and error-free. The cheaper and more consistent, they are the better. Applications developed with TDD in mind are inherently more testable, easier to maintain, and demonstrate a certain level of correctness not easily achieved otherwise.

In this chapter, we will gain an understanding of:

- Defining TDD and exploring the basics
- Creating our first tests in C# and JavaScript
- Exploring the basic steps of Red, Green, Refactor
- Growing complexity through tests

First, a little background

It's possible that you've had some exposure to unit tests in your career. It's highly likely that you've written a test or two. Many developers, unfortunately, haven't had the opportunity to experience the joys of Test-Driven Development.

John's story on TDD

I was first introduced to TDD about five years ago. I was interviewing for a lead developer position for a small startup. During the interview process, the CTO mentioned that the development team was practicing TDD. I informed him that I didn't have any practical TDD experience, but that I was sure I could adapt.

In all honesty, I was bit nervous. Up to that point, I had never even written a single unit test! What had I gotten myself into? An offer was extended and I accepted. Once I joined the small company I was told that, while TDD was the goal, they weren't quite there yet. Phew; crisis averted. However, I was still intrigued. It wasn't until a few months later that the team delved into the world of TDD, and the rest, as they say, is history.

Clayton's story on TDD

My introduction to TDD is a little different from John's. I have been writing code since I was in middle school in the early 1990s. From then until 2010, I always struggled with writing applications that didn't require serious architectural changes when new requirements were introduced. In 2010, I finally got fed up with the constant rewrites and began researching tools and techniques to help me with my problem. I quickly found TekPub, an e-learning site that was, at the time, owned and operated by Rob Conery. Through TekPub I began learning the SOLID principles and TDD. After banging my head against the wall for close to six months, I started to grasp what TDD was and how I could use those principles. Coupled with the SOLID principles, TDD helped me to write easy to understand code that was flexible enough to stand up to any requirements the business could throw at me. I eventually ended up at the same company where John was employed and worked with him and, as he said, the rest is history.

The SOLID principles, which will be explained in detail later, are guiding principles that help produce clean, maintainable, and flexible code. They help reduce rigidity, fragility, and complexity. Generally thought of as object-oriented principles, I have found them to be applicable in all coding paradigms.

So, what is TDD?

Searching online, you will certainly find that TDD is an acronym for Test-Driven Development. In fact, the title of this book will tell you that. We, however, use a slightly more meaningful definition. So, what *is* TDD? In the simplest terms, TDD is an approach to software development that is intended to reduce errors and enable flexibility within the application. If done correctly, TDD is a building block for rapid, accurate, and fearless application development.

Test-Driven Development is a means of letting your tests drive the design of the system. What does that mean, exactly? It means that you mustn't start with a solution in mind, you must let your tests drive the code being written. This helps minimize needless complexity and avoid over-architected solutions. The rules of Test-Driven Development

Staunch proponents of TDD dictate that you may not write a single line of production code without writing a failing unit test, and failing to compile is a failure. This means that you write a simple test, watch it fail, then write some code to make it pass. The system slowly evolves as the tests and the production application grow in functionality.

> *TDD is not about testing, it's about design.*

Many would argue that TDD is about testing, and by extension, about test coverage of an application. While these are great side-effects of TDD, they are not the driving force behind the practice.

Additionally, if code coverage and metrics become the goal, then there is a risk that developers will introduce meaningless tests just to inflate the numbers. Perhaps it is less a risk and more a guarantee that this will happen. Let delivered functionality and happy customers be the metrics with which you measure success.

TDD is about design. Through TDD, an application will grow in functionality without introducing needless complexity. It's incredibly difficult to introduce complexity if you write small tests and only enough production code to make the test pass. Refactoring, modifying the structure of the code without adding or changing behavior, should not introduce complexity, either.

An approach to TDD

TDD is also referred to as Test First Development. In both names, the key aspect is that the test must be written before the application code. Robert C. Martin, affectionately called "Uncle Bob" by the developer community, has created *The Three Laws of TDD*. They are as follows:

1. You are not allowed to write any production code unless it is to make a failing unit test pass
2. You are not allowed to write any more of a unit test than is sufficient to fail, and compilation failures are failures
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test

You can learn more about these laws at http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

By following these rules, you will ensure that you have a very tight feedback loop between your test code and your production code. One of the main components of Agile software development is working to reduce the feedback cycle. A small feedback cycle allows the project to make a course correction at the first sign of trouble. The same applies to the testing feedback cycle. The smaller you can make your tests, the better the end result will be.

> *For a video on Agile, check out Getting Started with Agile by Martin Esposito and Massimo Fascinari (https://www.packtpub.com/application-development/getting-started -agile-video).*

An alternative approach

The original approach to TDD has caused some confusion over the years. The problem is that the principles and approaches just weren't structured enough. In 2006, Dan North wrote an article in Better Software magazine (https://www.stickyminds.com/better-software-magazine/behavior-modification). The purpose of the article was to clear up some of this confusion and help to reduce the pitfalls that developers fell into while learning the TDD process. This new approach to TDD is called **Behavior Driven Development** (**BDD**). BDD provides a structure for testing, and a means of communicating between business requirements and unit tests, that is almost seamless.

The process

It's difficult to start any journey without a goal in mind. There are a few tips and tricks that can be used to help get you started in TDD. The first is *red, green, refactor*.

Red, green, and refactor

We already discussed writing a failing test before writing production code. The goal is to build the system slowly through a series of tiny improvements. This is often referred to as *red, green, refactor*. We write a small test (red), then we make it pass by writing some production code (green), then we refactor our code (refactor) before we start the process again.

Many TDD practitioners advocate an *It Exists* test first. This will help determine that your environment is set up properly and you won't receive false positives. If you write an *It Exists* test and don't receive a failure right off the bat, you know something is wrong. Once you receive your first failure, you're safe to create the class, method, or function under test. This will also ensure that you don't dive in too deeply right off the bat with lines and lines of code before you're sure your system is working properly.

Once you have your first failure and the first working example, it's time to grow the application, slowly. Choose the next most interesting step and write a failing test to cover this step.

At each iteration, you should pause and evaluate whether there is any cleanup that can happen. Can you simplify a code block? Perhaps a more descriptive variable name is in order? Can any sins committed in the code be corrected, safely, at this time? It's important that you evaluate both the production code and the test suite. Both should be clean, accurate, and maintainable. After all, if it's such a mess that no one would be able to make head or tail of it, what good is the code?

Coder's block

TDD will also help you avoid what writers often call *writer's block* and what we're calling *coder's block*. Coder's block happens when you sit down at the keyboard in an attempt to solve a problem but <mark>don't know where to begin</mark>. We begin at the beginning. Write the easiest, simplest test you can imagine. <mark>Write *It Exists*</mark>.

Why should we care?

We're professionals. We want to do a good job. We feel bad if someone finds fault with our code. If QA finds a bug, it makes us sad. If a user of our system encounters an error, we may cry. We should strive to deliver quality, error-free code and a fully functional, feature-rich application.

We're also lazy, but it's the good kind of lazy. We don't want to have to run the entire application just to validate that a simple function returns the proper value.

Arguments against TDD

There are arguments against TDD, some valid and some not. It's quite possible that you've heard some of them before, and likely that you've repeated some of these yourself.

Testing takes time

Of course, testing takes time. Writing unit tests takes time. Adhering to the *red, green, refactor* cycle of TDD does take time. But, how else do you check your work if not through tests?

Do you validate that the code you wrote works? How do you do this without tests? Do you manually run the application? How long does that take? Are there conditional scenarios that you need to account for within the application? Do you have to set up those scenarios while manually testing the application? Do you skip some and just *trust that they work*?

What about regression testing? What if you make a change a day, a week, or a month later? Do you have to manually regression-test the entire application? What if someone else makes a change? Do you trust that they were also as thorough in their testing, *as I'm sure you are*?

How much time would you save if your code were covered by a test suite that you could run at the click of a button?

Testing is expensive

By writing tests, you're effectively doubling the amount of code you're writing, right? Well, yes and no. Okay, in an extreme case, you might approach double the code. Again, *in an extreme case*.

 ==*Don't make tests a line item.*==

In some instances, consulting companies have written unit tests into a contract with a line item and dollar amount attached. Inevitably, this allows the customer the chance to argue to have this line item removed, thus saving them money. This is absolutely the wrong approach. Testing will be done, period, whether manually by the developer running the application to validate her work, by a QA tester, or by an automated suite of tests. Testing is not a line item that can be negotiated or removed (yikes!).

You would never buy an automobile that didn't pass quality control. Light bulbs must pass inspection. A client, customer, or company will never, ever, save money by foregoing testing. The question becomes, do you write the tests early, while the code is being authored, or manually, at a later date?

Testing is difficult

Testing can be difficult. This is especially true with an application that was not written with testability in mind. If you have static methods and implementations using concrete references scattered throughout your code, you will have difficulty adding tests at a later date.

We don't know how

I don't know how to test is really the only acceptable answer, assuming it is quickly followed by, *but I'm willing to learn*. We're developers. We're the experts in the room. We're paid to know the answers. It's scary to admit that we don't know something. It's even scarier to start something new. Rest assured, it will be OK. Once you get the hang of TDD, you'll wonder how you managed before. You'll refer to those times as *the ==dark ages, before the discovery of the wheel==*.

Arguments in favor of TDD

What we would like to focus on here are the positives, the arguments in favor of TDD.

Reduces the effort of manual testing

We already mentioned that we, as professionals, will not ship anything without first determining that it works. Throwing something over the wall to QA, to our users, or to the general public and hoping that it all works as expected just isn't how we do business. We will verify that our code and our applications work as expected. In the beginning, while the application is small and has little functionality, we can manually test everything we can think of. But, as the application grows in size and complexity, it just isn't feasible for developers or anyone else to manually test an entire application. It's too time-consuming and costly to do this manually. We can save ourselves time and our clients and companies money by automating our testing. We can do so quite easily, from the beginning, through TDD.

Reduces bug count

As our application grows, so do our tests. Or shall we say, our test suite has grown, and by making our tests pass, our application has grown. As both have grown, we've covered the happy path (for example: 2 + 2 = 4) as well as potential failures (for example: 2 + banana = exception). If the method or function under test can accept an input parameter, there is a potential for failure. You can reduce the potential for unexpected behavior, bugs, and exceptions by writing code to guard against these scenarios. As you write tests to express potential failures, your production code will inherently become more robust and less prone to errors. If a bug does slip by and make it to QA, or even to a production environment, then it's easy enough to add a new test to cover the newly discovered defect.

The added benefit of approaching bugs in this fashion is that the same bug rarely crops up again at some later date, as the new tests guard against this. If the same bug does appear, you know that, while the same result has happened, the bug occurred in a new and different way. With the addition of another test to cover this new scenario, this will likely be the last time you see the same old bug.

Ensures some level of correctness

With a comprehensive suite of tests, you can demonstrate some level of correctness. At some point, someone somewhere will ask you whether you are done. How will you show that you have added the desired functionality to an application?

Removes the fear of refactoring

Let's face it, we've all worked on legacy applications that we were scared to touch. Imagine if the class you were tasked with modifying were covered by a comprehensive set of unit tests. Picture how easy it would be to make a change and know that all was right with the world because all of the unit tests still passed.

A better architecture

Writing unit tests tends to push your code towards a decoupled design. Tightly coupled code quickly becomes burdensome to test, and so, to make one's life easier, a Test-Driven Developer will begin to decouple the code. Decoupled code is easier to swap in and out, which means that, instead of modifying a tangled knot of production code, often all that a developer needs to do to make the necessary changes is swap out a subcomponent with a new module of code.

Faster development

It may not feel like it at first (in fact, it definitely will not feel like it at first), but writing unit tests is an excellent way to speed up development. Traditionally, a developer receives requirements from the business, sits down, and begins shooting lightning from her fingertips, allowing the code to pour out until an executable application has been written. Before TDD, a developer would write code for a few minutes and then launch the application so that she could see if the code worked or not. When a mistake was found, the developer would fix it and launch the application once again to check whether the fix worked. Often, a developer would find that her fix had broken something else and would then have to chase down what she had broken and write another fix. The process described is likely one that you and every other developer in the world are familiar with. Imagine how much time you have lost fixing bugs that you found while doing developer testing. This does not even include the bugs found by QA or in production by the customer.

Now, let's picture another scenario. After learning TDD, when we receive requirements from the business, we quickly convert those requirements directly into tests. As each test passes we know that, as per the requirements, our code does exactly what has been asked of it. We might discover some edge cases along the way and create tests to ensure the code has the correct behavior for each one. It would be rare to discover that a test is failing after having made it pass. But, when we do cause a test to fail, we can quickly fix it by using the undo command in our editor. This allows us to hardly even run the application until we are ready to submit our changes to QA and the business. Still, we try to verify that the application behaves as required before submitting, but now we don't do this manually, every few minutes. Instead, let your unit tests verify your code each time you save a file.

Different types of test

Over the course of this book, we will be leaning towards a particular style of testing, but it is important to understand the terminology that others will use so that you can relate when they speak about a certain type of test.

Unit tests

Let's jump right in with the most misused and least understood test type. In Kent Beck's book, *Test-Driven Development by Example*, he defines a unit test as simply a test that runs in isolation from the other tests. All that means is that for a test to be a unit test, all that has to happen is that the test must not be affected by the side-effects of the other tests. Some common misconceptions are that a unit test must not hit the database, or that it must not use code outside the method or function being tested. These simply aren't true. We tend to draw the line in our testing at third-party interactions. Any time that your tests will be accessing code that is outside the application you are writing, you should abstract that interaction. We do this for maximum flexibility in the design of the test, not because it wouldn't be a unit test. It is the opinion of some that unit tests are the only tests that should ever be written. This is based on the original definition, and not on the common usage of the term.

Acceptance tests

Tests that are directly affected by business requirements, such as those suggested in BDD, are generally referred to as acceptance tests. These tests are at the outermost limit of the application and exercise a large swathe of your code. To reduce the coupling of tests and production code, you could write this style of test almost exclusively. Our opinion is, if a result cannot be observed outside the application, then it is not valuable as a test.

Integration tests

Integration tests are those that integrate with an external system. For instance, a test that interacts with a database would be considered an integration test. The external system doesn't have to be a third-party product; however, sometimes, the external system is just an imported library that was developed independently from the application you are working on but is still considered in-house software. Another example that most don't consider is interactions with the system or language framework. You could consider any test that uses the functions of C#'s `DateTime` object to be an integration test.

End to end tests

These tests validate the entire configuration and usage of your application. Starting from the user interface, an end to end test will programmatically click a button or fill out a form. The UI will call into the business logic of the application, executing all the way down to the data source for the application. These tests serve the purpose of ensuring that all external systems are configured and operating correctly.

Quantity of each test type

Many developers ask the question: How many of each type of test should be used? Every test should be a unit test, as per Kent Beck's definition. We will cover variations on testing later that will have some impact on specific quantities of each type; but, generally, you might expect an application to have very few end to end tests, slightly more integration tests, and to consist mostly of acceptance tests.

Parts of a unit test

The simplest way to get started and ensure that you have human-readable code is to structure your tests using *Arrange, Act,* and *Assert*.

Arrange

Also known as the ==context of a unit test==, *Arrange* includes anything that exists as a ==prerequisite== of the test. This includes everything from parameter values, stored in variables to improve readability, all the way to configuring values in a mock database to be injected into your application when the test is run.

> *For more information on Mocking, see* Chapter 3, *Setting Up the JavaScript Environment, the Abstract Third Party Software and Test Double Types sections.*

Act

An action, as part of a unit test, is simply the piece of production code that is being tested. Usually, this is a single method or function in your code. Each test should have only a single action. Having more than one action will lead to messier tests and less certainty about where the code should change to make the test pass.

Assert

The result, or *assertion* (the expected result), is exactly what it sounds like. If you expect that the method being tested will return a 3, then you write an assertion that validates that expectation. The Single Assert Rule states that there should be only one assertion made per test. This does not mean that you can only assert once; instead, it means that your assertions should only confirm one logical expectation. As a quick example, you might have a method that returns a list of items after applying a filter. After setting up the test context, calling the method will result in a list of only one item, and that item will match the filter that we have defined. In this case, you will have a programmatic assert for the count of items in the list and one programmatic assert for the filter criterion we are testing.

Requirements

While this book is not about business analysis or requirement generation, requirements will have a huge impact on your ability to effectively test-drive an application. We will be providing requirements for this book in a format that lends itself very well to high-quality tests. We will also cover some scenarios where the requirements are less than optimal, but for most of this book the requirements have been labored over to ensure a <mark>high-quality definition of the systems</mark> we are testing.

Why are they important?

We firmly believe that quality requirements are essential to a well-developed solution. The requirements inform the tests and the tests shape the code. This axiom means that with poor requirements, the application will result in a lower quality architecture and overall design. With haphazard requirements, the resulting tests and application will be chaotic and poorly factored. On the bright side, even poorly thought out or written requirements aren't the death knoll for your code. It is our responsibility, as professional software developers, to correct bad requirements. It is our task to ask questions that will lead to better requirements.

User stories

User stories are commonly used in Agile software development for requirement definitions. The format for a user story is fairly simple and consists of three parts: `Role`, `Request`, and `Reason`.

```
As a <Role>
 I want <Request>
 So that <Reason>
```

Role

The role of the user story can provide a lot of information. When specifying the role, we have the ability to imply the capabilities of the user. Can the user access certain functionalities, or are they physically impaired in such a way that requires an alternate form of interaction with the system? We can also communicate the user's mindset. Having a new user could have an impact on the design of the user interface, in contrast to what an experienced user might expect. The role can be a generic user, a specific role, a persona, or a specific user.

Generic users are probably the most used and, at the same time, the least useful. Having a story that provides no insight into the user limits our decision making for this story by not restricting our context. If possible, ask your business analyst or product owner for a more specific definition of who the requirement is for.

Defining a specific role, such as Admin, User, or Guest, can be very helpful. Specific roles provide user capability information. With a specific role, we can determine if a user should even be allowed into the section of the application we are defining functionality for. It is possible that a user story will cause the modification of a user's rights within the system, simply because we specified a role instead of a generic user.

Using a persona is the most telling of the wide-reaching role types. A persona is a full definition of an imaginary user. It includes a name, any important physical attributes, preferences, familiarity with the subject of the application, familiarity with computers, and anything else that might have an impact on the imaginary user's interactions with the software. By having all this information, we can start to roleplay the user's actions within the system. We can start to make assumptions or decisions about how that user would approach or feel about a suggested feature and we can design the user interface with that user in mind.

Request

The request portion of the user story is fairly simple. We should have a single feature or a small addition to functionality that is being requested. Generally, the request is too large if it includes any joining words, such as *and* or *or*.

Reason

The reason is where the business need is stated. This is the opportunity to explain how the feature will add value to the company. By connecting the reason to the role, we can enhance the impact of the feature's usefulness.

A complete user story might look like the following:

```
As a Conference Speaker
 I want to search for nearby conferences by open submission date
 So that I may plan the submission of my talks
```

Gherkin

Gherkin is a ==style of requirements definitions== that is often used for ==acceptance criteria==. We can turn these requirements directly into code, and QA can turn them directly into test cases. The Gherkin format is generally associated with BDD, and it is used in Dan North's original article on the subject.

The Gherkin format is just as simple as the user story format. It consists of three parts: `Given`, `When`, and `Then`.

```
Given <Context>
 And Given <More Context>
 When <Action>
 Then <Result>
 And Then <More Results>
```

Givens

Because the Gherkin format is fairly simple, givens are broken out to one per contextual criterion. As part of specifying the context, we want to see any and all preconditions of this scenario. Is the user logged in? Does the user have any special rights? Does this scenario require any settings to be put into force before execution? Has the user provided any input on this scenario? One more thing to consider is that there should only be a small number of givens.

The more givens that are present in a scenario, the more likely it is that the scenario is too big or that the givens can somehow be logically grouped to reduce the count.

> *When we start writing our tests, a Given is analogous to the Arrange section of a test.*

When

The when is the action taken by the user. There should be one action and only one action. This action will depend on the context defined by the Given and output the result expected by the Then. In our applications, this is equivalent to a function or method call.

When we start writing our tests, a When is analogous to the Act section of a test.

Then

Thens equate to the output of the action. *Thens* describe what can be verified and tested from the output of a method or function, not only by developers but also by QA. Just like with the Givens, we want our *Thens* to be singular in their expectation. Also like Givens, if we find too many *Thens,* it is either a sign that this scenario is getting too big, or that we are over-specifying our expectations.

> When we start writing our tests, a ==Then is analogous== to the ==Assert== section of a test.

Complete acceptance criteria based on the user story presented earlier might look like the following:

```
Given I am a conference speaker
 And Given a search radius of 25 miles
 And Given an open submission start date
 And Given an open submission end date
 When I search for conferences
 Then I receive only conferences within 25 miles of my location
 And Then I receive only conferences that are open for submission within the specified date ra
```

Just like in life, not everything in this book is going to be perfect. Do you see anything wrong with the preceding acceptance criteria? ==Go on and take a few minutes to examine it==; we'll wait.

If you've given up, we'll tell you. The above acceptance criteria are just too long. There are too many Givens and too many Thens. How did this happen? How could we have created such a mistake? When we wrote the user story, we accidentally included too much information for the reason that we specified. If you go back and look at the user story, you will see that we threw `nearby` in the request. Adding `nearby` seemed harmless; it even seemed more correct. I, as the user, wasn't so interested in traveling too far for my speaking engagements.

When you start to see user stories or acceptance criteria getting out of hand like this, it is your responsibility to speak with the ==business analyst== or ==product owner== and work with them to ==reduce the scope of the requirements==. In this case, we can extract two user stories and several acceptance criteria.

Here is a full example of the requirements we have been examining:

```
As a conference speaker
 I want to search for nearby conferences
 So that I may plan the submission of my talks
Given I am a conference speaker
 And Given search radius of five miles
 When I search for conferences
 Then I receive only conferences within five miles of my location
Given I am a conference speaker
 And Given search radius of 10 miles
 When I search for conferences
 Then I receive only conferences within 10 miles of my location
```

```
Given I am a conference speaker
 And Given search radius of 25 miles
 When I search for conferences
 Then I receive only conferences within 25 miles of my location

As a conference speaker
 I want to search for conferences by open submission date
 So that I may plan the submission of my talks
Given I am a conference speaker
 And Given open submission start and end dates
 When I search for conferences
 Then I receive only conferences that are open for submission within the specified date range
Given I am a conference speaker
 And Given an open submission start date
 And Given an empty open submission end date
 When I search for conferences
 Then an INVALID_DATE_RANGE error occurs for open submission date
Given I am a conference speaker
 And Given an empty open submission start date
 And Given an open submission end date
 When I search for conferences
 Then an INVALID_DATE_RANGE error occurs for open submission date
```

One thing that we have not discussed is the approach to the content of the user stories and acceptance criteria. It is our belief that requirements should be as agnostic about the user interface and data storage mechanism as possible. For that reason, in the requirement examples, you'll notice that there is no reference to any kind of buttons, tables, modals/popups, clicking, or typing. For all we know, this application is running in a Virtual Reality Helmet with a Natural User Interface. Then again, it could be running as a RESTful web API, or maybe a phone application. The requirements should specify the system interactions, not the deployment environment.

In software development, it is everyone's responsibility to ensure high-quality requirements. If you find the requirements you have received to be too large, vague, user interface-dependent, or just unhelpful, it is your responsibility to work with your business analyst or product owner to make the requirements better and ready for development and QA.

Our first tests in C#

Have you ever created a new MVC project in Visual Studio? Have you noticed the checkbox towards the bottom of the dialog box? Have you ever selected, Create Unit Test Project? The tests created with this Unit Test Project are largely of little use. They do little more than validate that the default MVC controllers return the proper type. This is perhaps one step beyond, ItExists. Let's look at the first set of tests created for us:

```csharp
using System.Web.Mvc;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using SampleApplication.Controllers;

namespace SampleApplication.Tests.Controllers
{
  [TestClass]
  public class HomeControllerTest
  {
    [TestMethod]
    public void Index()
    {
      // Arrange
      HomeController controller = new HomeController();

      // Act
      ViewResult result = controller.Index() as ViewResult;

      // Assert
      Assert.IsNotNull(result);
    }

    [TestMethod]
    public void About()
    {
      // Arrange
      HomeController controller = new HomeController();

      // Act
      ViewResult result = controller.About() as ViewResult;

      // Assert
      Assert.AreEqual("Your application…", result.ViewBag.Message);
    }

    [TestMethod]
    public void Contact()
    {
      // Arrange
      HomeController controller = new HomeController();

      // Act
      ViewResult result = controller.Contact() as ViewResult;

      // Assert
      Assert.IsNotNull(result);
    }
  }
}
```

Here, we can see the basics of a test class, and the test cases contained within. Out of the box, Visual Studio ships with MSTest, which is what we can see here. The test class must be decorated with the [TestClass] attribute. Individual tests must likewise also be decorated with the

`[TestMethod]` attribute. This allows the test runner to determine which tests to execute. We'll cover these attributes and more in future chapters. Other testing frameworks use similar approaches that we'll discuss later, as well.

For now, we can see that the `HomeController` is being tested. Each of the public methods has a single test, for which you may want to create additional tests and/or extract tests to separate files in the future. Later we'll be covering options and best practices to help you arrange your files in a much more manageable fashion. All of this should be part of your *refactor* step in your *red, green, refactor* cycle.

Growing the application with tests

Perhaps you want to accept a parameter for one of your endpoints. Maybe you will take a visitor's name to display a friendly greeting. Let's take a look at how we might make that happen:

```
[TestMethod]
public void ItTakesOptionalName()
{
  // Arrange
  HomeController controller = new HomeController();

  // Act
  ViewResult result = controller.About("") as ViewResult;

  // Assert
  Assert.AreEqual("Your application description page.", result.ViewBag.Message);
}
```

We start by creating a test to allow for the `About` method to accept an optional string parameter. We're starting with the idea that the parameter is optional since we don't want to break any existing tests. Let's see the modified method:

```
public ActionResult About(string name = default(string))
{
  ViewBag.Message = "Your application description page.";
  return View();
}
```

Now, let's use the `name` parameter and just append it to our `ViewBag.Message`. Wait, not the controller. We need a new test first:

```
[TestMethod]
public void ItReturnsNameInMessage()
{
  // Arrange
  HomeController controller = new HomeController();

  // Act
  ViewResult result = controller.About("Fred") as ViewResult;

  // Assert
  Assert.AreEqual("Your application description page.Fred", result.ViewBag.Message);
}
```

And now we'll make this test pass:

```
public ActionResult About(string name = default(string))
{
  ViewBag.Message = $"Your application description page.{name}";
  return View();
}
```

Our first tests in JavaScript

To get the ball rolling in JavaScript, we are going to write a `Simple Calculator` class. Our calculator only has the requirement to add or subtract a single set of numbers. Much of the code you write in TDD will start very simply, just like this example:

```javascript
import { expect } from 'chai'

class SimpleCalc {
  add(a, b) {
    return a + b;
  }

  subtract(a, b) {
    return a - b;
  }
}

describe('Simple Calculator', () => {
  "use strict";

  it('exists', () => {
    // arrange
    // act
    // assert
    expect(SimpleCalc).to.exist;
  });

  describe('add function', () => {
    it('exists', () => {
      // arrange
      let calc;

      // act
      calc = new SimpleCalc();

      // assert
      expect(calc.add).to.exist;
    });

    it('adds two numbers', () => {
      // arrange
      let calc = new SimpleCalc();

      // act
      let result = calc.add(1, 2);

      // assert
      expect(result).to.equal(3);
    });
  });

  describe('subtract function', () => {
    it('exists', () => {
      // arrange
      let calc;

      // act
      calc = new SimpleCalc();

      // assert
      expect(calc.subtract).to.exist;
    });
```

```
    it('subtracts two numbers', () => {
      // arrange
      let calc = new SimpleCalc();

      // act
      let result = calc.subtract(3, 2);

      // assert
      expect(result).to.equal(1);
    });
  });
});
```

If the preceding code doesn't make sense right now, don't worry; this is only intended to be a quick example of some working test code. The testing framework used here is Mocha, and the assertion library used is `chai`. In the JavaScript community, most testing frameworks are built with BDD in mind. Each described in the code sample above represents a scenario or a higher-level requirements abstraction; whereas, each `it` represents a specific test. Within the tests, the only required element is the expect, without which the test will not deliver a valuable result.

Continuing this example, say that we receive a requirement that the add and subtract methods must be allowed to chain. How would we tackle that requirement? There are many ways, but in this case, I think I would like to do a quick redesign and then add some new tests. First, we will do the redesign, again driven by tests.

By placing `only` on a `describe` or a `test`, we can isolate that `describe`/`test`. In this case, we want to isolate our `add` tests and begin making our change here:

```
it.only('adds two numbers', () => {
  // arrange
  let calc = new SimpleCalc(1);

  // act
  let result = calc.add(2).result;

  // assert
  expect(result).to.equal(3);
});
```

Previously, we have changed the test to use a constructor that takes a number. We have also reduced the number of parameters of the `add` function to a single parameter. Lastly, we have added a result value that must be used to evaluate the result of adding.

The test will fail because it does not use the same interface as the class, so now we must make a change to the class:

```
class SimpleCalc {
  constructor(value) {
    this._startingPoint = value || 0;
  }

  add(value) {
    return new SimpleCalc(this._startingPoint + value);
  }
  ...
  get result() {
    return this._startingPoint;
  }
}
```

This change should cause our test to pass. Now, it's time to make a similar change for the

`subtract` method. First, remove the `only` that was placed in the previous example:

```
it('subtracts two numbers', () => {
  // arrange
  let calc = new SimpleCalc(3);

  // act
  let result = calc.subtract(2).result;

  // assert
  expect(result).to.equal(1);
});
```

Now for the appropriate change in the class:

```
subtract(value) {
  return new SimpleCalc(this._startingPoint – value);
}
```

Out tests now pass again. The next thing we should do is create a test that verifies everything works together. We will leave this test up to you as an exercise, should you want to attempt it.

Why does it matter?

So, why does all this matter? Why write more code than we have to? Because it's worth it. And to be honest, most of the time it isn't more code. As you take the time to grow your application with tests, simple solutions are produced. Simple solutions are almost always less code than the slick solution you might have come up with otherwise. And inevitably, slick solutions are error-prone, difficult to maintain, and often just plain wrong.

Summary

If you didn't before, you should now have a good idea of what TDD is and why it is important. You have been exposed to unit tests in C# and JavaScript and how writing tests first can help grow an application.

As we continue, we'll learn more about TDD. We'll explore what it means to write testable code.

In Chapter 2, *Setting Up the .NET Test Environment,* we'll set up your development environment and explore additional aspects of a unit test.

Setting Up the .NET Test Environment

In this chapter, we'll explore setting up your development environment. We'll be covering both C# and .NET. In the following chapter, we will focus on setting up a JavaScript and React environment. We'll delve into more examples, starting with the classic code kata entitled *FizzBuzz,* and then into more real-world samples from the *Speaker Meet* site.

In this chapter, you will gain an understanding of:

- Installing your IDE
- How to set up your testing framework
- Writing your first tests in C#

Installing the .NET Core SDK

Before you get started with the development environments, you will want to install the .NET Core SDK. You'll need to navigate to the .NET Core download page on the Microsoft website (https://www.microsoft.com/net/download/core). Select the proper installer for your system. For Windows machines, the `.exe` download is recommended.

Follow the onscreen instructions for the install wizard to install the .NET Core SDK.

Getting set up with VS Code

One benefit of choosing VS Code for your development is that it is an excellent IDE for both .NET and JavaScript. To get started using VS Code, you must first download the IDE.

Downloading the IDE

Visit the VS Code website (https://code.visualstudio.com/) and choose the proper version for your operating system:
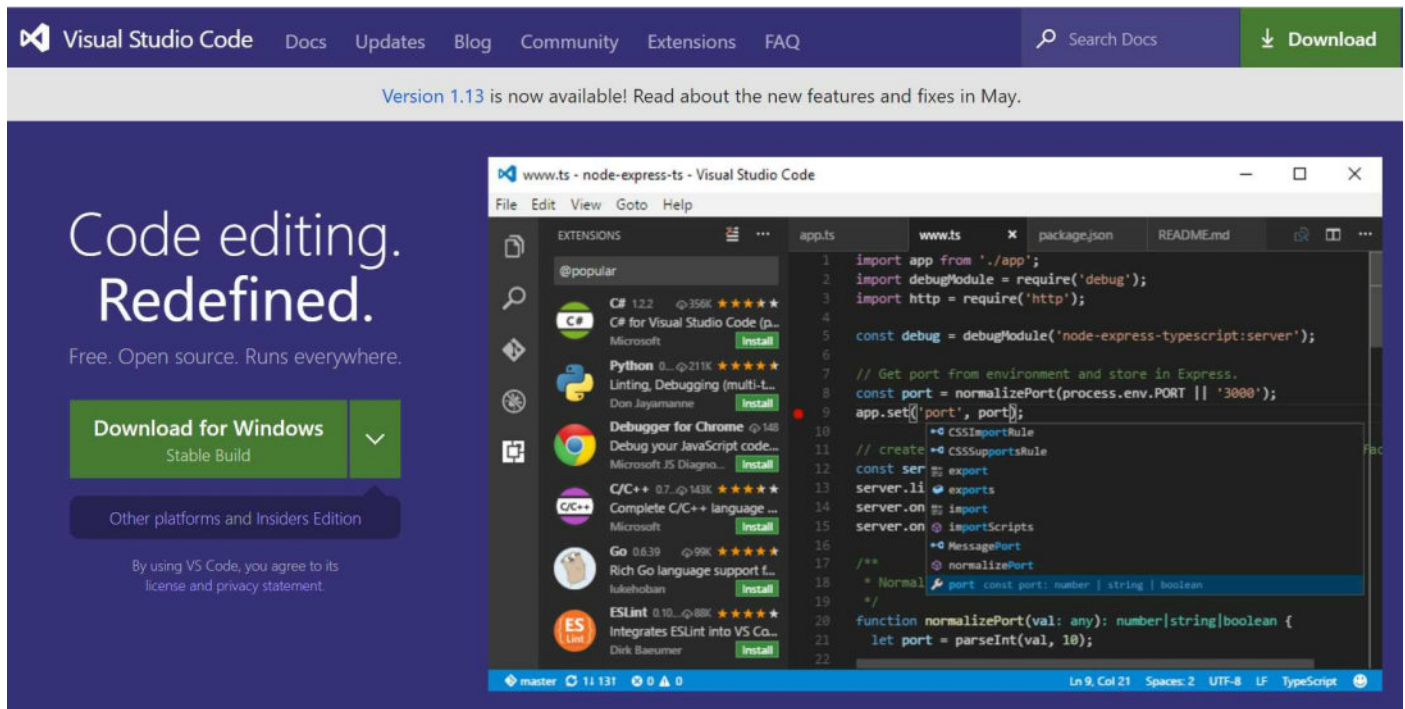
Installing the VS Code

Follow the instructions in the wizard to install the VS Code:
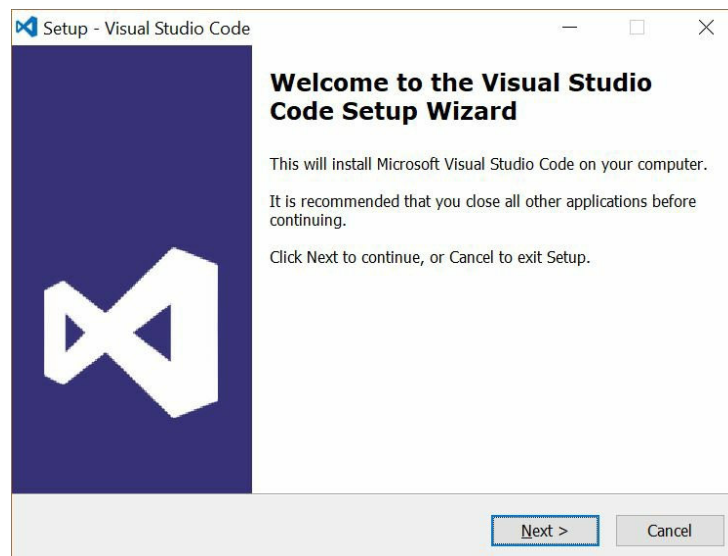
Be sure to read and accept the License Agreement:
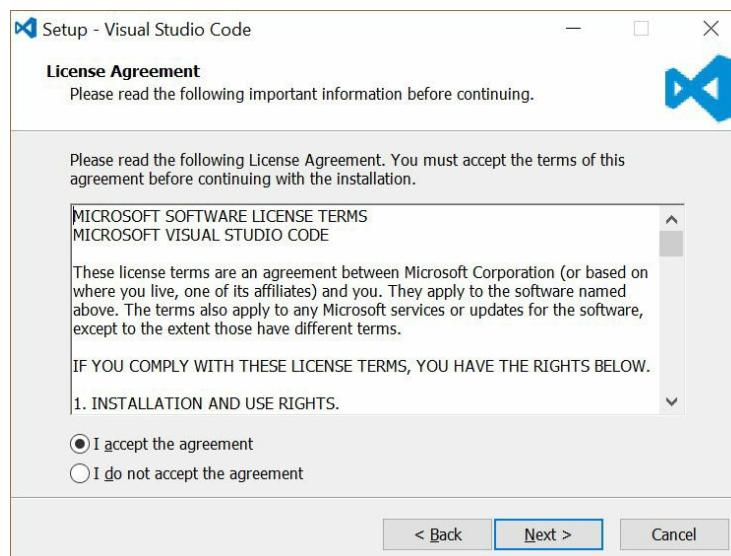
Choose a location on your hard drive to install VS Code. The default path is usually acceptable:
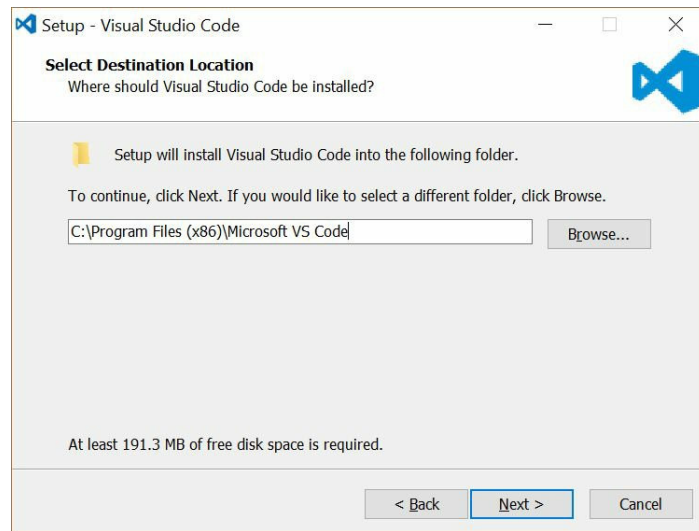
Choose to create a Start menu folder for the application, select a location, or choose to not create a Start menu folder:
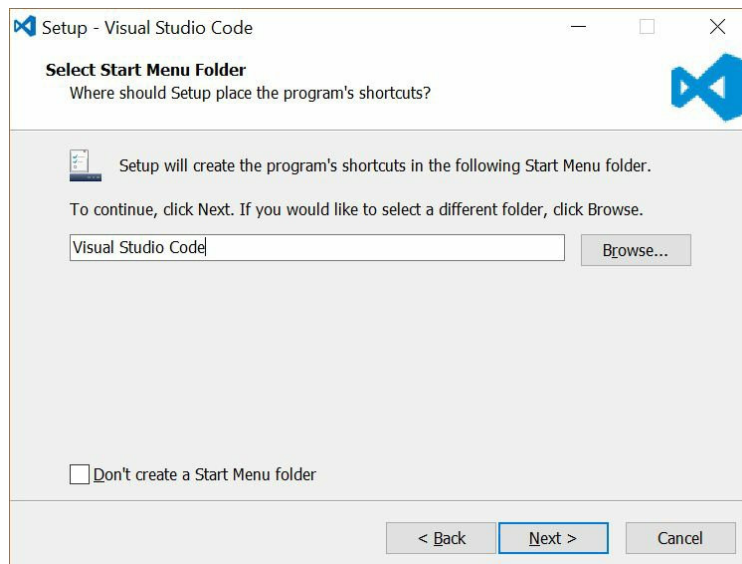
Select additional tasks. The default should be fine for our purposes, as shown in the following screenshot:
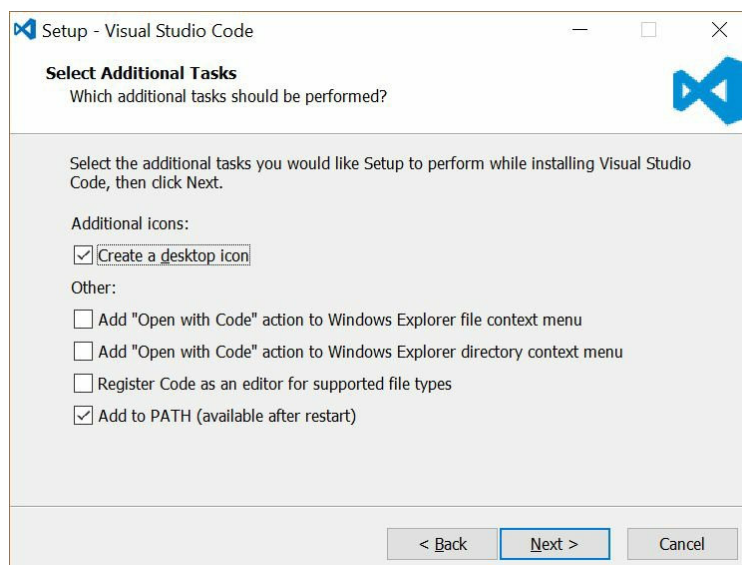
Review your installation settings and click on Install:
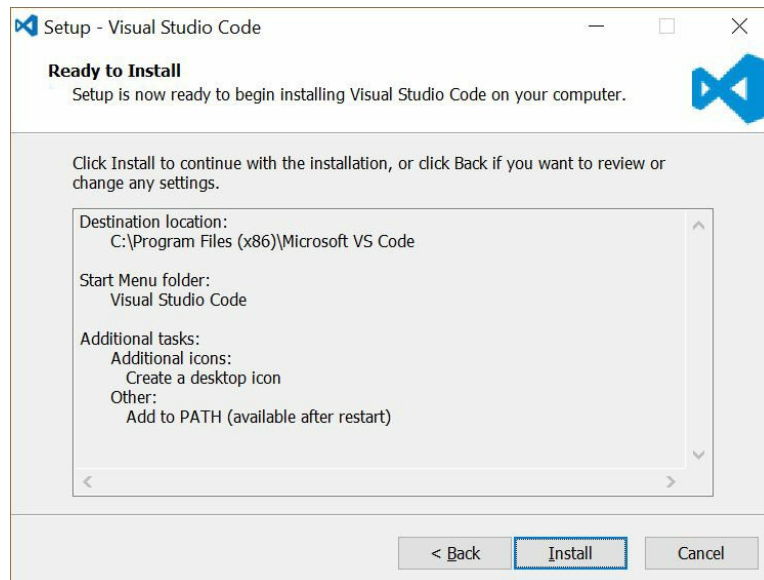
Once the install is finished, you're OK to launch the application:
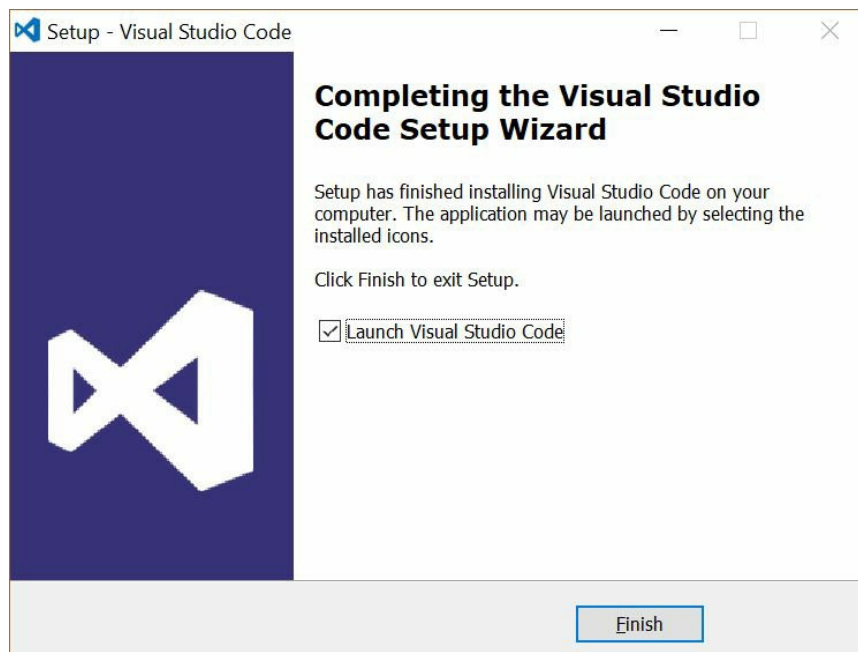
Adding extensions

VS Code is a fairly lightweight and bare-bones IDE. You'll need to install C# to get started. When you launched VS Code for the first time, your browser should have opened to the Getting Started page on the VS Code website. If it didn't, go there now (https://code.visualstudio.com/docs).

There are a variety of useful extensions that you can install from the marketplace. For now, all you will need is C#. At the time of writing, C# was listed near the top of the Top Extensions list. Click on the C# tile (or find it by searching in the marketplace) to learn more about this extension.

You should see that the installation instructions direct you to launch VS Code Quick Open (*Ctrl-P*) and paste the following command:

```
ext install csharp
```

From within VS Code, paste the command into the Quick Open section and press *Enter*. Find the C# version powered by OmniSharp and choose Install. Once the C# extension is installed, you will need to reload VS Code to activate the C# extension (choose reload).

Creating a project in VS Code

Now that your VS Code IDE is properly installed with the C# extension enabled, you are ready to create your first project.

With VS Code open, choose Open Folder from the File menu. Choose a location that is easily accessible. Many developers will create a `Development` folder on the root of their drive. Whatever convention you're used to will be fine. You now need to create an *MSTest* project.

Create a new folder named `Sample`. Open the Integrated Terminal window from the View menu or by using the shortcut keys (*Ctrl + `*). From within the Terminal window, type `dotnet new mstest` and hit *Enter*. Now, you need to restore your packages by typing `dotnet restore` into the Terminal window and hitting *Enter*.

You should now see a file named `UnitTest1.cs` within the `Sample` folder. If you open the file, it should look something like this:

```csharp
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Sample
{
  [TestClass]
  public class UnitTest1
  {
    [TestMethod]
    public void TestMethod1()
    {
    }
  }
}
```

Change the first test method to an `ItExists` test. Do this by changing the name to `ItExists` and trying to declare an instance to a class that does not yet exist:

```csharp
var sampleClass = new SampleClass();
```

You should see that your sample application will not compile and you will have received the error message, `The type or namespace 'SampleClass' could not be found (are you missing a using directive or an assembly reference?)`.

Now that you have a test failure (remember, failing to compile counts as a failing test in this instance), it's safe to move on to the *Green* step in our *red, green, refactor* cycle. Make the test pass by creating a definition for `SampleClass`. Feel free to create the class in the same file as your unit tests, just to get you started. This can always be extracted and moved to a more appropriate location later:

```csharp
public class SampleClass
{
}
```

Now that you've made the change, run the `test` command `dotnet test` and see the results:

```
Total tests: 1. Passed: 1. Failed: 0. Skipped: 0.
```

Continue exploring VS Code and growing your new class through tests. The C# and .NET examples throughout the rest of the book will be using Visual Studio Community. If you prefer, you may choose to stick with VS Code.
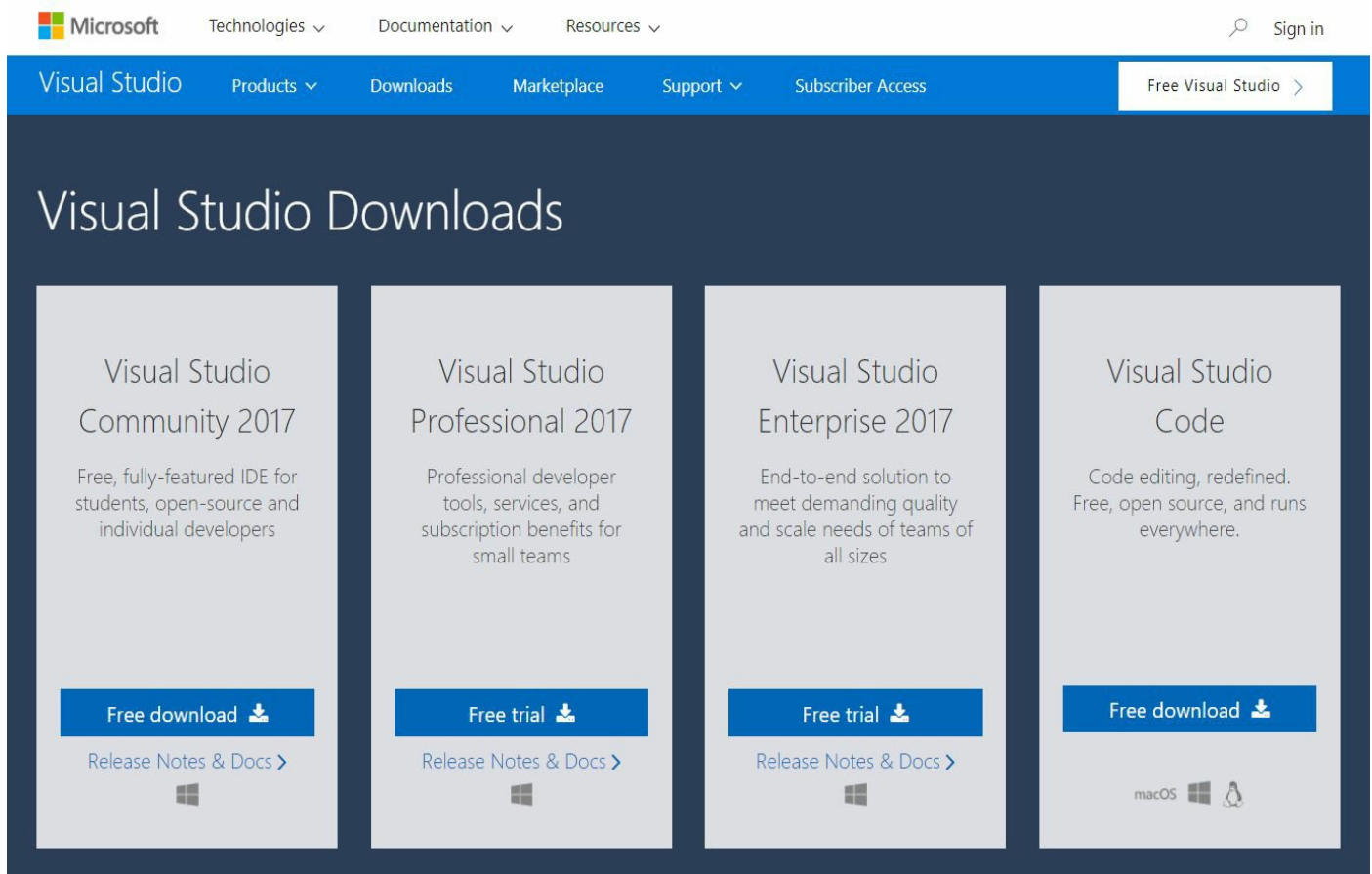
Setting up Visual Studio Community

Most C# and .NET developers will be familiar with Visual Studio. There are a variety of versions available, ranging from free to many thousands of dollars annually. As of this writing, the Enterprise version was the most fully featured version, offering some of the best features for tests and testing. For our purposes, we'll be using Visual Studio Community. This is a free, fully featured development environment that should suit us well.

The Community edition does have some very important caveats. There are limitations to the software license and the use of the Community edition, based on the terms of the license agreement. Please make sure to read the terms before deciding to use Visual Studio Community edition to develop software that you intend to sell. The current terms can be found at https://www.visualstudio.com/license-terms/mlt553321/.

Downloading Visual Studio Community

To get started, download Visual Studio Community (https://www.visualstudio.com/downloads/). Feel free to explore and compare the different versions of Visual Studio while you're there:

Installing Visual Studio Community

The wizard for installing Visual Studio Community is a little different from the install wizard for VS Code. Of course, to get started, you'll need to read and agree to the license agreement.

At a minimum, you'll want to choose ASP.NET and web development and .NET Core cross-platform development if you plan to follow along with the book. We've also chosen to include ASP.NET MVC 4, .NET Framework 4.6.2 development tools, and .NET Framework 4.7 development tools from the right pane, or from the *Individual components* tab. You might want to explore other components and/or language packs, as well.

Switching to xUnit

MSTest has long shipped with Visual Studio. There are a few other options when it comes to testing frameworks for C# and .NET. Many of these frameworks have feature parity and differ only slightly in their choices of attributes, assertions, and exception handling. Among the top contenders for testing frameworks is xUnit. Many developers actually prefer this to MSTest and would argue that it is more feature-rich and has stronger community support. Arguments aside, we'll be using xUnit for our C# and .NET tests from here on out.

Feel free to stick with MSTest if you prefer. Just know that you'll need to account for the semantic differences (such as *TestMethod* vs *Fact*) and slight differences in functionality.

Code katas

What is a code kata? Code katas are nothing more than repeatable exercises. Generally, these exercises are meant to take no more than 20 minutes to complete. Most code katas are directed at a specific classification of a problem to solve. We'll be utilizing the classic example, FizzBuzz, as a way to get you more comfortable with TDD using xUnit.

FizzBuzz

The rules of FizzBuzz are quite simple. If the number provided is divisible by 3, then you must return `Fizz`. If the number supplied is divisible by 5, then you must return `Buzz`. If the number is divisible by both 3 and 5, then you must return `FizzBuzz`. If it is divisible by neither 3 nor 5, then simply return the number itself.

There are a plethora of options in which to solve the problem. It can be solved in nearly every programming language, in a variety of different ways. What's important here is to practice the techniques of solving the problem simply and effectively.

Let's get started.

Creating the test project

Within Visual Studio Community, create an xUnit test project by choosing New | Project from the File menu or by using the shortcut keys (*Ctrl - Shift - N*). Under .NET Core, choose xUnit Test Project. Give your project the name `CodeKata` and click on OK. You will see a filename `UnitTest1.cs`. This file is fine to get you started. Let's create our first test.

The Given3ThenFizz test

The first test method in the `UnitTest1.cs` file is named `Test1`. Let's change the name of this method to `Given3ThenFizz` and write our first test:

```
[Fact]
public void Given3ThenFizz()
{
  // Arrange
  // Act
  var result = FizzBuzz(3);

  // Assert
  Assert.Equal("Fizz", result);
}
```

Note that the `Fact` attribute and `Assert.Equal` assertion differ only slightly from our previous MSTest example. We're leaving the `Arrange`, `Act`, and `Assert` comments in place, and recommend you do the same. These comments will help you as you get started. They'll also serve to help describe the process to any developers that come behind you in the future.

Now, run the test to see whether it passes by selecting Run | All Tests from the Test menu, or by using the shortcut keys (*Ctrl + R, A*). You should see a compilation error. Let's resolve the error by creating a `FizzBuzz` method preceding our `test` class. Once you've created the `FizzBuzz` method, rerun your test to see it pass. Remember, based on the third law of TDD, you should only write enough code to make it pass:

```
private object FizzBuzz(int value)
{
  return "Fizz";
}
```

The Given5ThenBuzz test

Our next requirements state that we must return `Buzz` when 5 is supplied. Let's write that test:

```
[Fact]
public void Given5ThenBuzz()
{
  // Arrange
  // Act
  var result = FizzBuzz(5);

  // Assert
  Assert.Equal("Buzz", result);
}
```

How might we make that test pass? Perhaps a simple ternary operator? Let's take a look at what that might look like:

```
private object FizzBuzz(int value)
{
  return value == 3 ? "Fizz" : "Buzz";
}
```

You might see a problem with our algorithm already. That's OK! We're not done yet. We've only gotten as far as the tests have guided us, and so far we're passing all of our tests. Let's move on to the next most interesting test.

The Given15ThenFizzBuzz test

You might want to write a test method entitled `GivenDivisibleBy3and5ThenFizzBuzz`, but that may be too large of a leap at this point. We know that the first such number divisible by 3 and 5 is 15, so it might make more sense to start with this:

```
[Fact]
public void Given15ThenFizzBuzz()
{
  // Arrange
  // Act
  var result = FizzBuzz(15);

  // Assert
  Assert.Equal("FizzBuzz", result);
}
```

How would you choose to make this test pass? Would you use an *if/else* statement? Perhaps a *switch* statement? We'll leave this one as an exercise for the reader. Feel free to make this test pass in any way that you're comfortable with implementing. Remember to run your tests along the way to ensure you don't introduce a breaking change. If you do experience a test failure, feel free to ignore a test (*Ignore* attribute in MSTest, *Skip* parameter in xUnit), but only one test, while you fix your error(s).

The Given1Then1 test

We've covered `Fizz`. We've covered `Buzz`. And, we've covered `FizzBuzz`. Now we must account for numbers that are divisible by neither 3 nor 5. Remember, in the event that a number is divisible by neither 3 nor 5, we simply return the number supplied. Let's take a look at this test:

```
[Fact]
public void Given1Then1()
{
  // Arrange
  // Act
  var result = FizzBuzz(1);

  // Assert
  Assert.Equal(1, result);
}
```

Theories

This is great! Things are going quite smoothly. Hopefully, you're starting to get the hang of Test-Driven Development. Now, let's look into a slightly more advanced test method using the `Theory` and `InlineData` attributes.

Looking back at our tests, we see that we have a `test` method named `Given15ThenFizzBuzz`. While this is fine, it's a little too specific. Remember, our requirement was that, if the number is divisible by 3 and 5, then we should return `FizzBuzz`. Let's ensure we didn't take too big a leap in logic by writing a new test. This time, we'll supply a number of values, expecting the same results:

```
[Theory]
[InlineData(0)]
[InlineData(15)]
[InlineData(30)]
[InlineData(45)]
public void GivenDivisibleBy3And5ThenFizzBuzz(int number)
{
  // Arrange
  // Act
  var result = FizzBuzz(number);

  // Assert
  Assert.Equal("FizzBuzz", result);
}
```

When you run the test suite, you should now see four new passed test results. If you do experience a failure, the results pane in the Test Explorer window should provide a detailed explanation as to which test failed.

Now, do the same thing for `Fizz` and `Buzz` by creating two more test cases using `Theories` and `InlineData`. Go ahead and add `GivenDivisibleBy3ThenFizz`, `GivenDivisibleBy5ThenBuzz`, and `GivenNotDivisibleBy3or5ThenNumber`. Be sure to run your test suite after adding each test and `InlineData` value, fixing any failures along the way.

Solution to the FizzBuzz Problem

What we came up with looks something like this:

```
private object FizzBuzz(int value)
{
  if (value % 15 == 0)
    return "FizzBuzz";

  if (value % 5 == 0)
    return "Buzz";

  if (value % 3 == 0)
    return "Fizz";

  return value;
}
```

Don't worry if you chose to solve the problem a different way. The important thing is that you gained knowledge and understanding during this exercise. Additionally, you now have a comprehensive set of tests and you're comfortable refactoring and/or adding functionality.

What is Speaker Meet?

We're using the *Speaker Meet* application as a case study in Test-Driven Development. Speaker Meet is a website dedicated to connecting technology speakers, user groups, and conferences. Anyone who has helped organize a user group or tech conference knows it's often difficult to find speakers. And as technology speakers, it's often difficult to coordinate speaking engagements outside your immediate area. Speaker Meet helps bring technology speakers and communities together.

At the time of writing, the application is still in development, but it is a terrific platform to explore TDD concepts and principles as they relate to real-world applications. Speaker Meet consists of a RESTful API in .NET with a **Single Page Application** (**SPA**) in JavaScript, utilizing the React library.

Web API project

For our first exercise, we'll be creating a new API endpoint. This new endpoint will return a list of speakers based on a supplied search term. We'll be utilizing this endpoint in our React examples in a later chapter.

Listing Speakers (API)

A list of Speakers will be returned from the database by accessing the back-end API. Before starting on writing the code, a set of requirements must first be established. It's difficult to know where to begin if an agreed upon set of functionality hasn't been defined.

Requirements

Below are the requirements, you might expect to receive from a business analyst or product owner. These are often a good starting point for a broader conversation. If something is not clear, it's best to resolve any ambiguity before you begin.

```
As a conference organizer
 I want to search for available speakers
 So that I may contact them about my conference

Given I am a conference organizer
 And Given a speaker in mind
 When I search for speakers by name
 Then I receive speakers with a matching first name

Given I am a conference organizer
 And Given a speaker in mind
 When I search for speakers by name
 Then I receive speakers with a matching last name
```

Upon speaking with our product owner, we determined that by the requirement of *matching*, what was truly desired was a *starts-with* match. If a conference organizer were to search for the string "Jos," the results for *Josh, Joshua, Joseph*, should be returned by the search routine.

A new test file

We'll start by creating a new test file. Let's name this file `SpeakerControllerSearchTests.cs`. Now, create the first test, `ItExists`:

```
[Fact]
public void ItExists()
{
  var controller = new SpeakerController();
}
```

To make this compile, you'll need to create a Web API controller called `SpeakerMeetController`. Add a new ASP.NET Core Web Application project to your solution. Give your project a name of `SpeakerMeet.API` and choose the Web API template to get started. Add a reference to this project from your test project and add the appropriate using statement.

Now, let's ensure that there is a `Search` endpoint available. Let's create another test:

```
[Fact]
public void ItHasSearch()
{
  // Arrange
  var controller = new SpeakerController();

  // Act
  controller.Search("Jos");
}
```

Make this test pass by creating a `Search` method that accepts a string.

Let's confirm that the `Search` action result returns an `OkObjectResult`:

```
[Fact]
public void ItReturnsOkObjectResult()
{
  // Arrange
  var controller = new SpeakerController();

  // Act
  var result = controller.Search("Jos");

  // Assert
  Assert.NotNull(result);
  Assert.IsType<OkObjectResult>(result);
}
```

Note the multiple `Asserts`. While we want to limit our tests to a single `Act`, sometimes it is acceptable, even necessary, to have multiple `Asserts`.

Once the `ItReturnsOkObjectResult` test passes, you should delete the `ItExists` and `ItHasSearch` tests. Remember, we want to finish the *red, green, refactor* cycle and keep our code neat and clean. This includes the test suite, so if you have tests that are no longer valid or add no value, then you should feel free to remove them. You don't want to have to maintain more code than is required. This will help your test suite stay relevant and run nice and fast.

Now, let's test that the result is a collection of speakers:

```
[Fact]
public void ItReturnsCollectionOfSpeakers()
{
  // Arrange
  var controller = new SpeakerController();

  // Act
  var result = controller.Search("Jos") as OkObjectResult;

  // Assert
  Assert.NotNull(result);
  Assert.NotNull(result.Value);
  Assert.IsType<List<Speaker>>(result.Value);
}
```

We're starting to get a little redundant here. Now is a good time to refactor our tests to make them cleaner. Let's extract the creation of the `SpeakerController` and initialize this value in the constructor. Be sure to remove the creation in your tests and use this new instance:

```
private readonly SpeakerController _controller;

public SpeakerControllerSearchTests()
{
  _controller = new SpeakerController();
}
```

Finally, we're ready to start testing the value of the results. Let's write a test entitled `GivenExactMatchThenOneSpeakerInCollection`:

```
[Fact]
public void GivenExactMatchThenOneSpeakerInCollection()
{
  // Arrange
  // Act
  var result = _controller.Search("Joshua") as OkObjectResult;

  // Assert
  var speakers = ((IEnumerable<Speaker>)result.Value).ToList();
  Assert.Equal(1, speakers.Count);
}
```

To get this test to work, we'll need to hard-code some data. Don't worry, we're building this application slowly. The hard-coded data will be removed at a later point:

```
[Fact]
public void GivenExactMatchThenOneSpeakerInCollection()
{
  // Arrange
  // Act
  var result = _controller.Search("Joshua") as OkObjectResult;

  // Assert
  var speakers = ((IEnumerable<Speaker>)result.Value).ToList();
  Assert.Equal(1, speakers.Count);
  Assert.Equal("Joshua", speakers[0].Name);
}
```

Ensure that our search string is not case-sensitive:

```
[Theory]
[InlineData("Joshua")]
[InlineData("joshua")]
[InlineData("JoShUa")]
public void GivenCaseInsensitveMatchThenSpeakerInCollection (string searchString)
```

```
{
  // Arrange
  // Act
  var result = _controller.Search(searchString) as OkObjectResult;

  // Assert
  var speakers = ((IEnumerable<Speaker>)result.Value).ToList();
  Assert.Equal(1, speakers.Count);
  Assert.Equal("Joshua", speakers[0].Name);
}
```

Next, we need to test to verify that, if the string provided does not match any of our data, then an empty collection is returned:

```
[Fact]
public void GivenNoMatchThenEmptyCollection()
{
  // Arrange
  // Act
  var result = _controller.Search("ZZZ") as OkObjectResult;

  // Assert
  var speakers = ((IEnumerable<Speaker>)result.Value).ToList();
  Assert.Equal(0, speakers.Count);
}
```

And finally, we'll test that any speaker that begins with our search string will be returned:

```
[Fact]
public void Given3MatchThenCollectionWith3Speakers()
{
  // Arrange
  // Act
  var result = _controller.Search("jos") as OkObjectResult;

  // Assert
  var speakers = ((IEnumerable<Speaker>)result.Value).ToList();
  Assert.Equal(3, speakers.Count);
  Assert.True(speakers.Any(s => s.Name == "Josh"));
  Assert.True(speakers.Any(s => s.Name == "Joshua"));
  Assert.True(speakers.Any(s => s.Name == "Joseph"));
}
```

Here's what the code we came up with looks like. Your implementation may vary somewhat:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;

namespace SpeakerMeet.Api.Controllers
{
  [Route("api/[controller]")]
  public class SpeakerController : Controller
  {
    [Route("search")]
    public IActionResult Search(string searchString)
    {
      var hardCodedSpeakers = new List<Speaker>
      {
        new Speaker{Name = "Josh"},
        new Speaker{Name = "Joshua"},
        new Speaker{Name = "Joseph"},
        new Speaker{Name = "Bill"},
      };

      var speakers = hardCodedSpeakers.Where(x => x.Name.StartsWith(searchString, StringCompar

      return Ok(speakers);
```

```
    }
  }

  public class Speaker
  {
    public string Name { get; set; }
  }
}
```

Summary

You should now feel quite comfortable with your .NET development environment. The .NET Core SDK should now be installed and your IDE configured. You've had some exposure to unit tests and continuous test runners in Visual Studio and VS Code.

In Chapter 3, *Setting Up the JavaScript Environment,* we'll focus on getting our JavaScript environment set up.

Setting Up a JavaScript Environment

In this chapter, we'll explore setting up your JavaScript development environment with examples in pure JavaScript and React.

In this chapter, you will gain an understanding of:

- Installing your IDE
- How to set up your testing framework
- Writing your first tests in JavaScript

Node.js

Node.js, commonly called Node, is practically a requirement for doing modern web application development. In this section, we will discuss what Node is exactly, provide reasons why you need Node, and, finally, talk about where you can get Node installation instructions.

If you are already familiar with these subjects, then feel free to jump to the next section, where we discuss NPM in a similar fashion.

What is Node?

Node was created in late 2009 by Ryan Dahl. Based on Chrome's V8 engine, Node provides a JavaScript runtime built for the purpose of providing evented, non-blocking I/O (input/output) for serving web applications.

At the time, Chrome had created the fastest JavaScript engine available. At the same time, they had decided to open-source the code for it. For these two extremely compelling reasons, Node decided to use the V8 engine.

Ryan Dahl was unhappy with the performance, at the time, of the very popular Apache HTTP server. One of the problems with the way that Apache was handling concurrent connections was that it was creating a new thread for each connection. Task creation and task switching between these threads are both CPU-and memory-intensive. For these reasons, instead of using threads for concurrent connections, Dahl decided to write Node with the intent of using an event loop coupled with a callback paradigm.

Why do we need Node?

To perform TDD in JavaScript for a modern web application, we absolutely need Node. When writing a modern web application, you will very likely be using one of these popular frameworks: ReactJS, Angular, Ember.js, Vue.js, or Polymer. The majority of these applications require a compilation step in Node.

Another reason for using Node is that we want to take advantage of new features in JavaScript. Node doesn't support these features itself, but libraries have been written that will allow you to transpile the newer versions of JavaScript, ECMAScript 2015+, into a version of JavaScript that is supported by your target browsers.

Lastly, for the purposes of this book, we need Node so that we can run our tests. Later, we will discuss how we can also run our tests continuously while we are writing our code. This is known as continuous testing and is a must-have for rapid development.

Installing Node

There are several options for installing Node on your machine. We will cover installing manually and installing from a package management repository.

The benefits of using a package management repository are many. The main reason that you would want to install this way would be the benefit of version management. Node updates versions frequently, and using a package manager can help to notify you of available updates. It can also help install those updates in a simple and efficient manner. We will start with a manual install, followed by installing using a Linux package manager, a Mac OSX package manager, and finally a Windows package manager.

To install Node manually, open your favorite browser and go to http://nodejs.org. You should see something similar to the following screenshot. Regardless of your operating system, the Node website will have a download link for both current and **Long Term Support** (**LTS**) versions of Node installation files. For Windows and Mac, the Node website provides installers. For Linux, Node provides binaries and source code. Assuming you are familiar with your operating system, the installation process is fairly straightforward and shouldn't present any issues.
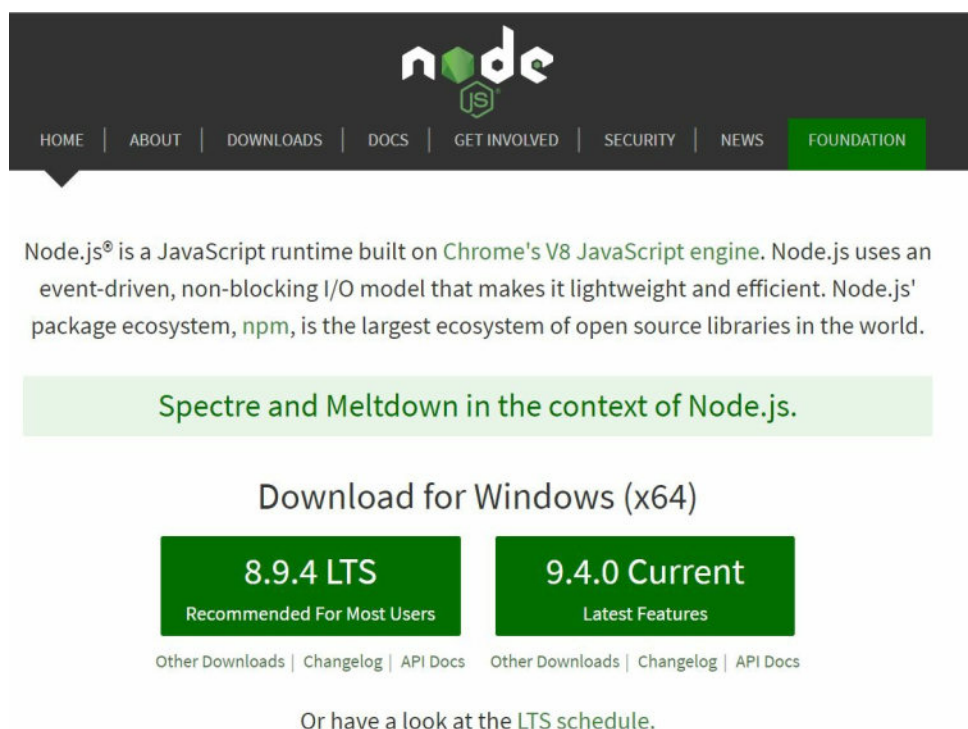
Package managers greatly simplify the installation of many applications. If you are unfamiliar with package managers in general, they are based on the concept of having a repository of applications and tools that are available for installation on the system the package manager is for. Almost every system available has a package manager for it now. Linux has a different package manager for many distributions. Mac uses a system called *Homebrew,* and Windows has a package manager named *Chocolatey*.

Linux

First, we will cover using the Ubuntu package manager, called `apt`, as Ubuntu is one of the most popular Linux systems. If you are using a different distribution, the process should be very similar. The only difference is the name of your package manager. Open a terminal window and enter the following commands to install Node for Ubuntu:

```
$ sudo apt-get update
$ sudo apt-get install nodejs
```

It's that simple; now the latest version of Node is installed and ready for you to begin using. These same commands will update Node when a new version is available.

Mac OSX

Mac doesn't come with a package manager preinstalled. To install Homebrew, you must open a Terminal and execute the following command:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Now that you have Homebrew installed, you also have one more requirement that you must fulfill. You must install Apple's Xcode, which can be found by searching the App Store. Just like any other application on a Mac, once you have found it, just click the Install application button, and Xcode will download and install:
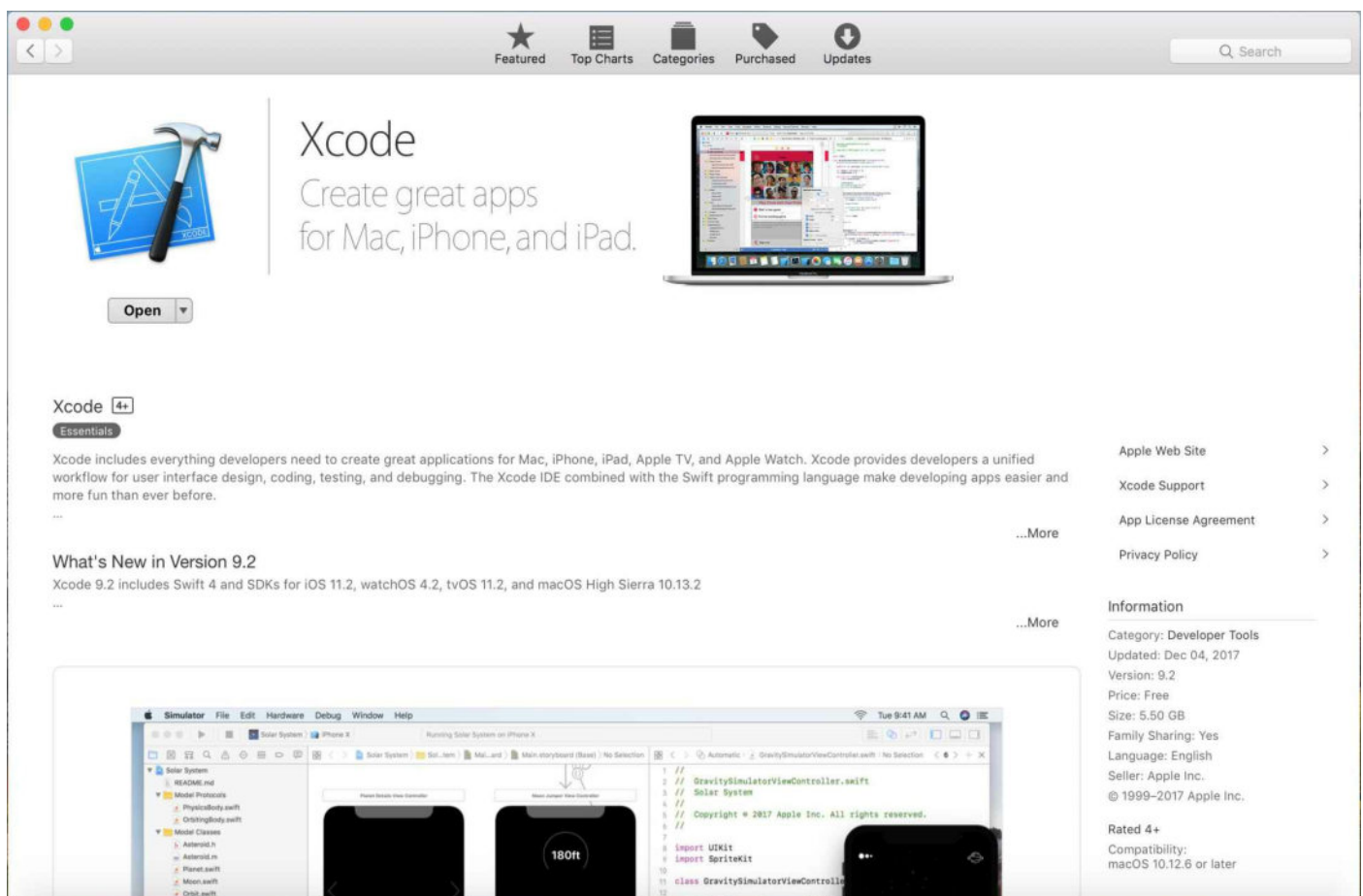
Now that we have both prerequisites for Node installed on the system, installation is extremely simple. From a Terminal window, execute the following command:

```
$ brew install node
```

Updating Node is also just as simple. Occasionally, when you want to update, execute the following command:

```
$ brew update
$ brew upgrade node
```

You now have the latest version of Node installed on your Mac.

Windows

Windows also has a package manager. Just like Mac, the Windows package manager does not come preinstalled. The package manager for Windows is named Chocolatey and can be found at https://chocolatey.org:
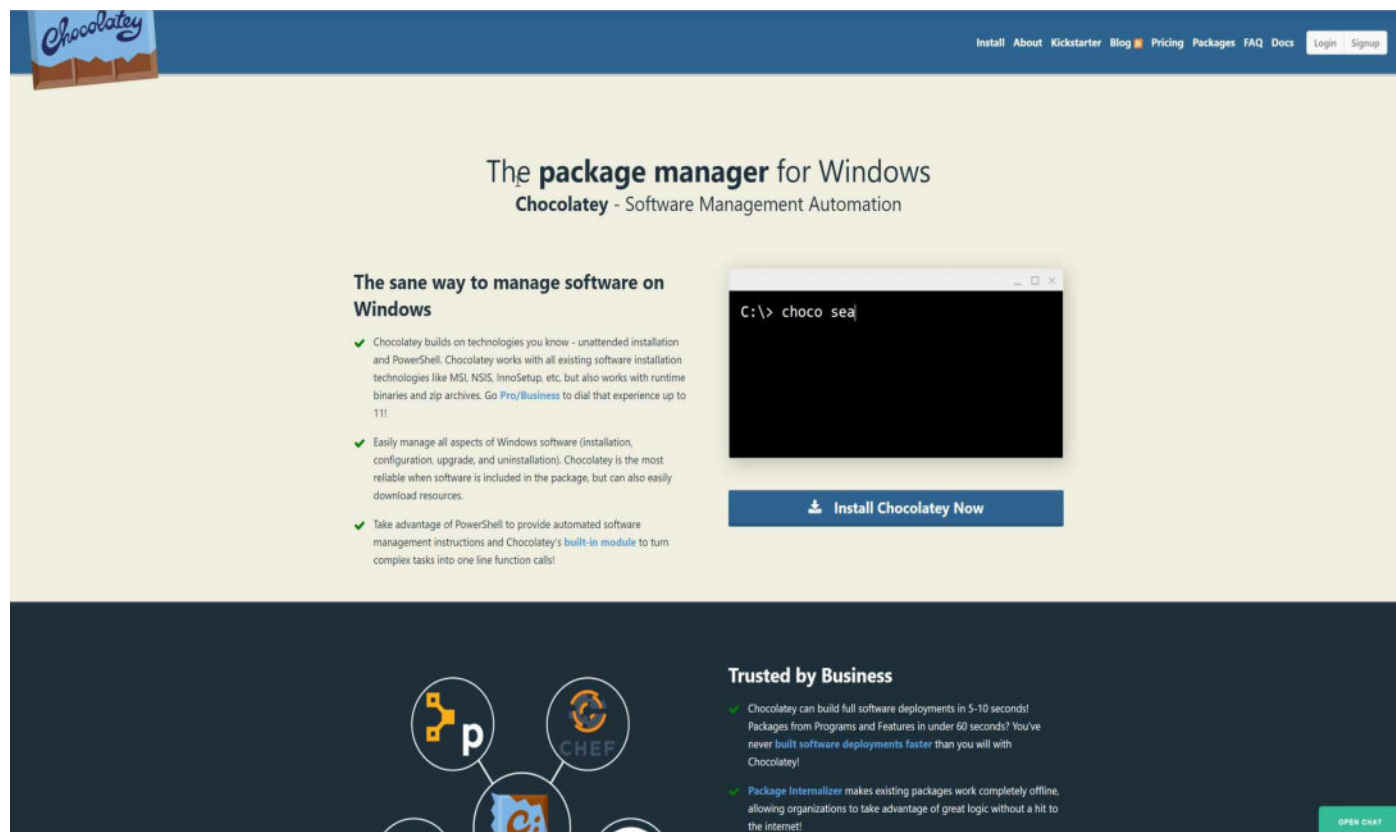
To install Chocolatey, open a Command Prompt (`cmd.exe`) as administrator and execute the following command:

```
@"%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell.exe" -NoProfile -ExecutionPolicy Byp
```

After the Chocolatey installation has finished, you may need to restart the command window before you can use it. Restart the Command Prompt as an administrator, and then execute the following commands to install Node on windows using the Chocolatey package manager:

```
C:\>choco install nodejs
C:\>refreshenv
```

Once you've executed the first command, you will be prompted to execute a script. You will need to agree to run the script to install Node.

To upgrade Node using Chocolatey, execute the following command:

```
C:\>choco upgrade nodejs
```

You may need to agree to run the installation script. If prompted, simply hit the *Y* key and press

Enter. You now have the latest version of Node.

NPM

NPM is a critical piece of the Node environment and community. Without NPM, Node would not have taken off quite like it has. In this section, we will discuss what NPM is, what NPM isn't, why you need NPM for doing Node development, and finally, where you can get NPM and how to install it.

What is NPM?

Initially released in early 2010, **NPM** (**Node Package Manager**). NPM was written by Isaac Z. Schlueter, and is now maintained by a team of developers. Although NPM has Node in the acronym, it can also be used to manage packages for the browser.

In the past few years, many package managers were created specifically for browser packages. One of the most prominent is *Bower*. These secondary package managers were created because NPM was perceived as a package manager best suited for, or perhaps only suited for, managing Node packages. This belief has subsided; however, Bower's own website now suggests that it not be used.

Why do we need NPM?

While you do need to install NPM, you don't necessarily have to use it. In late 2016, Facebook released an alternative package manager named *Yarn*, which uses the NPM registry, so all of your favorite packages are available.

There are likely other package management alternatives to NPM as well. These alternative package managers are important because they drive improvements in NPM, but ultimately, they will likely fade and NPM will continue to be the preferred package manager for Node, and for JavaScript in general. If you do decide to use an alternative package manager such as Yarn, you will need to install it using NPM.

Installing NPM?

Good news; you already have NPM installed if you have gone through the process of installing Node. You may occasionally want to upgrade NPM outside Node's release cycle. To attempt an upgrade, simply open your operating system's preferred console or Terminal window and execute the following command:

```
>npm install -g npm
```

NPM is just another executable on your computer. It takes a list of arguments or parameters. In this case, we are asking NPM to install a package. The second argument, `-g`, tells NPM that we want to install the requested package globally. Lastly, the package we are asking NPM to install is NPM.

A quick introduction to JavaScript IDEs

While you don't need an **IDE** (**Integrated Development Environment**) per se, you will need a text editor. Why not have a text editor that does a little bit of the heavy lifting for you? There are, essentially, two types of IDEs available for JavaScript development. The first kind is really more of a text editor than anything else, whereas the second kind is a full-blown editor with compiling and source control built in.

While you can work on JavaScript with only a simple text editor and a console/Terminal window, we recommend using something with at least a little more power.

Visual Studio Code

Visual Studio Code, as described in the C# section, is a lightweight editor based on the Electron framework and developed in TypeScript, a language designed by Microsoft to extend JavaScript with static types. TypeScript compiles to JavaScript, so ultimately Visual Studio Code is a JavaScript application.

Why Visual Studio Code?

For working with JavaScript, there are several reasons why you might choose Visual Studio Code or one of the other Electron-based editors. VSCode is lightweight, has an extensive plugin architecture, is integrated with source control, and is very easy to set up and use.

Installing Visual Studio Code

Installing VSCode is extremely simple. If you have already followed along in the *C#* section, then you likely have VSCode installed already. If not, here are some alternative ways to install it that were not discussed there.

Linux

To install on Linux (again, this is an example for Ubuntu), simply execute the following commands:

```
sudo apt-get update
sudo apt-get install code # or code-insiders
```

Mac

Unfortunately, I don't have a fancy command-line way to install VSCode on a Mac. Instead, go to the Visual Studio Code homepage at https://code.visualstudio.com and follow the installation instructions there.

Windows

For Windows, just like with Node, we can install VSCode using Chocolatey. To install using Chocolatey (https://chocolatey.org/install), execute the following command in a console window. Remember, you may want to run the console as an administrator:

```
C:\> choco install visualstudiocode
```

Installing the plugins you will need

The two plugins that we would recommend getting are `npm` and `npm-intellisense` because they will aid in the flow and provide hints when you are not 100% sure that you are using the correct package name.

Configuring the testing environment

Visual Studio Code offers built-in test running capabilities. We are not going to choose those options for JavaScript development, however. For the purposes of test driving our application and demonstrating our approach to testing, we think using the available Terminal inside VSCode will be more appropriate and better suited to the flow that will be used.

WebStorm

WebStorm is a full-blown IDE written by JetBrains in Java.

Why WebStorm?

WebStorm basically comes with everything you need to develop a JavaScript-based application. It also supports many of the JavaScript ecosystem alternatives to JavaScript, such as TypeScript, Flow, and React JSX. WebStorm also integrates seamlessly with many code-quality tools such as ESLine, TSLint, and JSHint.

The only downside to WebStorm is that it does cost money. But, when you look at it, a paid product is actually a good thing. The company making the paid product has a good reason to continue maintaining it. JetBrains offers the purchase of WebStorm through a single purchase or through a subscription. We suggest the subscription, as the upfront cost to you is minimized and JetBrains has more motivation to keep you happy this way.

Installing WebStorm

To install WebStorm, we are going to use a newer program created by JetBrains called *The JetBrains Toolbox App*. The Toolbox App is designed to track version updates and provide a common launching point for all JetBrains products. Once installed, it becomes very easy to install any of the JetBrains tools.

Linux

There does not appear to be, at the time of writing, a way to install ToolBox or WebStorm from `apt-get`. So, we will have to do it the hard way. Go to the ToolBox download page at https://www.jetbrains.com/toolbox/app/ and download the Linux tarball. Then, open a Terminal to your download directory. Once there, execute the following commands:

```
mv jetbrains-toolbox-<version>.tar.gz <application directory root>/jetbrains-toolbox-<version>
cd <application directory root>
tar –xj jetbrains-toolbox-<version>.tar.gz
chmod –R 777 jetbrains-toolbox-<version>
cd jetbrains-toolbox-<version>
./jetbrains-toolbox
```

Mac

On Mac, we can again use Homebrew for installation. Just execute the following command:

```
brew cask install jetbrains-toolbox
```

Windows

On Windows, we are able to use Chocolatey to install Toolbox. Execute the following command and then launch the app:

```
choco install jetbrainstoolbox
```

Our ultimate goal when installing the Toolbox was to install WebStorm. So, with the Toolbox open, either sign in if you have purchased any JetBrains products or skip the sign in if you just want to try them out. Next, find WebStorm in the products list and click the button to install it. Once the install has finished, you will be able to click a Launch button that will replace the Install button.

Installing the plugins you will need

We've got good news for plugins with WebStorm. WebStorm offers a great plugin community, with every plugin you could possibly want accessible through a plugin management system built into the application. For the purposes of this book, however, you don't actually need any. So, we are done installing plugins! In fact, if anything, WebStorm has too much functionality built in, and we will be ignoring or even turning off some of it so that we can work the way we want to.

Configuring the testing environment

Just as in Visual Studio Code, for WebStorm we are not going to cover setting up any of the built-in test running capability. WebStorm offers a Terminal display that can be turned on and supports having multiple contexts open at the same time.

Create React App

Now that you have Node and NPM installed and up-to-date, turn your attention to the application you want to test drive. Due to its constantly increasing popularity, we are choosing to explain and demonstrate Test-Driven Development by testing a React application.

According to the React website, React is *A JavaScript library for building user interfaces*. We are going to focus on using it for a front-end browser application, but it can be used to create mobile and desktop applications as well.

React was created and is maintained by Facebook. React was created to solve issues that Facebook had in its own user interface, and it is now taking the internet by storm. Facebook has also created a library called Create React App to quickly get a React application going.

What is Create React App?

Create React App is an NPM package created by Facebook for the purpose of providing a zero-configuration way to create a react application. React requires quite a lot to get started, and it can take days to configure a React application manually. Create React App can reduce that time to under a minute.

Installing the global module

Create React App has a global NPM package that must be installed before you can use the command-line utility to actually create a React application. To install the latest version of the Create React App global script, execute the following command in a console or Terminal window:

```
>npm install -g create-react-app
```

Creating a React application

Once the global module is installed, you will be ready to start using Create React App. Creating a React application is extremely streamlined and simple. On my system, I have a directory `\projects` that I use to house all my front-end application projects. Open a console/Terminal window to a similarly purposed directory on your machine and execute the following command to create a new React application:

```
\projects>create-react-app <projectName>
```

In our case, the name of our test case is Speaker Meet, so as an example, my command is displayed as follows:

```
\projects>create-react-app speakermeet-spa
```

SpeakerMeet, as mentioned in the C# section, has both a back-end (the RESTful Web API) and a front-end (a React-based **SPA** (**Single Page Application**)).

Running the Create React App script

When the `create-react-app` script finishes running, a list of available commands is displayed. You want to make sure that everything was successfully created. You can launch the application by executing the following command:

```
>npm start
```

If everything installed correctly, your default browser will open and a new React application will be running.

Mocha and Chai

Create React App supports testing right out of the box. Initially, Create React App uses a testing library named Jest. We want to use Mocha and Chai due to their popularity in the JavaScript community.

Jest

Jest is a testing framework written by Facebook. Just like Create React App, Jest is designed to be a zero-configuration tool. Jest also supports continuous testing and code coverage analysis.

Jest is designed to work within the common **BDD** (**behavior-driven development**) paradigms, as are many other JavaScript testing frameworks. As such, the testing functions `describe` and `it` can both be used to write your tests.

Mocha

Mocha is another JavaScript testing framework and is the one we would like to use. As for library interaction differences, there doesn't appear to be much different in terms of base interactions. The differences come down to the assertion library and the mocking library.

> *Mocking, which will be covered in detail in* Chapter 4, *What to Know Before Getting Started, is essentially a way to provide alternative implementations of objects, classes, and functions, specifically to aid the testing process.*

Mocha itself doesn't come with an assertion library, so one must be provided. The assertion library is what controls test results and how to verify that your code is executing correctly. Most developers who use Mocha rely on Chai for assertions.

As mentioned previously, another consideration is what mocking library you want to use. For many Mocha users, that library is unquestionably *Sinon*.

We will explain the purpose of any and all parts of Mocha that we use in this book. If you want to know more or want the documentation for quick reference while you are developing, you can go to the Mocha home page at https://mochajs.org.

Mocha can be installed into a JavaScript application using the following command:

```
>npm install mocha
```

Chai

Chai is a BDD assertion library. Chai uses a fluent API to allow for extremely flexible assertions. The two most popular ways that Chai is used are through the *should* and *expect* interfaces provided. The way that Chai works, and in fact, the way that every testing frameworks assertion works, is by throwing an exception when the check done by the assertion fails.

For instance, if you had a variable named `foo` with a value *3* and your assertion was `expect(foo).to.equal(5)` when the test ran, that assertion would throw an exception with a message that says `expected 3 to equal 5`.

To install Chai into your project, run the following command:

```
>npm install chai
```

Once you have Chai installed, there is one more step that must be taken to be able to use it within your project. You must include the following import at the top of each test file in your application:

```
import { expect } from 'chai';
```

If you wish to use should assertions, you can either replace `expect` with `should` or add `should` inside the curly braces, separating it from `expect` with a comma.

For more information or to refer to the documentation, the Chai home page is https://chaijs.com.

Sinon

We won't be getting into mocking until Chapter 4, *What to Know Before Getting Started*, but Sinon is the generally preferred mocking library among Mocha + Chai users. Some testing frameworks, such as Jest and Jasmine, come with their own mocking library features, but Mocha does not, and Sinon provides an excellent mocking experience.

To install Sinon into your project, execute the following command:

```
>npm install sinon
```

Once installed, you will need to import Sinon before you can use it. Use the following import statement to enable the use of Sinon:

```
import sinon from 'sinon';
```

Enzyme

Enzyme is a library designed to aid in the testing of React components.

To install Enzyme into your project, execute the following command:

```
>npm install enzyme react-test-renderer react-dom
```

The extra libraries listed, `react-test-renderer` and `react-dom`, are dependencies of Enzyme that it needs to function correctly.

As with the other testing utilities mentioned in this section, we will get into usage as needed, while we discuss the topics covered in this book. But here is a quick example of a test using Enzyme from the Enzyme documentation at https://github.com/airbnb/enzyme:

```javascript
import React from 'react';
import { expect } from 'chai';
import { render } from 'enzyme';
import Foo from './Foo';

describe('<Foo />', () => {
  it('renders three `.foo-bar`s', () => {
    const wrapper = render(<Foo />);

    expect(wrapper.find('.foo-bar').length).to.equal(3);
  });

  it('renders the title', () => {
    const wrapper = render(<Foo title="unique" />);

    expect(wrapper.text()).to.contain('unique');
  });});
```

Ejecting the React app

Unfortunately, this is where we part company with Create React App. In order to use Mocha, we will need to find an alternative way to work with the application. Because of the zero-configuration setup of Create React App, we cannot simply update the testing framework being used, and that is a problem for us.

Thankfully, Create React App gives us an out in the form of ejecting the application. Ejecting the React application will install all the necessary configuration files and utilities into our project and remove Create React App. Once the ejection process is finished, we will have access to all the configuration files and we will have the ability to switch to using Mocha.

To eject Create React App, execute the following command:

```
>npm run eject
```

If you take a look at the `package.json` in the root of the project, you will see that a lot of information and configuration have been added.

> *After any major modification to `package.json`, it is a good idea to delete `node_modules` and `package-lock.json` and then re-run `npm install`.*

Configuring to use Mocha and Chai

After you have ejected the React app, before you make any further modifications, you should make sure everything still works. Execute the following command before making any further modifications:

```
>npm start
```

Now check that a browser launched and that the application is running correctly. You will have to *Ctrl + C* to exit the running process:

```
>npm run build
```

After this command, check that a build folder was created at the root of your project and that there were no errors displayed in the console:

```
>npm test
```

Even though you are about to change the test configuration, you will be using some of the libraries that were provided by Create React App. You want to make sure that those prerequisites transitioned properly when you ejected. As with `npm start`, you will have to *Ctrl + C* to exit this process.

Assuming all of the commands executed without issues, you can now start the process of switching the test environment over to Mocha. Execute the following command to ensure the installation of the necessary dependencies:

```
>npm install mocha chai sinon enzyme
```

Open `package.json` and update the following lines:

```
"babel": {
  "presets": [
    "react-app"
  ]
},
```

Change the preceding code to:

```
"babel": {
  "presets": [
    "react",
    "es2015"
  ]
},
```

You will also need to install the BabelJS preset for ES2015:

```
>npm install babel-preset-es2015
```

Next, find and delete the `jest` setting in `package.json`. You are now ready to change the `test` script to execute Mocha instead of Jest. Find NPM scripts and update the test script as follows:

```
"test": "node scripts/test.js --env=jsdom"
```

Change the preceding code to:

```
"test": "mocha --require babel-core/register ./scripts/test.js --require babel-core/register .
```

The change you just made will cause Mocha to execute all of your tests. It will only execute them once, though. You want a way to have your tests running continuously while you work, so you need to add an additional script. Add a comma to the end of the line you just modified and then add the following script just beneath `test`:

```
"test:watch": "npm test -- -w"
```

Now, you need to update the `test.js` file provided when you ejected. Open `<project root>/scripts/test.js` and replace all the code inside with the following:

```
'use strict';

import jsdom from 'jsdom';

global.document = jsdom.jsdom('<html><body></body></html>');
global.window = document.defaultView;
global.navigator = window.navigator;

function noop() {
  return {};
}

// prevent mocha tests from breaking when trying to require a css file
require.extensions['.css'] = noop;
require.extensions['.svg'] = noop;
```

This file just sets up the base environment for your tests to execute inside. Make note of the `noop` function and usage. Currently, you are ignoring the `css` and `svg` extensions that are required by your production code when you are testing. In the course of testing, if you run into issues while requiring a different extension, you might have to come back to this file and add the troublesome extension to the list.

You are almost done; you only have one more modification to make before you are officially switched over to Mocha. Find the file `App.test.js` in your `src` directory, and change its name to `App.spec.js`, then update the contents to the following:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { expect } from 'chai';

import App from './App';

describe('(Component) App', () => {
  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<App />, div);
  });
});
```

All you have really done here is import Chai and add a `describe` block. The rest of this test has remained unchanged and is the default test provided with Create React App.

A quick kata to check our test setup

For this setup test, you are going to do the Palindrome code kata. This code kata can get complex, but you are only going to concern yourself with the most basic form.

The requirements

The requirements for this kata are as follows:

```
Given a string value
 And Given the provided string is not a palindrome
 When checked
 Then return false

Given a string value
 And Given the provided string is a palindrome
 When checked
 Then return true
```

The execution

As you always should, you will begin with a templated test file designed to verify that the unit testing framework is configured correctly:

```
import { expect } from 'chai';

describe('Test Framework', () =>  {
  it('is configured correctly', () => {
    expect(1).to.equal(0);
  }
}
```

We have shown a failing version of this template. Before you begin writing the tests that are actually for the kata, run the `test:watch` NPM script and verify that the test fails. Also, verify that it fails for the right reason. It should fail because *1* was expected but *0* was the actual result. After the test properly fails, change the zero to a one and verify that the test now passes. As long as these two validations work correctly, we will continue and begin working through the code kata.

Starting the kata

The first thing you will do for the kata is write an `ItExists` test. Again, these types of test help to get the ball rolling and prevent writer's block. You will replace the code for checking the framework with the following.

Red phase; write a failing test that expects an `isPalindrome` function to exist:

```
describe('Is Palindrome', () => {
  it('exists', () => {
    expect(isPalindrome).to.exist;
  });
});
```

Verify that this test fails. It's important to see a failure before moving on to making the test pass. This will help confirm that your test setup is working properly.

And now the *green phase*; make the test pass. Define an `isPalindrome` function and run the test once more to see it pass.

```
function isPalindrome() {
}
```

For the next test, we want to think of the simplest test case that would produce a result. Again, we are skipping bad data issues. The simplest test case we can think of that would product a result is a single letter. For the definition of palindrome that you will be using for these tests, a single letter is a palindrome. Add the following test under the previous one:

```
it('a single letter is a palindrome', () => {
  // arrange
  const value = 'a';

  // act
  const result = isPalindrome(value);

  // assert
  expect(result).to.be.true;
});
```

Now, to make it pass:

```
function isPalindrome() {
  return true;
}
```

Now you have a passing test, but you are always returning true. You want the next test to fail when you write it. So, you should write a test for when the value passed in is not a palindrome. The simplest non-palindrome would be two letters that are not the same:

```
it('two non-matching letters is not a palindrome', () => {
  // arrange
  const value = 'at';
```

```
  // act
  const result = isPalindrome(value);

  // assert
  expect(result).to.be.false;
});
```

Now, make it pass:

```
function isPalindrome(value) {
  if(value.length === 1) {
    return true;
  }

  return false;
}
```

Okay, so now you only return true single letters. This opens us up for our next test, flipping back to something that is a palindrome; write a test for two letters that are the same:

```
it('two matching letters are a palindrome', () => {
  // arrange
  const value = 'oo';

  // act
  const result = isPalindrome(value);

  // assert
  expect(result).to.be.true;
});
```

Now, to make it pass:

```
function isPalindrome(value) {
  if(value.length === 1) {
    return true;
  }

  if(value.length === 2 && value[0] === value[1]) {
    return true;
  }

  return false;
}
```

The next test is to have a three-letter word that is a palindrome. Currently, this should fail:

```
it('three letter palindrome', () => {
  // arrange
  const value = 'mom';

  // act
  const result = isPalindrome(value);

  // assert
  expect(result).to.be.true;
});
```

To make this test pass, think about what you have so far. One algorithm for checking a palindrome is to simply start on the outsides and check the two outermost letters. If those two letters are a match, then move in one letter on each side. Repeat this check until you get to the center of the word or phrase. If the center is one letter, then it's a palindrome; otherwise, check if the two center-most characters are a match. Let's try this concept out by using recursion to make the latest test pass:

```
function isPalindrome(value) {
  if(value.length === 1) {
    return true;
  }

  if(value.length === 2 && value[0] === value[1]) {
    return true;
  }

  if(value[0] === value[value.length -1]) {
    return isPalindrome(value.substring(1, value.length - 1));
  }

  return false;
}
```

We now need to check if a four-letter palindrome will pass:

```
it('four letter palindrome', () => {
  // arrange
  const value = 'abba';

  // act
  const result = isPalindrome(value);

  // assert
  expect(result).to.be.true;
});
```

It passes; excellent! We will end this code kata with two exercises for you. The first exercise is to add a test for "a man a plan a canal panama" and make it pass. The second exercise is to refactor the code for isPalindrome. While this is a small function, it could still do with some tidying up, and potentially some optimizations.

Summary

You should now have Node installed and your JavaScript development environment configured. JavaScript examples throughout the rest of the book will assume your use of WebStorm.

But, before diving right in, Chapter 4, *What to Know Before Getting Started,* will focus on what more you need to know before getting started.

What to Know Before Getting Started

You're off to a pretty good start. By now, you should be starting to feel comfortable with the basic concepts behind Test-Driven Development. You know the basic premise behind TDD and how to write a unit test in C# and JavaScript.

In this chapter:

- We'll cover more of the practices behind TDD
- Specific advice will be given on how to avoid pitfalls along the way
- We'll explain the importance of defining and testing boundaries, abstracting away third-party code (including the .NET Framework)
- We'll begin to introduce more advanced concepts, such as spies, mocks, and fakes

First, let's cover some issues you may run into while trying to test an existing application. Hopefully, this will help you avoid problems in your next green-field application.

Untestable code

There are a variety of telltale signs that an application, class, or method will be difficult, or even impossible, to test. Sure, there are ways around some of the following examples but it's usually best to just avoid workarounds and programmatic acrobatics. Simple is usually best, and your future self and/or future maintainers will thank you for keeping things simple.

Dependency Injection

If you're creating instances of external resources within your constructors or inside methods instead of having them passed in, it will be very difficult to write tests to cover these classes and methods. Generally, in today's modern applications, Dependency Injection frameworks are used to create and provide the external dependencies to a class. Many choose to define an *interface* as the contract for the dependency, providing a more flexible method for testing and the coupling to external resources.

Static

You may have a need to access static third-party classes or methods. Instead of accessing static resources directly, <mark>it would be better to access these through an *interface*</mark>. In the example of `DateTime` in C#, `Now` is a static property, which prevents you from being able to control the `DateTime` value used by the class or method being tested. This makes it more difficult to verify your test cases and ensure your program's logic is behaving correctly, <mark>based on specific dates or times.</mark>

Singleton

Singletons are the essence of the shared state. In order to ensure your tests run in an isolated environment, it would be best to avoid them. If a *singleton* is required (for example, `Logging`, `Data Context`, and so on), most Dependency Injection frameworks allow for the substitution of a non-singleton class as a single instance, which gives the functionality and flexibility of effectively having a singleton. For production code, this allows you to control the scope of the singleton instance.

Global state

It has long been understood that global state within an application will wreak havoc on a system and cause unexpected behavior that is difficult to trace. Changing the code in one place will possibly have far-reaching side-effects on the rest of your system. For testability, this often means much more effort in <mark>setup</mark> and <mark>slower test execution.</mark>

Abstracting third-party software

As your application grows, you'll likely introduce external dependencies. Assuredly, the developers of these systems, applications, and libraries have thoroughly tested their offerings. You should focus your attention on testing your application, not on testing third-party code. Your application should be robust enough to handle edge cases, and you'll want to account for expected and unexpected behavior. You'll want to abstract away the details of the third-party code and test for expected (and unexpected) results.

So, what is *third-party code*? Anything you didn't write. That includes the .NET Framework itself. One way to achieve the abstraction of third-party code is with the use of test doubles.

Test doubles

Test doubles are functions and classes that aid in the testing process by allowing you to either verify functionality or bypass a dependency that would otherwise be difficult to test. Test doubles are used at all levels to isolate the code being tested. Many times, the need for a test double drives the architecture of the code.

The `DateTime` object in C# is an example of when this is the case. `System.DateTime` is part of the .NET Framework, and normally you wouldn't think that you would abstract this in your code. The instinct of most developers is to simply reference it in a *using statement* and then access `DateTime.Now` directly within their code.

A test that can't be repeated is a bad test.

This is usually difficult to test. If we were to try to test a method using `DateTime.Now`, we would be unable to prevent `DateTime.Now` default functionality. `DateTime.Now` returns the current date and time stored in a `DateTime` object. Not having the ability to manipulate the return of this object causes our tests to be unpredictable and unrepeatable. A test that can't be repeated is a bad test.

Many developers already understand the need for predictability. You may have heard the phrase, *If it can't be reproduced, it's not a bug or a similar sentiment*. This is because, in order to verify that we have fixed a bug, we must be able to predictably repeat the error. This way, once the steps to reproduce it no longer produce the bug, we can confidently say that the bug is fixed. At least we can for that series of steps.

Testing is no different from bug fixing; it follows all the same steps. We just know exactly what caused the bug; the code hasn't been written yet, or the refactoring we just attempted failed.

Creating test doubles can get a little involved at times. For this reason, frameworks to support the creation of these test doubles have been created for nearly every language that has a testing framework. The frameworks are generally referred to as mocking frameworks or libraries. In C#, the predominant framework currently in use is *Moq,* pronounced *mock*. Similarly, in JavaScript, the most referenced mocking library seems to be *Sinon*, pronounced *sign on.*

Mocking frameworks

Mocking frameworks are a great utility to alleviate some of the pressure of testing in a large project. They are especially useful when trying to wrap tests around a legacy system. A legacy system, in this case, is defined as an application that does not already have tests around it. This definition is from Michael Feather's book, *Working Effectively with Legacy Code*.

Use caution while learning Test-Driven Development and using mocking frameworks. Mocking frameworks provide a very attractive alternative to carefully considering your code. It is possible to write a complete set of tests that, in the end, only really test the mocking framework.

Many mocking frameworks are overpowered in this respect. In C#, a classification of mocking frameworks exists that allows you to replace external code. This external code includes `DateTime.Now` and any other class that you don't control. In JavaScript, this is called monkey patching, and every framework allows you to do it.

What's the harm, you ask? One of the benefits of TDD is that it encourages smart architectural choices. When you have the power to override the functionality of the third-party code, you no longer have the need to abstract in order to test.

Why is that a problem? Abstraction of the third-party is necessary if we want to keep the code flexible and if we want to follow the SOLID principles.

The SOLID principles

The SOLID principles are a collection of concepts originally put together by Robert C. Martin, aka "Uncle Bob." Usually advertised as *Object-Oriented Principles*, you should think of them as just plain good architectural choices. The SOLID principles consist of five principles: the <mark>Single Responsibility Principle</mark>, the <mark>Open/Closed</mark> principle, the <mark>Liskov Substitution</mark> principle, the <mark>Interface Segregation</mark> Principle, and the <mark>Dependency Inversion</mark> Principle.

The original articles on the SOLID principles are available at http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

The Single Responsibility Principle

In Uncle Bob's article, the **Single Responsibility Principle** (**SRP**) is defined as, "<mark>A class should have only one reason to change.</mark>"

What does this mean? That is the tough part; there are many approaches to understanding the meaning. One way to look at it is that the class should only support one business user. Another is that, within the application, the class should only be used with a limited or specific scope. Another is that the class should have a limited range of functionality. These are all correct, and yet insufficient. One way to ensure you are following it is to use what we will refer to as the "Rule of Three to Five."

If we're discussing requirements, for example, when a requirement has between three to five acceptance criteria, then it is most likely appropriately sized for its level of detail. Similarly, if we are discussing a method or function, then three to five lines of code is probably appropriately sized.

The Rule of Three to Five is a generic way to know that you are honoring the SRP. The rule states, "Less than three is good. Between three and five is fine. Above five is strongly consider refactoring." It's not quite as elegant as many other laws, principles, and rules, but the rule of three to five is easy to follow. This rule is just a guideline and should not be used as an ultimatum. You should try to apply this rule to just about everything in software development. You have already seen it in action in this book. This rule was used to determine the scope of the requirements in Chapter 1, *Why TDD is Important*, and in all the code samples that have been included so far.

If you use the Rule of Three to Five, it nearly guarantees that you are following SRP, and it keeps your code, file structure, and requirements small and maintainable.

The Open/Closed principle

The Open/Closed principle states, "Software entities (classes, modules, functions, and so on) should be open for extension, but closed for modification." The second of the SOLID principles doesn't sound like it is saying much, but it has a large impact.

There are many ways to honor this principle. You or your development team could put into place a rule that only allows for new development. That is, any existing functionality cannot be updated or changed, only replaced by new methods or classes. When we get to creating dividing lines in the code, you could use those dividing lines to create a place for this functionality swap to take place.

The Open/Closed principle also enables continuous integration and deployment. This is because, if your application never breaks a contract it has with the user, itself, or a third party, then it can always be deployed without fear of causing a production issue.

The Liskov Substitution principle

The Liskov Substitution principle may be difficult to understand at first due to its somewhat complex and mathematical definition. From Barbara Liskov's *Data Abstraction and Hierarchy* [https://pdfs.semanticscholar.org/36be/babeb72287ad9490e1ebab84e7225ad6a9e5.pdf], the principle is stated as follows:

What is wanted here is something like the following substitution property. If for each object o1 of type S there is an object o2 of type T, such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Uncle Bob has simplified this definition to be, *Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.* Looking at this principle, it seems like it would just be inheritance. Except, it's not just inheritance. This principle implies that not only does the object replacing the other have to implement the same interface or contract as the original, it must also adhere to the same expectations as the original.

The classic example of a violation of this principle is the use of a square class in the place of a rectangle class. A typical rectangle class would need to have both length and width properties. In mathematics, a square is just a special type of rectangle. So, many would assume that creating a square class with length and width would be an acceptable swap for the rectangle class.

The problem here is that a square requires that both length and width have the same value. So, when you change either one on the square class, the class will update the other to have the same value. This is a problem because the application using the object doesn't expect this behavior. Therefore, the application must be aware of the possibility of the length or width changing without notice.

A failure to meet the expectations of the application is known as a refused bequest. A refused bequest can cause inconsistent behavior in an application and, at the very least, requires more code to compensate for the mismatch.

The Interface Segregation principle

The Interface Segregation principle is about keeping to the contract of interaction presented by your class small. More than small, the contract presented by your class should have a single responsibility.

Sometimes, having a class with a small single responsibility contract is difficult or not desired. In those instances, the class should implement multiple contracts instead of creating a combined contract. We want multiple contracts to reduce the number of far-reaching dependencies.

Every time a base class or interface is modified, the child classes must also be modified. At the very least, the child classes must now be recompiled. By limiting the scope of a contract, we can reduce the impact of changing that contract and improve the overall system architecture.

The Dependency Inversion principle

Inversion of dependencies is important for several reasons, among which are that inverted dependencies increase flexibility, decrease fragility, and help the code to be potentially reused.

Inversion of dependencies allows for a plugin type architecture. By defining a contract of interaction, a module can determine how it wants to interact with dependencies. Then the dependencies depend on that contract.

Because the top-level module has no outgoing dependencies, it can be deployed independently. Deploying a piece of an application independently almost never happens, but having an independently deployable library has the tremendous benefit of not needing to be recompiled when a dependency changes.

In normal development, the dependencies fluctuate a lot more than the higher-level modules. This fluctuation causes the need to recompile. When your application dependencies flow downward, a dependency recompile also triggers a recompile of the dependent library. So, in effect, changing a utility helper class in a tiny, but common, library will trigger a recompile of your entire application.

If you are inverting your dependencies, however, a change like this will only trigger a recompile of the utility helper library and the application library. It will not trigger a recompile of every library between.

That does it for the SOLID principles. Please keep them in mind if you choose to use a mocking framework. Make sure you don't allow the mocking framework to trick you into building a rigid, fragile, immobile system.

Timely greeting

Expanding on the classic *Hello World* example, what if you wanted to change your greeting based on the time of day? An example is as follows:

```
As a visitor to the site
 I want to receive a time-appropriate greeting
 So that I may plan the submission of my talks

Given it is before noon
 When greeting is requested
 Then morning message is returned

Given it is afternoon
 When greeting is requested
 Then afternoon message is returned
```

You might think to yourself, This is simple; I can just write a quick method to return the proper message. Of course, you would be right. This is a pretty easy task. You might come up with something like this:

```
public string GetGreeting()
{
  if (DateTime.Now.Hour < 12)
    return "Good morning";

  return "Good afternoon";
}
```

Remember, back in Chapter 1, *Why TDD is Important*, we discussed the Three laws of TDD. The all-important first law states that you aren't allowed to write a single line of production code without a failing test.

Fragile tests

"But, this is such a simple method," you might say. What if you encountered a bug? What if you wanted to write some tests for this method after the fact? Would you have to run your test suite at a specific time of day to ensure a passing test? Would you have to alter your tests based on the time of day that you ran them?

False positives and false failures

If we left the code in our *Message* example as-is and wrote a test to cover the method, it might look something like this:

```
[Fact]
public void GivenEvening_ThenAfternoonMessage()
{
  // Arrange
  // Act
  var message = GetGreeting();

  // Assert
  Assert.Equal("Good afternoon", message);
}
```

Can you spot the problem with this test? There's nothing inherently wrong with the test, per se. The problem is that the production code will return a different message, based on the time of day. This means that if you ran the test in the afternoon, it would pass. If you ran the test in the morning, it would fail.

Abstract DateTime

`DateTime` is part of the .NET Framework, and therefore, it should be abstracted away from our system. Typically, we want our system to depend on interfaces, allowing us to substitute implementations at runtime.

Following is an example of `ITimeManager`:

```
public interface ITimeManager
{
  DateTime Now { get; }
}
```

For testing purposes, you might end up with an implementation of `ITimeManager` that looks like this:

```
public class TestTimeManager : ITimeManager
{
  public Func<DateTime> CurrentTime = () => DateTime.Now;

  public void SetDateTime(DateTime now)
  {
    CurrentTime = () => now;
  }

  public DateTime Now => CurrentTime();
}
```

This allows us to set the value for `Now` so that we can supply a known value to our test methods. Now, let's revisit our tests:

```
[Theory]
[InlineData(12)]
[InlineData(13)]
[InlineData(14)]
[InlineData(15)]
[InlineData(16)]
[InlineData(17)]
[InlineData(18)]
[InlineData(19)]
[InlineData(20)]
[InlineData(21)]
[InlineData(22)]
[InlineData(23)]
public void GivenAfternoon_ThenAfternoonMessage(int hour)
{
  // Arrange
  var afternoonTime = new TestTimeManager();
  afternoonTime.SetDateTime(new DateTime(2017, 7, 13, hour, 0, 0));
  var messageUtility = new MessageUtility(afternoonTime);

  // Act
  var message = messageUtility.GetGreeting();

  // Assert
  Assert.Equal("Good afternoon", message);
}

[Theory]
[InlineData(0)]
```

```
[InlineData(1)]
[InlineData(2)]
[InlineData(3)]
[InlineData(4)]
[InlineData(5)]
[InlineData(6)]
[InlineData(7)]
[InlineData(8)]
[InlineData(9)]
[InlineData(10)]
[InlineData(11)]
public void GivenMorning_ThenMorningMessage(int hour)
{
  // Arrange
  var morningTime = new TestTimeManager();
  morningTime.SetDateTime(new DateTime(2017, 7, 13, hour, 0, 0));
  var messageUtility = new MessageUtility(morningTime);

  // Act
  var message = messageUtility.GetGreeting();

  // Assert
  Assert.Equal("Good morning", message);
}
```

Our production code would end up looking something like this:

```
public class MessageUtility
{
  private readonly ITimeManager _timeManager;

  public MessageUtility(ITimeManager timeManager)
  {
    _timeManager = timeManager;
  }

  public string GetMessage()
  {
    if (_timeManager.Now.Hour < 12)
      return "Good morning";

    return "Good afternoon";
  }
}
```

Test double types

Test doubles come in many varieties. Those varieties can generally be grouped as dummies, stubs, spies, mocks, and fakes. Coming up, we will discuss the different types and provide examples in C# and in JavaScript for each.

Dummies

Dummies are the simplest form of test double. A dummy has no appreciable functionality. We don't actually expect the `dummy` class or method to be used in the result of the class or method we are testing.

Dummies are most often used when the class you are testing has a dependency that the method or function you are testing does not use.

You create a dummy by creating a new copy or instance of a class or method and then doing absolutely nothing in the body of the code. Void methods will be empty and methods or functions expecting a return value will either throw when called or return the simplest form of that return value.

Dummy logger

A *Logging* service is a perfect example of something that can be replaced with a dummy. While you are testing specific methods it is unlikely (and not recommended) to also test logging functionality.

Example in C#

The following is an example of a `DummyLogger` in C#. You'll note that when `Log` is called nothing happens.

```
enum LogLevel
{
  None = 0,
  Error = 1,
  Warning = 2,
  Success = 3,
  Info = 4
}

interface ILogger
{
  void Log(LogLevel type, string message);
}

class DummyLogger: ILogger
{
  public void Log(LogLevel type, string message)
  {
    // Do Nothing
  }
}
```

Example in JavaScript

The following is an example of a `DummyLogger` in JavaScript. You'll note that when `info`, `warn`, `error`, and `success` are called nothing happens.

```javascript
export class DummyLogger {
    info(message) {
    }

    warn(message) {
    }

    error(message) {
    }

    success(message) {
    }
}
```

Stubs

Stubs are the next level up from dummies. A Stub test double will provide the same response regardless of the parameters passed into it.

Stubs are used when you want to test different paths of execution in your code. One instance is an error that must be thrown under a particular condition.

Stubs are created by creating a copy or override of the class or method that needs to return the stub value and then setting it to return the needed value. Remember, stubs don't evaluate parameters, so you need to just return the desired value.

Example in C#

The following is an example of a `StubSpeakerContactServiceError` in C#. You'll note that, when `MessageSpeaker` is called then a new `UnableToContactSpeakerException` error is thrown.

```csharp
class StubSpeakerContactServiceError : ISpeakerContactService
{
  public void MessageSpeaker(string message)
  {
    throw new UnableToContactSpeakerException();
  }
}
```

Example in JavaScript

The following is an example of a `stubSpeakerReducer` in JavaScript. You'll note that regardless of the action passed in, a new *UNABLE_TO_RETRIEVE_SPEAKERS* error is pushed to the error array in the state.

```javascript
import { SpeakerErrors } from './errors';
import { SpeakerFilters } from './actions';

const initialState = {
  speakerFilter: SpeakerActions.SHOW_ALL,
  speakers: [],
  errors: []
};

export function stubSpeakerReducer(state, action) {
  state = state || initialState;

  state.speakerFilter = action.filter || SpeakerFilters.SHOW_ALL;
  state.errors.push(SpeakerErrors.UNABLE_TO_RETRIEVE_SPEAKERS);

  return state;
}
```

Spies

Spies are the next evolution in test doubles. A spy returns a value similar to a stub but has an extremely important and helpful difference. Spies can report back on the information related to the function call.

Spies are most often used when you want to verify that a function was called with specific parameters. This is most useful at third-party boundaries in your application. For instance, it is important to know whether your application is correctly configuring a database connection using the credentials supplied by some configuration service. Also, in some cases, it is difficult to measure the side-effects of the method or function being tested. In those cases, you can use a spy to just make sure you are calling the method or function in the first place.

Spies are created by starting with a stub and adding the functionality to determine whether a function has been called, how many times a function is called, or reporting what values were passed into that function.

Example in C#

The following is an example of a `SpySpeakerContactService` in C#. The `SpySpeakerContactService` allows you to determine if the service has been called and how many times it might have been called.

```csharp
class SpySpeakerContactService : ISpeakerContactService
{
  public bool MessageSpeakerHasBeenCalled { get; private set; }

  public int MessageSpeakerCallCount { get; private set; }

  public void MessageSpeaker(string message)
  {
    MessageSpeakerHasBeenCalled = true;
    MessageSpeakerCallCount++;
  }
}
```

Example in JavaScript

The following is an example of a `spySpeakerReducer` in JavaScript. The `spySpeakerReducer` allows you to determine how many times it might have been called.

```javascript
import { speakerReducer as original_speakerReducer } from './reducers';

export let callCounter = 0;

export function spySpeakerReducer(state, action) {
  callCounter++;

  return original_speakerReducer(state,action);
}
```

Mocks

Mocks are essentially programmable spies. Mocks are useful when you want to use the same test double in multiple tests. Mocks have the ability to return whatever values you set them to return. It is important to note that mocks are still not doing any logic. They return the value that is specified and do not check the parameters passed to the function.

Mocks are used in all the situations where dummies, stubs, and spies are used. Mocks are a heavier implementation of a test double, which is why you may not want to use them all the time. Mocks get less reuse than the previous test doubles because a mock's data must be set for each test, whereas a dummy, stub, or spy has a set return value that does not need to be configured. Setting up the test data that gets returned is often more difficult than simply creating a whole stub or spy class.

Mocks are created by making a copy of a class or method and creating a property that can be set as the return value for a method; then, in the method being mocked, the property value is returned. Once created, before each test, the mock's return value must be set.

Example in C#

The following is an example of a `MockDateTimeService` in C#. The `MockDateTimeService` allows you to set the `DateTime` to be returned by the service in order to reliably test how other parts of the system might behave based on specific `DateTime`.

```csharp
class MockDateTimeService
{
  public DateTime CurrentDateTime { get; set; } = new DateTime();

  public DateTime UTCNow()
  {
    return CurrentDateTime.ToUniversalTime();
  }
}
```

Example in JavaScript

The following is an example of a `MockDateTimeService` in JavaScript. Much like the `MockDateTimeService` in C#, this allows you to set the `DateTime` to be returned by the service in order to reliably test how other parts of the system might behave based on specific `DateTimes`.

```javascript
export class MockDateTimeService {
  constructor() {
    this.currentDateTime = new Date(2000, 0, 1);
  }

  now() {
    return this.currentDateTime;
  }
}
```

Fakes

Fakes are the last and most powerful type of test double. A fake is a class that attempts to behave as if it weren't a test double. While a fake will not connect with a database, it will attempt to behave just like it is connecting to a database. A fake will not use the system clock, but it will attempt to have an internal clock that behaves as close to the system clock as possible.

Fakes either add extra testing functionality or prevent external interference from third-party libraries and systems. Most applications are connected to some data source. A fake repository can be created that uses its own in-memory data source but otherwise behaves just like a normal data connection.

Fakes are created by generating a whole new class or method and then writing enough functionality to be indistinguishable from the production class or method. The only important distinction for a fake versus a production class or method is that the fake does not make external connections and likely has the ability for the tester to control the underlying data set.

Example in C#

The following is an example of a `FakeRepository` and associated *interfaces*. The `FakeRepository` is a fake implementation of a generic repository.

```csharp
public interface IRepository<T>
{
  T Get(Func<T, bool> predicate);
  IQueryable<T> GetAll();
  T Save(T item);
  IRepository<T> Include(Expression<Func<T, object>> path);
}

public interface IIdentity
{
  int Id {get;set;}
}

public class FakeRepository<T> : IRepository<T> where T : IIdentity
{
  private int _identityCounter = 0;
  public IList<T> DataSet { get; set; } = new List<T>();

  public T Get(Func<T, bool> predicate)
  {
    return GetAll().Where(predicate).FirstOrDefault();
  }

  public IQueryable<T> GetAll()
  {
    return DataSet.AsQueryable();
  }

  public T Save(T item)
  {
    return item.Id == default(int) ? Create(item) : Update(item);
  }

  public IRepository<T> Include(Expression<Func<T, object>> path)
  {
    // Nothing to do here since this function is for EntityFramework
    // We are using Linq to Objects so there is not need for Include
    return this;
  }

  private T Create(T item)
  {
    item.Id = ++_identityCounter;
    DataSet.Add(item);
    return item;
  }

  private T Update(T item)
  {
    var found = Get(x => x.Id == item.Id);

    if(found == null)
    {
      throw new KeyNotFoundException($"Item with Id {item.Id} not
        found!");
    }

    DataSet.Remove(found);
    DataSet.Add(item);
```

```
    return item;
  }
}
```

Example in JavaScript

The following is an example of a FakeDataContext in JavaScript.

```javascript
export class FakeDataContext {
  _identityCounter = 1;
  _dataSet = [];

  get DataSet() {
    return this._dataSet;
  }

  set DataSet(value) {
    this._dataSet = value;
  }

  get(predicate) {
    if (typeof(predicate) !== 'function') {
      throw new Error('Predicate must be a function');
    }

    const resultSet = this_dataSet.filter(predicate);

    return resultSet.length >= 1 ? {...resultSet[0]} : null;
  }

  getAll() {
    return this._dataSet.map((x) => {
      return {...x};
    });
  }

  save(item) {
    return item.id ? this.update(item) : this.create(item);
  }

  update(item) {
    if (!this._dataSet.some(x => x.id === item.id)) {
      this._dataSet.push({...item});
    } else {
      let itemIndex = this._dataSet.findIndex(x => x.id === item.id);
      this._dataSet[itemIndex] = {...item};
    }

    return {...item};
  }

  create(item) {
    let newItem = {...item};
    newItem.id = this._identityCounter++;
    this._dataSet.push({...newItem});

    return {...newItem};
  }
}
```

N-Tiered example

Now, turn your attention back to the API controller in Chapter 2, *Setting Up the .NET Test Environment*. Hard-coded data being returned directly from the controller does not make for a solid foundation on which to build an application. Most modern .NET applications of any size are written in some sort of N-tiered architecture. You'll want to separate your business logic from your presentation, in this instance, the presentation in the API endpoint.

We'll introduce an *interface* for a speaker service in preparation for using Dependency Injection to provide the concrete implementation to the controller, then verify that the proper method in the new service is being called. You'll need to rearrange some tests in order to remove the business logic from the controller.

Presentation layer

To get started, add a new test to verify that the controller accepts an *interface* of `ISpeakerService`:

```
[Fact]
public void ItAcceptsInterface()
{
  // Arrange
  ISpeakerService testSpeakerService = new TestSpeakerService();

  // Act
  var controller = new SpeakerController(testSpeakerService);

  // Assert
  Assert.NotNull(controller);
}
```

Now, make your test pass by creating a constructor in the `SpeakerController` to accept the `ISpeakerService` interface, introducing a field variable and a constructor in your `speaker controller` class:

```
public SpeakerController(ISpeakerService speakerService)
{
}
```

Your test project should now fail to compile. This is because in our previous example from , *Setting up the .NET Test Environment*, we're defining the controller instance in the constructor of the test class. Modify the constructor to create an instance of `TestSpeakerService`, which implements the `ISpeakerService` interface, and pass this to the `SpeakerController`. Feel free to create the `TestSpeakerService` in your test class:

```
public SpeakerControllerSearchTests()
{
  var testSpeakerService = new TestSpeakerService();

  _controller = new SpeakerController(testSpeakerService);
}
```

Now, you'll want to verify that the `Search` method of the `SpeakerService` is called from the controller. But, how do you do that? One way is to use a mocking framework called *Moq*.

Moq

To add *Moq* to your unit test project, right-click on your test project and choose Manage NuGet Packages. Browse for *Moq*, and choose to install the latest stable version. We won't delve too deeply into *Moq*, but we will show how mocking frameworks help facilitate testing the boundaries of your application.

Add a test to verify that the Search method of the SpeakerService is called once from the Search action result of the controller:

```
[Fact]
public void ItCallsSearchServiceOnce()
{
  // Arrange
  // Act
  _controller.Search("jos");

  // Assert
  _speakerServiceMock.Verify(mock => mock.Search(It.IsAny<string>()),
      Times.Once());
}
```

In order to make the test pass, you will also be required to do a little more setup in the constructor of the test class:

```
private readonly SpeakerController _controller;
private static Mock<ISpeakerService> _speakerServiceMock;

public SpeakerControllerSearchTests()
{
  var speaker = new Speaker
  {
    Name = "test"
  };

  // define the mock
  _speakerServiceMock = new Mock<ISpeakerService>();

  // when search is called, return list of speakers containing speaker
  _speakerServiceMock.Setup(x => x.Search(It.IsAny<string>()))
      .Returns(() => new List<Speaker> { speaker });

  // pass mock object as ISpeakerService
  _controller = new SpeakerController(_speakerServiceMock.Object);
}
```

Be sure to modify the *interface* so that the application will compile:

```
public interface ISpeakerService
{
  IEnumerable<Speaker> Search(string searchString);
}
```

Now, make your test pass by ensuring the Search method of the SpeakerService is called from the Search action result of the controller. If you haven't done so already, create a field variable for _speakerService that is assigned in the constructor by the speakerService parameter:

```
  private readonly ISpeakerService _speakerService;

  public SpeakerController(ISpeakerService speakerService)
  {
    _speakerService = speakerService;
  }

  [Route("search")]
  public IActionResult Search(string searchString)
  {
    var hardCodedSpeakers = new List<Speaker>
    {
      new Speaker{Name = "Josh"},
      new Speaker{Name = "Joshua"},
      new Speaker{Name = "Joseph"},
      new Speaker{Name = "Bill"},
    };

    _speakerService.Search("foo");

    var speakers = hardCodedSpeakers.Where(x =>
                       x.Name.StartsWith(searchString,
                       StringComparison.OrdinalIgnoreCase)).ToList();

    return Ok(speakers);
  }
}
```

Next, add a test to validate that the `searchString` supplied to the `Search` action result of the controller is the `searchString` being passed to the `Search` method of the `SpeakerService`:

```
[Fact]
public void GivenSearchStringThenSpeakerServiceSearchCalledWithString(){
  // Arrange
  var searchString = "jos";

  // Act
  _controller.Search(searchString);

  // Assert
  _speakerServiceMock.Verify(mock => mock.Search(searchString),
      Times.Once());
}
```

And make the test pass by supplying `searchString` to the `Search` method on the `_speakerService`:

```
  _speakerService.Search(searchString);
```

Now, ensure that the results of the `search` method from the `SpeakerService` are what is being returned by the action result:

```
[Fact]
public void GivenSpeakerServiceThenResultsReturned()
{
  // Arrange
  var searchString = "jos";

  // Act
  var result = _controller.Search(searchString) as OkObjectResult;

  // Assert
  Assert.NotNull(result);
  var speakers = ((IEnumerable<Speaker>)result.Value).ToList();
  Assert.Equal(_speakers, speakers);
}
```

Remember, ==the results returned by the `search` method of the `SpeakerService` are being defined by the `Mock`==. You'll need to extract a `field` in order to test that the results being returned by the *action*

result are the same as those being defined for our `Mock`:

```
private readonly SpeakerController _controller;
private static Mock<ISpeakerService> _speakerServiceMock;
private readonly List<Speaker> _speakers;

public SpeakerControllerSearchTests()
{
  _speakers = new List<Speaker> { new Speaker
  {
    Name = "test"
  } };

  _speakerServiceMock = new Mock<ISpeakerService>();
  _speakerServiceMock.Setup(x => x.Search(It.IsAny<string>()))
                 .Returns(() => _speakers);

  _controller = new SpeakerController(_speakerServiceMock.Object);
}
```

There's still the problem of the hard-coded data. Don't forget to remove unnecessary and unneeded code while you're making your test pass. Remember *red, green, refactor*. This applies to your production code as well as your tests.

You may encounter some failing tests once you remove the hard-coded data. For now, skip these tests, as we'll be moving this logic to another part of the application. Now it's time to create a `SpeakerService`:

```
xUnit
 [Fact(Skip="Reason for skipping")]
MSTest
 [Skip]
```

Business layer

You might want to start thinking about how to organize your tests effectively. As your application grows, and the number of test files increases, <mark>you may find it more and more cumbersome to navigate your solution</mark>. One answer might be to create individual folders per class under test and a single file per public method within the class folder. This might look something like this:

```
SpeakerService -> Search
```

You don't necessarily need to tackle this now, but it wouldn't hurt to have a plan for the future. Applications tend to grow quite quickly, and before you know it you will have thirteen projects within your solution. You may choose to go ahead and create a `Services` project with a `ServicesTest` project at this time, to separate your business layer and associated tests from your presentation layer and its tests. That will be left as an exercise for the reader.

Now, create a new test class for the `SpeakerService`. Here is where you'll be creating all of your test methods for `Search` in the `SpeakerService`:

```
[Fact]
public void ItExists()
{
  var speakerService = new SpeakerService();
}
```

Once you make this test pass, create a few new tests to confirm the `Search` method exists and that it returns a collection of speakers:

```
[Fact]
public void ItHasSearchMethod()
{
  var speakerService = new SpeakerService();

  speakerService.Search("test");
}
```

Next, test that the `SpeakerService` implements the `ISpeakerService` interface:

```
[Fact]
public void ItImplementsISpeakerService()
{
  var speakerService = new SpeakerService();

  Assert.True(speakerService is ISpeakerService);
}
```

Your `SpeakerService` should now look something like this:

```
public class SpeakerService : ISpeakerService
{
  public IEnumerable<Speaker> Search(string searchString)
  {
    return new List<Speaker>();
  }
}
```

Remember, take slow and methodical steps. You are not allowed to write a line of production code without writing a failing test, and you're not to write more production code than is sufficient to make the tests pass.

Now, begin to move the *skipped* tests from the `controller test` file to the `Speaker Service Search Test` file. Start with `GivenExactMatchThenOneSpeakerInCollection`:

```
[Fact]
public void GivenExactMatchThenOneSpeakerInCollection()
{
  // Arrange
  // Act
  var result = _speakerService.Search("Joshua");

  // Assert
  var speakers = result.ToList();
  Assert.Equal(1, speakers.Count);
  Assert.Equal("Joshua", speakers[0].Name);
}
```

Make this test pass, then move on to `GivenCaseInsensitveMatchThenSpeakerInCollection`:

```
[Theory]
[InlineData("Joshua")]
[InlineData("joshua")]
[InlineData("JoShUa")]
public void GivenCaseInsensitveMatchThenSpeakerInCollection(string searchString)
{
  // Arrange
  // Act
  var result = _speakerService.Search(searchString);

  // Assert
  var speakers = result.ToList();
  Assert.Equal(1, speakers.Count);
  Assert.Equal("Joshua", speakers[0].Name);
}
```

And finally, `GivenNoMatchThenEmptyCollection` and `Given3MatchThenCollectionWith3Speakers`:

```
[Fact]
public void GivenNoMatchThenEmptyCollection()
{
  // Arrange
  // Act
  var result = _speakerService.Search("ZZZ");

  // Assert
  var speakers = result.ToList();
  Assert.Equal(0, speakers.Count);
}

[Fact]
public void Given3MatchThenCollectionWith3Speakers()
{
  // Arrange
  // Act
  var result = _speakerService.Search("jos");

  // Assert
  var speakers = result.ToList();
  Assert.Equal(3, speakers.Count);
  Assert.True(speakers.Any(s => s.Name == "Josh"));
  Assert.True(speakers.Any(s => s.Name == "Joshua"));
  Assert.True(speakers.Any(s => s.Name == "Joseph"));
}
```

As you get more comfortable with the practice and gain more experience with TDD, you may find it helpful to list the tests which you want to implement. This could be simply jotting them down on a piece of paper, or stubbing out some skipped or ignored tests in your IDE.

If done correctly, your code should look something like this:

```
public class SpeakerService : ISpeakerService
{
  public IEnumerable<Speaker> Search(string searchString)
  {
    var hardCodedSpeakers = new List<Speaker>
    {
      new Speaker{Name = "Josh"},
      new Speaker{Name = "Joshua"},
      new Speaker{Name = "Joseph"},
      new Speaker{Name = "Bill"},
    };

    var speakers = hardCodedSpeakers.Where(x =>
                          x.Name.StartsWith(searchString,
                          StringComparison.OrdinalIgnoreCase)).ToList();

    return speakers;
  }
}
```

We've now moved the hard-coded data out of our controller and into our business layer in the SpeakerService. You may think that a lot of effort was expended simply to move the problem into a new file! While this is true to an extent, this actually puts us in a better place for future development. The "logic", such as it is, has been moved into a class that can be reused by other parts of the application, and by potential new interfaces (think native and/or mobile applications) that would not have access to our original controller.

We'll continue with this example in future chapters. We will *finally* rid ourselves of hard-coded data and implement a data access layer using the Entity framework. All of this can be accomplished with Test-Driven Development.

Summary

In this chapter, we covered some pitfalls that will hinder TDD, such as dependence on third-party libraries, direct instantiation of classes, and fragile tests. We also discussed ways to avoid or work around those issues. We introduced and discussed each of the SOLID principles. We also discussed the different types of test double and when each type is appropriate. Lastly, we gave a short example of an N-tiered application and how it could be tested.

In Chapter 5, *Tabula Rasa – Approaching an Application with TDD in Mind*, we'll explore how to approach and application with a TDD in mind, turning theory into practice, and how better to grow an application through tests.

Tabula Rasa – Approaching an Application with TDD in Mind

It might seem a daunting task to develop an entire application with **Test-Driven Development** (**TDD**). Until now, all of the examples have been relatively small. The functions and methods have had a tiny, limited scope. How does TDD translate when developing full-fledged applications? Quite well, actually.

Topics discussed in this chapter include:

- Yak shaving
- Big design up front
- YAGNI
- Test small
- Devil's advocate

Where to begin

The best place to begin is at the beginning. Before a developer can start coding, they must know what the goal of the program is. What is the purpose of the application? Without a clear understanding of the problem that they are attempting to solve, it can be difficult to get started. At the very least, it is ill-advised to begin without some kind of plan.

The sooner you start to code, the longer the program will take.

– Roy Carlson, University of Wisconsin

Have you ever started a craft project without any objective in mind? How did you know what it was you were making? Did the project turn out well? If it did, you more than likely picked a direction at some point and set out to achieve a goal. You may have even had to start over or make adjustments along the way in order to complete the project.

Now, imagine starting the same craft project with the desired result defined ahead of time. Perhaps you wanted to make a drawing. Maybe you developed a set of plans.It isn't until a clear understanding is achieved before you start the physical act of beginning the project. In this example, the likelihood of success is much greater. The chance for stumbling blocks along the way are minimized.

Does this mean that all questions need to be asked ahead of time? Should all answers be obtained before you begin? Certainly not. Sometimes just a cursory understanding of a problem is enough to get started. But, the clearer the objectives, the better the likelihood of developing the proper solution.

Yak shaving

In the examples provided in previous chapters, you may have noticed there was a lot of moving around of code that didn't seem to have any immediate benefit. In TDD, especially at the beginning of a project, some work must be done that doesn't seem to make much sense. Tests are written that do nothing more than prove the existence of a class or method. The code is refactored in a way that only pushes hard-coded values into another dependency. This means that more files are created, and you may find yourself writing a significant amount of helper classes. All of these activities are referred to as yak shaving.

Yak shaving has two meanings that pertain to software development. The first and the one to be avoided is writing things that aren't needed as a means of procrastination. The second is the act of doing all the things that must be done to prepare the code. The difference between the two is a fine line. The side of the line you are on is determined by your intent in writing your code. Are you avoiding the code that you should be writing or are you laying the groundwork for efficient and effective software development using TDD?

In our case, as discussed in earlier chapters, we are either laying the groundwork for future tests, or we are implementing a known technique for preventing writer's block in our tests. Sometimes, the process of preparing an application for being tested can take quite a while.

When working in a legacy application, you could spend the better part of a week simply creating factories, adding interfaces to existing classes, writing test doubles, or doing safe refactoring techniques. All of these activities can help to improve testability and ensure a smooth testing experience. It is important to avoid getting carried away with these activities though. We only want to do them as a means of driving the next test forward.

Big design up front

It used to be common practice to have a lengthy, and expensive, **Software Development Life Cycle** (**SDLC**). Large teams were assembled. Meetings were scheduled and discussions had, ad nauseam. Requirements were gathered and documents were created which consumed reams of paper that would fill the filing cabinets of each and every team member. A design for the system would often be democratically assembled and a plan laid out for the system.

Once management and/or executive teams were satisfied, development could start. This long and cumbersome process often meant that budgets had already been significantly depleted with the cost of everyone's time in the planning stages. If for some reason, a flaw in the design was discovered during the development cycle, change orders and a slew of meetings would often occur.

Should the requirements change due to a change in markets or other external conditions it could potentially derail an entire program. If the SDLC did not allow for quick adaptation to change and rapid course correction, it would often spell doom for the entire project. Worse, if the change were significant enough it could render the need for the application obsolete.

Unfortunately, developing software in this manner was quite costly and would end in failure more often than success. The cost of change was too great and the resulting disruption was often detrimental to the process. These days software projects are more likely to be developed in some sort of Agile fashion.

A clean slate

So where do we begin a new application with TDD? Starting with TDD in mind is really no different from beginning any software development project. A developer must have some idea as to the goal of the application. The basic requirements should be understood. Just as we grow our application with tests, the requirements should grow with time.

One bite at a time

How do you eat an elephant? One bite at a time.

It is a massive undertaking to try to define and develop a monolithic application all at once. If you were tasked with creating Facebook you might not know where to begin. But, if you break the application down into logical portions such as *Login*, *User Dashboard*, and *News Feed*, it becomes much more manageable.

Minimum Viable Product

Each definition of work should be broken down into small deliverables. The concept of a **Minimum Viable Product** can apply to all aspects of our code. As the requirements for the monolithic application are broken down into manageable chunks, it might be possible to start coding. If a programming task is small enough to take only a couple of hours to complete, it's quite difficult to deliver something that wildly misses the mark. However, if a change is required, feedback should be given, and adjustments can be made quickly.

Different mindset

As an application is being developed with a view towards TDD, you should take the same approach to <mark>small deliverables.</mark> Write a little test, write just enough code to make it pass, then refactor. If you're constantly running your test suite, or better yet, you are using a continuous test runner such as NCrunch, your feedback cycles should be quite quick indeed.

> *Never leave an ignored test, or ignore more than one test at a time.*

If a test begins to fail during the development cycle it should be easy to recover. The code just written must be at fault. Pause the current effort and evaluate. Is the change necessary? Does the failing test need to change? *Skip* (xUnit) or *Ignore* (MSTest) your current test, if needed. Fix the code and resume by un-ignoring your test. Never leave an ignored test, or ignore more than one test at a time. Doing so will only risk the test (or worse, tests) never being completed, fixed, or recovered. An ignored test has no value. If a test is un-ignored at a later date by you or someone else and is now (or still) failing, it may be difficult to determine if the test is valid and indicates a true failure, or invalid and possibly sending you on a wild goose chase. Make sure your tests are valid, accurate, and provide value.

YAGNI – you aren't gonna need it

At times, you might be compelled to write some code because you think you'll need it. It's just a simple method. If you have a table full of data you'll probably need a `GetAll` method and a `GetById` method. A word of caution here: don't write any code until you have a true need for it. The more code that is written, the more code needs to be maintained. If you write code that you think you might need, but never actually use, you've wasted effort. Worse yet, you've introduced code that must be maintained until or unless it is removed.

> *Don't write code in anticipation of a future need. This is wasteful and often costly to develop and maintain.*

Test small

One of the <mark>most important things to consider when doing TDD is the size and scope of your tests</mark>. TDD is an exercise in fully understanding the problem you are trying to solve and being able to break the solution up into as many <mark>tiny little pieces as possible</mark>.

As an example, let us consider something simple: an application to manage a list of items that need to be done. How can we break up the use cases for this application?

First, using what we discussed with yak shaving, we can verify that the application even exists.

```
public class ToDoApplicationTests
{
  [Fact]
  public void TodoListExists()
  {
    var todo = new TodoList();

    Assert.NotNull(todo);
  }
}

internal class TodoList
{
  public TodoList()
  {
  }
}
```

Next, verify that you are able to retrieve a listing of items to be done.

```
[Fact]
public void CanGetTodos()
{
  // Arrange
  var todo = new TodoList();

  // Act
  var result = todo.Items;

  // Assert
  Assert.NotNull(result);
}
```

Devil's advocate

We will continue to demonstrate testing small, but already we have hit our next example. Playing devil's advocate is a useful technique in many circumstances. The way that we play devil's advocate in TDD is by imagining the simplest, and possibly most erroneous, approach to making the test pass. We want to force the test to make the code right instead of writing the code that we believe to be correct. For instance, in this case the desire is to make the test that was just written pass by adding an *Items* list. But the test doesn't require that at this point. It only requires that Items exists as a property on the class. There is no designation of a type in the test. So, to play devil's advocate, make the test pass by using *Object* as the type and setting the `Items` object to a simple non-null value.

```
internal class TodoList
{
  public object Items { get; } = new object();

  public TodoList()
  {
  }
}
```

Okay, now all the tests pass but that clearly isn't a proper solution. Thinking small steps, we could force the implementation to have a count, surely that will require it to be a list of *Todos*. Add the following to the last test:

```
Assert.Empty();
```

To make that pass, `Items` must change:

```
public IEnumerable<Object> Items { get; } = new List<Object>();
```

Remember what we discussed about the SOLID principles in , *What to Know Before Getting Started*. We want to use interface segregation and limit ourselves only to the interface we need. We don't need the full `IList` interfaces capability so we don't need to use it. All that is needed is the ability to iterate over a collection of items. The simplest interface for doing this is `IEnumerable`.

We still have a problem though: we are using an `object` as our enumerable type. We want to use only a specific class. Let's fix that now. Modify the last test one more time to include a type assertion.

```
[Fact]
public void CanGetTodos()
{
  // Arrange
  var todo = new TodoList();

  // Act
  var result = todo.Items;

  // Assert
  Assert.NotNull(result);
  Assert.IsAssignableFrom<IEnumerable<Todo>>(result);
```

```
    Assert.Empty();
  }
```

Now, update the class, shown as follows:

```
internal class TodoList
{
  public IEnumerable<Todo> Items { get; } = new List<Todo>();

  public TodoList()
  {
  }
}

public class Todo
{
}
```

As you can see, we added what seemed to be a fairly small test and ended up creating a property, assigning a default value, and creating a class. Can you think of any way we could have made this smaller?

Our next test might verify that the Todo items start as empty, but if we think back to the laws of TDD, the first law is to write a failing test. Right now, if we wrote a test that verified `Items` to be empty we would expect that test to pass. So, what test should we write?

The test we have decided to write next is a test to verify a means to add a Todo item.

```
[Fact]
public void AddTodoExists()
{
  // Arrange
  var todo = new TodoList();
  var item = new Todo();

  // Act
  todo.AddTodo(item);
}

internal class TodoList
{
  public IEnumerable<Todo> Items { get; } = new List<Todo>();

  public TodoList()
  {
  }

  internal void AddTodo(Todo item)
  {
  }
}
```

Up to this point, we have been taking steps that would likely resemble the same steps you would take in normal development, cutting giant swathes of functionality into the code. This is the first test where we stop before we have actually achieved valuable functionality. This is part of taking those small steps though. We could deploy the application right now. It wouldn't be very useful but we do have that option. If we had reached the end of our sprint, the product owner might request that, in order to deploy as soon as possible, we hard-code in some Todo items just so something is available in the UI.

Our next test seems to be fairly straightforward. We will verify that we can actually add a Todo using our new method. There is a catch though because this test is testing functionality and not

general class structure. We suggest having a test class specifically for this method.

```csharp
public class TodoListAddTests
{
  [Fact]
  public void ItAddsATodoItemToTheTodoList()
  {
    // Arrange
    var todo = new TodoList();
    var item = new Todo();

    // Act
    todo.AddTodo(item);

    // Assert
    Assert.Single(todo.Items);
  }
}

internal class TodoList
{
  private List<Todo> _items = new List<Todo>();

  public IEnumerable<Todo> Items
  {
    get
    {
      return _items;
    }
  }

  public TodoList()
  {
  }

  public void AddTodo(Todo item)
  {
    _items.Add(item);
  }
}
```

Now, that really was a flying leap off a cliff. That one test nearly changed all of our application code. We just completely changed the implementation of `Items`, and we added code to the `AddTodo` method. Is there a way that we could have broken those into two or more steps? We still have a lot to do with this application, and we will cover some of it. But, before we go on, write down the next few tests that you think you would write. Try not to skip this exercise because breaking up functionality into small chunks like this is one of the areas where most developers struggle when learning TDD.

We are going to temporarily pause the forward progress of this sample application because we have already begun to work ourselves into a corner. To prevent getting blocked, we should be testing negative cases first.

Test negative cases first

What does it mean to test negative cases first? In many computer games, especially role-playing games, it is common for the game designers to make it very difficult to win the game if you simply go straight to the boss. Instead, you must make side quests, make wrong turns, and get lost in the story before you can fight the boss. Testing is no different. Before the problem can be solved, we must first handle bad input, prevent exceptions, and resolve conflicts in the business requirements.

In the Todo application, we mistakenly flew through and added an item to the Todo list without verifying that the item was valid. Now, the sprint is over and our user interface developers are mad at us because they do not know what to do with a Todo item that has no details at all. What we should have done is handle the cases where we receive bad data first. Let's rewind and temporarily skip the test we just made.

```
[Fact(Skip="Forgot to test negative cases first")]
public void ItAddsATodoItemToTheTodoList()
```

The test we need to write now should go above the test that was just ignored, but in the same file. Remembering that we need to have small test increments, we can write a test that guards against the simplest bad data, `null`.

```
[Fact]
public void OnNullAnArgumentNullErrorOccurs()
{
  // Arrange
  var todo = new TodoList();
  Todo item = null;

  // Act
  var exception = Record.Exception(() => todo.AddTodo(item));

  // Assert
  Assert.NotNull(exception);
  Assert.IsType<ArgumentNullException>(exception);
}

public void AddTodo(Todo item)
{
  throw new ArgumentNullException();
}
```

Notice that we have removed the code that was in place for `AddTodo`. We could have left the code in place, but at this point it is clutter and there is currently no test that forces that code to be present. Sometimes, when you ignore a test, it is easier to remove the functionality that test was verifying instead of working around the functionality. There are times when the clutter could restrict your refactoring efforts and could result in worse code. Don't be afraid to delete code for tests that are being skipped, and don't be afraid to delete skipped tests that make their way into source control.

One other issue that we encountered when making this change is that the `AddTodoExists` method defined earlier in the `TodoApplicationTests` class is now failing. This test was a yak shaving test to

start with and does not add any real value to the test suite, so just remove it.

Now that we have the null case covered by our method, what is the next thing that could go wrong? Thinking about it, are there any required fields for a Todo? We should probably make sure the Todo has a title or description at least before we add it to the list.

First, before we can verify that the field has been populated, we need to verify that the field exists on the model. Writing model tests might seem a bit like overkill to you, but we find that having these tests helps to better define the application for others coming into it. They also provide a good attachment point for field validation tests later on when your business decides that the description field of a Todo has a maximum length of 255 characters. Let's create a new class for the Todo model tests.

```
public class TodoModelTests
{
  [Fact]
  public void ItHasDescription()
  {
    // Arrange
    var todo = new Todo();

    // Act
    todo.Description = "Test Description";
  }
}

public class Todo
{
  public string Description { get; set; }
}
```

As you can see, there is no real assert for this type of test. Simply verifying that we can set the description value without throwing an error will suffice.

Now that we have a description field, we can verify that it is required.

```
[Fact]
public void OnNullADescriptionRequiredValidationErrorOccurs()
{
  // Arrange
  var todo = new TodoList();
  var item = new Todo()
  {
    Description = null
  };

  // Act
  var exception = Record.Exception(() => todo.AddTodo(item));

  // Assert
  Assert.NotNull(exception);
  Assert.IsType(typeof(DescriptionRequiredException), exception);
}

internal class TodoList
{
  ...

  public void AddTodo(Todo item)
  {
    item = item ?? throw new ArgumentNullException();

    item.Description = item.Description ?? throw new
                                   DescriptionRequiredException();
```

```
    }
}
```

We are long overdue for some refactoring and this is a good place to pause our testing efforts and refactor. We would like to move the model validation into the model. Let's create a quick test for a validation method on the Todo model and then move that logic into the `Todo` class.

```
public class TodoModelValidateTests
{
  [Fact]
  public void ItExists()
  {
    // Arrange
    var todo = new Todo();

    // Act
    todo.Validate();
  }
}

public class Todo
{
  public string Description { get; set; }

  internal void Validate()
  {
  }
}
```

Now, at least for the moment, we want to move our validation logic over from the Todo list into the model. In creating the validation test and moving the logic, we have caused our yak shaving test to fail. The test is failing because, although the required method exists, it is throwing an exception because we have not populated the description of our Todo. We will have to remove this test as it no longer adds value.

```
public class TodoModelValidateTests
{
  [Fact]
  public void OnNullADescriptionRequiredValidationErrorOccurs()
  {
    // Arrange
    var item = new Todo()
    {
      Description = null
    };

    // Act
    var exception = Record.Exception(() => item.Validate());

    // Assert
    Assert.NotNull(exception);
    Assert.IsType(typeof(DescriptionRequiredException), exception);
  }
}

public class Todo
{
  public string Description { get; set; }

  internal void Validate()
  {
    Description = Description ?? throw new DescriptionRequiredException();
  }
}
```

Finally, the tests we needed to write before we could make the refactoring change we wanted to

make are complete. Now we can simply replace the exception logic dealing with model validation in the `TodoList` class with a call to `Validate` on the model.

```
public void AddTodo(Todo item)
{
  item = item ?? throw new ArgumentNullException();

  item.Validate();
}
```

This change should have no effect on our tests or our resulting logic. We are simply relocating the validation code. There are many more validations that could happen. Can you think of a few that might be valuable?

It is now time to add back in our skipped test, with some minor modifications to pass validation.

```
[Fact]
public void ItAddsATodoItemToTheTodoList()
{
  // Arrange
  var todo = new TodoList();
  var item = new Todo
  {
    Description = "Test Description"
  };

  // Act
  todo.AddTodo(item);

  // Assert
  Assert.Single(todo.Items);
}

public void AddTodo(Todo item)
{
  item = item ?? throw new ArgumentNullException();

  item.Validate();

  _items.Add(item);
}
```

When testing is painful

There may come a time when you may encounter some pain. Perhaps you've forced yourself into a corner with your design. Maybe you're unsure what the next, most interesting test would be. Sure, you didn't mean to, but conceivably you could have taken too great a <mark>leap</mark> between tests. Whatever the case may be, there may come a time when testing becomes painful.

A spike

If you find that you're stuck or you're debating between options on how to proceed, it might be beneficial to run a spike. A spike is a means with which you can investigate an idea. Give yourself a time-limit or some other limiting metric. Once sufficient knowledge or insight has been gained by the exercise, throw away the results. The purpose of the spike is not to walk away with working code. The goal should be to gain understanding and provide a better idea of a path forward.

Assert first

At times, you may know the next test you want to write without being quite sure how to start. If this happens, start with *Assert* to determine the expected result. With the expectation defined, set out to make the actual value match the expected value. You might want to take this approach more often to assure that you're only writing enough code to make the desired test pass.

Stay organized

Remember, tests are the first consumer of your application. The best and most accurate documentation you can provide is a thorough and well-maintained set of tests. Within your test suite, create folders, nested classes, or utilize features of your test framework to make your tests more readable. Remember, if you do encounter a test failure at a later date, a descriptive test name and proper assertion will go a long way in describing how the result came to be.

Use `Describe` to better organize your JavaScript tests. Nest multiple levels by using more than one `Describe` within your tests.

Breaking down Speaker Meet

The Speaker Meet application started with a simple goal: connecting technology speakers, communities, and conferences. The idea was simple but could evolve into broad complexity. It was decided at an early stage to start small and add features if and when it made sense. New ideas should be able to be implemented and tested with little effort. If an idea turned out to be the wrong direction for the site, the new functionality could easily be removed and abandoned. Start simply and release small features for quick feedback.

Three main sections of the initial site were defined as *Speakers, Communities,* and *Conferences.* Each would need to have a listing of all speakers/communities/conferences, provide a way to view details about a selected item, and provide a way to search items based on a predefined set of criteria. This would be the Minimum Viable Product for the initial release.

Speakers

In the beginning, it was decided that speakers would be the initial focus. Speakers would contain a name, email address, technology selections, and location. *Gravatar* would be used to provide an avatar. Future enhancements that were excluded from the Minimum Viable Product include a list of talks, travel distance, and ratings. By focusing on this limited functionality, initial feedback can be collected and future effort can be directed appropriately.

Communities

The secondary focus of the Speaker Meet application revolved around technology communities. Meetups and user groups are typically run by dedicated volunteers that are always looking for new and interesting speakers for their meetings. The main goal of the community section of the website is to define a name, location, meeting day/times, and technology selections of member communities.

Conferences

Technology conferences are the third and final focus of the Speaker Meet site. Conferences have similar requirements to communities, in that they require a name, location, dates, and technology selections. They differ primarily in size, scope, and dates. User groups typically will have one meeting per month where one speaker may present to a small crowd. Conferences typically occur once a year, from one to many days, with many speakers presenting to many more attendees.

Technical requirements

The technologies to be used for this project were decided early on, based on the knowledge and experience of the team. JavaScript and ReactJS were to be utilized for the front-end website. The back-end would utilize C# and WebAPI with .NET Core, Entity Framework, and SQL Server. All would be hosted in Azure. Knowing these technical requirements before coding starts goes long way towards defining parts of your system.

Summary

Now, you should have a basic understanding of Yak shaving and how it might help you get started. You've been cautioned about *Big design up front* and creating things that you might not need in anticipation of a time when they might be needed (YAGNI). Be sure to test small, play devil's advocate, and test negative cases.

In , *Approaching the Problem*, the three sections of the Speaker Meet site will be discussed in much greater detail. More effort will be put into breaking down these initial statements into meaningful requirements and manageable units of work.

Approaching the Problem

In Chapter 5, *Tabula Rasa – Approaching an Application with TDD in Mind*, the details of the Speaker Meet application were discussed. The requirements have been defined at a very high level. A picture has been painted with very, very broad strokes. This is often how the concept for many applications begins, with a high-level description and an important key functionality defined. It may have started with a bar napkin or a whiteboard sketch, but somewhere, somehow an idea was formed.

In this chapter, we'll cover:

- Defining the Speaker Meet application
- Architectural choices
- Testing direction

Defining the problem

To define the problem, first the vision must be defined. Clear objectives should be described and outlined. The Speaker Meet problem came about as a result of technology speakers looking for a single, centralized place in which to find speaking opportunities and venues. It was determined that user group and conference organizers were equally in need of a solution to seek and find speakers for their meetings.

Thus, the idea for Speaker Meet was born. But, how would the application work? Should it be a mobile application or a website? How would the data be collected and managed? Would users be allowed to create their own profiles? Could users submit speaker, community, and conference information? Where would the application live and how would it be hosted? And where in the world do we begin?

Digesting the problem

The problem the application will be designed to solve has been defined. Speaker Meet will bring technology speakers, communities, and conferences together. Now that the purpose has been defined, it must be digested.

As was suggested before in a previous chapter, attacking a new application from all directions is ill-advised. It can be quite a daunting task to attempt to approach a new software project by implementing each and every desired feature all at once. It can also be a large chore to define every want and need of the system.

It would be far better to define small, manageable chunks of the application that can be delivered quickly in order to evaluate their correctness and effectiveness. The trouble is, how does one define what can be separated into small pieces and determine that this small piece is of sufficient value?

Epics, features, and stories; oh my!

Many software development projects will maintain what is referred to as a *product backlog*. This is where everything that the system might be asked to do is compiled. The product backlog might contain the largest of ideas down to the most minute detail. The important thing is that these ideas are recorded.

The backlog should be regularly groomed and maintained. Items should be evaluated for their importance and ordered appropriately. If it is determined that an item is the next most important thing on which to be worked, it should be broken down into appropriately sized stories for the team to effectively deliver in a timely manner.

Epics

Larger, broader ideas are defined as *epics*. These could potentially be quite significant and wide-ranging in scope and size. Speakers would be defined as an epic. The speakers epic is a segment of the application devoted to anything and everything related to technology speakers.

The term epic is used to signify that the features and stories contained within the epic all revolve around a single, central idea. These essentially start life as a single, large user story, and are broken down into smaller features and stories. Epics can often take several sprints to complete.

Features

Features are generally smaller than epics and are contained within epics. A feature will usually contain many stories related to the subject matter it is responsible for. Think of features the same way you think of epics, they are just a smaller grouping.

A feature might comprise a s*peaker catalog* or s*peaker detail*. The speaker catalog might contain everything associated with displaying, sorting, filtering, and searching for speakers within the system. The speaker detail feature might define details and functionality regarding single, individual speaker information and how it is displayed within the application.

Stories

Depending on team preference, a story might be as small as seeing a <mark>speaker's name when viewing their details</mark>. A word of caution: it is possible to have too small a story. It is better to break a story down so it's just small enough and begin work than to waste time on minute details. If done correctly, the details will emerge during the development cycle.

Determining what is small enough should be left to the team to decide. A good rule of thumb is that stories should take <mark>from half a day to three days to comple</mark>te. Less than half a days' worth of work and the story will likely be broken down into pieces that are too small. More than three day's work and there is likely an opportunity to break the story into two or more stories.
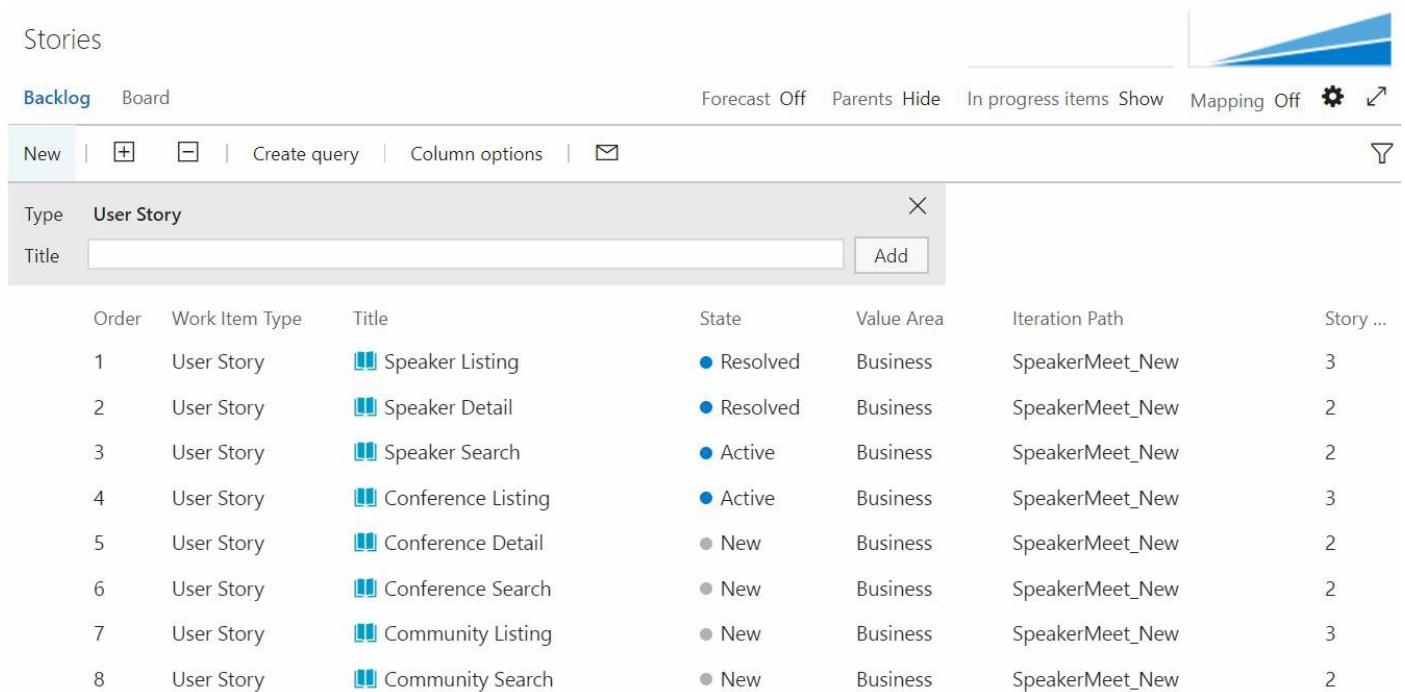
It is possible to have too small a story.

Don't fall into the trap of breaking down stories into piece that are too small. Effort can be wasted by trying to compose smaller and smaller stories. If you're practicing Scrum, remember that small improvements can and should be proposed at the conclusion of each sprint. During the retrospective, story size and its effectiveness should be discussed. If it is decided that the size was not appropriate, whether that be too large or too small, adjustments should be made before the next sprint begins.

Maintain your backlog

So why is it so important to maintain the product backlog? A well-maintained backlog will define what the team should work on and help them plan for known, upcoming tasks. This will also help a team put together forecasting in order to plan for when a particular feature might be completed.

If appropriate metrics are captured, a well-disciplined team can deliver a reliable velocity in each sprint. With appropriately sized and estimated stories a reasonable timeline can be predicted for upcoming items in the backlog. For example, if a team has reliably delivered 20 points per sprint and the next five stories have been estimated at 8 points each it would not be unreasonable to expect these five stories to be completed in about two sprints. Of course, this is not a commitment; it is merely an estimate.

The Speaker Meet problem

Remember that the application scope will initially be kept small and limited in functionality. Some features might be identified now as an item for the future, but the limited scope for the <mark>Minimum Viable Product</mark> still needs to be better defined. More features will continue to be added to the product backlog and prioritized quite low if it is determined that they are not needed for a first release. Consider, though, that a <mark>minimum viable product still needs to deliver some value</mark>. A software application that doesn't do anything isn't worth much to anyone.

Taking the time to prioritize the potential value of features and stories will help decide what should be included in an initial release, and what can wait. By determining the effort involved to deliver specific functionality and combining that information with the proposed value, an educated decision can be made regarding which <mark>features will be delivered</mark> first.

Meaningful separation

Brainstorming the proposed features of the application will help to describe the system. Finding meaningful, logical separation will help define the scope of particular sections of the software solution. Logical boundaries could include epics and features as defined in the product backlog. They could also be determined by divisions in technology.

Speakers

The speaker epic will be made up of all of the features and stories surrounding the speakers portion of the application. This will include the speaker catalog and speaker details. This section will also contain any future enhancements and features that might be added at a later date. Future functionality might include speaker ratings and reviews, slide decks, and presentations, YouTube or Vimeo videos, and so on. These have yet to be determined and can be evaluated at a later date when a proposed value can be weighed.

> *Not all functionality needs to be decided up front. Remember, work towards a Minimum Viable Product and build functionality as it's needed.*

Here are some basic features stories for the speaker epic:

```
As user group organizer
 I want to see a listing of all speakers
 So that I can find speakers for my user group.

As conference organizer
 I want to see details of a particular speaker
 So that I might view more information about them.
```

> *We are using a story format to describe detail at all levels of the application. That is, epics are presented as a story, themes or features are presented as a story, and specific requirements are presented as a story. In the hierarchy, only the specific requirements are called user stories though. The reason for giving them all the story format is simple. We want to be able to write a requirement and be able to transition it from user story to feature or even to epic with minimal hassle. So, we use the same format for a requirement regardless of the abstraction level of that requirement.*

These feature stories are a good start. This will provide the business with an opportunity to grade and prioritize features before determining what should be worked on first. When presented to the team, these will likely need to be broken down into smaller, more detailed user stories with acceptance criteria.

Good acceptance criteria will help the team determine when a story can be marked complete. If all conditions have been met, then the story is done and can be delivered. If at some point, it is decided that more work is needed in order to deliver the requested functionality, additional criteria should be included or new stories added to the backlog.

```
As user group organizer
 I want to see a listing of all speakers
 So that I can find speakers for my user group.

Given system contains speakers
 When viewing speaker catalog
 Then a listing of all speaker summaries is returned.

As conference organizer
```

I want to see details of a particular speaker
So that I might view more information about them.

Given specified speaker exists
 When speaker selected
 Then speaker details are returned.

Given specified speaker does not exists
 When speaker selected
 Then a friendly error message should be returned.

Communities

User groups and meetups comprise the community section of the application. The main purpose of this portion of the application is to provide a place for speakers and potential members and attendees to find and discover technical communities in their area. Anyone traveling to a specific city might also be interested in learning which user groups or meetups are available to them, whether for speaking purposes or general attendance. The community segment of the application will include the community directory and user group details. If any future enhancements are proposed, they can be added as new features or user stories to the community epic.

At some point in the near future, location search will be added to the community portion of the system. This would allow the user to search for communities based on distance, perhaps allowing speakers to determine communities within a 200-mile radius at which they may be interested in speaking. This feature was determined to be unnecessary for an initial release of the Speaker Meet application.

A small list of community feature stories can be found here:

```
As a speaker
 I want to see a listing of all communities
 So that I can find potential user groups at which to speak.

As a speaker
 I want to see details of a particular community
 So that I can learn more about the user group.
```

Much like the speaker feature stories, the community feature stories will help the product owner prioritize the functionality to be developed. These, too, will likely need to be broken down into smaller, more detailed user stories with acceptance criteria. Take a look at the stories here:

```
As a speaker
 I want to see a listing of all communities
 So that I can find potential user groups at which to speak.

Given system contains communities
 When viewing community catalog
 Then a listing of all user groups is returned.

As a speaker
 I want to see details of a particular community
 So that I can learn more about the user group.

Given community selected
 When specified community exists
 Then community detail returned.

Given community selected
 When specified community does not exists
 Then a friendly error message should be returned.
```

Conferences

The details and functionality within the application regarding conferences are defined and described in the conferences epic. This will include the conference catalog and conference details. Future enhancements and features proposed at a later date may be added to the conferences epic.

Conferences, too, may utilize location search. There are a variety of third-party services available and they can be evaluated for inclusion in a future release. Like all third-party code, these will be abstracted away from the main application, so that the system is insulated from potential changes.

```
As a speaker
 I want to see a listing of all conferences
 So that I can find conferences at which to speak.

As a speaker
 I want to see details of a particular conference
 So that I can learn more about the conference.
```

Conferences differ from communities in that they happen only once per year and often have many speakers and sessions for the event. The conference feature stories will help the product owner prioritize the functionality to be developed. These, too, will likely need to be broken down into smaller, more detailed user stories with acceptance criteria. Take a look at the stories here:

```
As a speaker
 I want to see a listing of all conferences
 So that I can find conferences at which to speak.

Given system contains conferences
 When viewing conference catalog
 Then a listing of all conferences is returned.

As a speaker
 I want to see details of a particular community
 So that I can learn more about the user group.

Given specified conference exists
 When conference selected
 Then conference detail returned.

Given specified conference does not exists
 When conference selected
 Then a friendly error message should be returned.
```

Separate by team function

Many self-organizing teams split themselves by expertise. This might mean that members divide themselves into front-end developers, back-end developers, QA, and so on. Likewise, stories and tasks can be separated by functionality.

It is best left up to the team to decide how to effectively organize themselves and their body of work. For example, Sally may be the most knowledgeable developer when it comes to the .NET framework, while Steve may have more expertise with React. It might prove better to let Sally take a majority of the back-end stories and let Steve focus on front-end functionality.

Note that it is an easy pitfall to prioritize stories in such a manner that each team member has the most suitable work for them available. This will be efficient but not effective. Instead, priorities should focus on value delivered and optimized later on. There's no harm in letting someone (for example, Sally) work on UI features together with Steve when for example large UI design changes are needed.

Technical separations

There may be a time when you must perform some work that doesn't fit neatly into the epics defined previously. Non-functionality requirements may include items related to the technology chosen for parts of the system. Stories could comprise purely web or front-end functionality, such as bundling JavaScript files. Alternatively, back-end or server-side functionality may need to be defined outside the previous epics.

There will likely be a number of non-functional or system specifications that will also need to be evaluated. Examples of these requirements might include response times, throughput, or memory consumption. These are commonly added to the checklist for *Definition of Done* so that each story should confirm the non-functional requirements.

Many modern web-enabled applications are built as a **Single Page Application** (**SPA**) using JavaScript. These applications are hosted by a web server and delivered to a web browser on request. The entire application, or rather large pieces of the application, are delivered all at once. As requests are made by the client browser, the SPA will update the data on screen or mimic a page transition. Full-page postbacks and page reloading are not used with an SPA. This provides a perceptible increase in performance and increases in responsiveness for the end user. It also allows for distributing some of the processing of an application to client machines.

With this division of the SPA, much of the functionality can be split into *web* and *non-web* designations. A team may choose to write their stories in this way. Similarly, a team may choose to designate web specialists to work primarily on web-related functionality. One issue with this split is that the single story with just a front-end or back-end is not potentially shippable software. They do not separately deliver value. Instead, the story could be split by stripping out special case handling, offering only one purpose, keeping the UI simpler, and so on.

Like the web designation, a team may decide to separate stories into server-side or back-end functionality. This might cover all functionality from the API to the database and everything in between. The back-end of the Speaker Meet application is written in .NET with C# and Entity Framework Core and a SQL Server database. These technologies provide an excellent opportunity to create technical separations.

Defining a consistent API, for example, is an excellent place to start. How the back-end might be further subdivided is discussed later in this chapter.

Technical requirements

The Speaker Meet application has an assortment of technical requirements. Language choices and platform decisions can have an immense effect on an application. These decisions will determine how an application is delivered to a client and how many parts of the application are expected to behave.

Technology specifications can have a big impact on an application. Whether **LAMP** (**Linux, Apache, MySQL, PHP/Perl/Python**), **MEAN** (**MongoDB, Express, Angular, Node**), or in the case of Speaker Meet .NET and React, programming languages and frameworks can play a big role in a software system.

React web user interface

The first *user interface d*efined for the Speaker Meet application is an SPA using React, a JavaScript library. React was written by the Facebook team for the purpose of developing modern web applications. This equates to the View in the traditional *Model-View-Controller* template. By using a one-way data flow model along with the *Virtual DOM,* React is an extremely powerful library that performs well and scales nicely.

Many additional libraries will be included using the JavaScript package manager NPM. Additional libraries include webpack, a bundler for JavaScript, CSS, and other such files. More will be introduced in the following chapters.

.NET Core

The primary language for the server-side application will be C# in .NET Core. With the release of the latest overhaul to the .NET Framework, developers can choose which parts of the framework to include within their application and keep core level libraries to a minimum.

.NET Web API

The way to expose internal information and behavior to an external system, and the SPA is considered an external system, is to provide an **Application Programming Interface** (**API**). The API layer exposes data functionality to the outside world. The primary gateway into the application is an assemblage of APIs using .NET Web API.

Entity Framework

For the Speaker Meet application, an **Object-Relational Mapper** (**ORM**) is utilized to convert database objects into C# objects. There are many such ORMs available for a variety of different languages and platforms. In .NET alone there is NHibernate, LLBLGen, Dapper, and many more. For the Speaker Meet application, **Entity Framework** (**EF**) Core was selected.

Choosing an ORM mapper such as EF Core in and of itself is a requirement that will affect architectural choices for an application. The team will likely need to determine the pros and cons of ORM options available to them, and whether to use an ORM at all.

Azure

The Speaker Meet application is hosted using Microsoft Azure. Choosing Azure allows the team to scale up or down parts of the application as demand arises. Of course, there are architectural decisions that must be made to effectively leverage the available functionality that Azure provides.

> *Knowing about upcoming or desired future functionality can allow a team to make wise decisions while developing parts of an application.*

Future enhancements are planned to employ the power of Azure Search. The core search functionality was written in such a way that switching to Azure Search would have minimal impact on the rest of the system. Implementing Azure Search, of course, would be developed using TDD.

Database

Microsoft SQL Azure is utilized to persist speaker information, user group and community particulars, information about conferences, as well as user login details. SQL Azure is very similar to using SQL Server on-premise, with a few caveats. For example, SQL Azure requires clustered indexes on each table. Knowing the requirements and differences of available database options allows the team to make an informed decision about their data storage choices.

An N-Tiered hexagonal architecture

In a previous chapter, the N-Tiered architecture was discussed, where a software application is divided up into layers. N-Tiered applications are typically separated in successive layers, like the layers of a cake, from *A* to *B* to *C* and so on. There is a danger in defining an application in this way, as sometimes pieces of functionality don't cleanly fall into one layer. As long as the layers remain loosely coupled and functionality does not cross the boundaries, your application should remain well-structured and organized.

Hexagonal architecture

The hexagonal architecture was first described by Alistair Cockburn in the 2000s. Hexagonal architecture has also been referred to as ports and adapters, in which ports are abstractions and adapters are the implementations. This approach to designing applications changes the concept of layers to one of internal and external pieces to the application.

Some may argue that the hexagonal architecture and N-Tiered architecture are one and the same. While it's possible to achieve a hexagonal architecture using an N-Tiered linear layered approach, the main distinction lies in how the layers interact with one another—linear or through specific ports and adapters: two distinct zones, internal and external bits. In the simplest of terms, a hexagonal method will save you in the event that something doesn't fit neatly into a series of sequential layers and helps to prevent tight coupling between layers.

The main thing to remember is that there are things that need to be separated—data source, user interface, third-party libraries, frameworks—essentially anything that isn't written by your team, possibly even the layers themselves. With the use of the *Dependency Inversion Principle*, as discussed in a previous chapter, and the *Repository Pattern*, coupling can be kept to a minimum. This allows for greater flexibility, maintainability, and testability.

Greater flexibility can be provided by minimizing coupling between parts. New features can be plugged into the existing application. Existing parts of an application can be swapped out in favor of something else entirely. This simply cannot be done if existing parts of your application are tightly coupled to other parts.

If an application is segmented properly, it becomes much easier to maintain. By strictly adhering to the SOLID Principles as outlined in a previous chapters, this becomes almost effortless. With the strict adherence to hexagonal design and by keeping internal logic free from outside dependencies, it is simple to make modifications without impacts on other parts of the system.

Testing a loosely coupled system is much easier than the alternative. By limiting the dependencies, tests can be limited to the functionality of the method, function, or system under test.

Basic yet effective N-Tiered divisions

Typical layers in a three-tiered application include presentation, business logic, and data access. These can be and often are subdivided even further, but this is a basic starting point for many applications.

By dividing an application in this way, the first separation of concerns is born. Business logic should not be found in the presentation layer. Data access code should not be found in the business logic layer.

A place for everything and everything in its place.

-Mary Poppins

Service layer

The business layer, or *service layer,* is where the business logic for the application resides. Whether you choose to use the idea of individual services, managers, or domain objects, the idea is effectively the same. The logic of the application should reside in a separate place from the presentation information and data access code.

Microservices

You may have heard the term *microservices* at some point in your development career. These are typically very small, independent applications that serve one and only one purpose for the rest of the system. Whether they be standalone APIs or executables deployed to Azure Service Fabric, they can be developed and deployed independently from the rest of the application. Microservices tend to be small, reusable functions, often consumed by a number of different applications or deployed user interfaces.

Data access layer

Instead of littering the rest of the application with data persistence code, many applications rely on a data access layer of some sort. This allows for a centralized location of all data retrieval and storage procedures.

As the Speaker Meet application relies on EF Core, the data access layer is where much of this information will reside.

Repository Pattern

The Repository Pattern allows for abstraction between the domain layer and the data access layer. This allows for the rest of the application to be agnostic to the way data is persisted or retrieved. This allows for improved testability and for code reuse within the repositories themselves.

Generic repository

As much of the data access functionality is the same across database models, a g*eneric repository* is used to minimize duplication of code. Many standard **CRUD** (**Create, Read, Update, Delete**) operations are used across all database objects. This provides the opportunity to create a generic repository to be used across all models and this will be covered in Chapter 7, *Test Driving C# Applications*.

As in life, often one size does not fit all. While the generic repository fits most cases, there may come a time when you need to create a specific repository or to extend the generic repository. These instances should be carefully evaluated and a proper solution should be put in place for them.

User interface adapter layer

The user interface adapter layer is where a user interface can "plug in" to the rest of the application. The Speaker Meet application provides a collection of web APIs to provide data and functionality to external systems. The first such external system is the React SPA. Utilizing a user interface adapter layer allows for the replacement or addition of a new UI application. This could be in the form of native mobile application, a Facebook application, or integration with another external site such as Meetup.

User interface layer

Modern applications have a dual N-Tiered approach with architectures on both the back-end and front-end. This means that as much planning and separation as is done on the server side, the same amount of effort could also be spent architecting a UI application.

With much of the functionality of an entire system being delivered to the client, the SPA in the case of the Speaker Meet system can be treated as a wholly independent application. It, too, must have its own application architecture specification.

Front-end business layer

Using Redux action creators allows for front-end business logic to be contained in a single layer or location. Within an action creator, behavior can be encapsulated and concerns separated. Reusable functions may be exposed, minimizing code duplication.

Front-end user interface layer

React components and containers provide the presentation to the end user. Reusable components should be created and kept small, and without external dependencies.

Front-end data source layer

Using React with Redux, data will be stored in state on the client's machine by way of a reducer. The shape of the data store should be carefully planned and evaluated. If something is not shared by more than one component, then it should likely not be placed in state. If you need the same data to take on many shapes, consider the use of something such as React Reselect, which provides a way to transform or compute derived data for use throughout your application.

Testing direction

Now that you have a basic plan for your architecture you have to think about where you should begin your testing. There are a few options for where to start:

- You could choose to start testing at the data access or data source layers and work your way up to the user interface layers. This method is a back-to-front approach to testing.
- You could start at the user interface layers and work your way to the data access layers. Approaching the tests in this manner is a front-to-back method of testing.
- Lastly, you could start testing in the business layers and work your way out to the hexagonal boundaries of the system. This method is an inside-out testing approach.

As a demonstration of the three testing directions to be examined, the same scenario of user login will be used.

Back-to-front

Most back-end developers have been taught to think in a database-first manner. This style of thinking will lead them to find that a back-to-front style of testing makes more sense. As mentioned previously, in back-to-front testing you start at the data access layers. Mentally you really start by imagining the data structure within the data source. Once a data source has been defined, you can move up a layer and begin thinking about the business layer's design. Finally, you can apply the models and functionality you have created to a user interface.

Defining a data source

By starting in the data layer, you are presented with defining your data model as early as possible. For this application and the requirements you have received, we suggest you go with a SQL database and use an entity framework for your data connections. Since you are working in a relational database, you will need some kind of primary key. These keys are for relational database concerns and are often not mentioned in the system requirements. In a situation like this you might end up with a table that looks something like this.

UserProfile		
ID	**Integer**	**Primary Key, Identity**
Username	`Varchar(255)`	Unique, Not Null
PasswordHash	`Binary(64)`	Not Null
FirstName	`Varchar(255)`	Not Null

Now that you have a table defined, you can see that, instead of having a simple password field, you must use a password hash for security reasons. The next step is to create the data access layer code that will interact with this table.

Start with tests to properly define the model. These tests will provide some of the validations defined in your requirements and put you in a good place to define the entity framework model builder relationships.

```
public class UserProfileDtoTests
{
  [Fact]
  public void ItExists()
  {
    var dto = new UserProfileDto();
  }

  [Fact]
  public void ItHasAnId()
  {
    // Arrange
    var dto = new UserProfileDto();
```

```
        dto.Id = 1;

        // Act
        // Assert
        Assert.Equal(1, dto.Id);
    }
}
```

These are the tests that will get you started testing the model, the rest is up to you as an exercise. At the end of it, you should have a model that looks similar to this one.

```
public class UserProfileDto
{
  public int Id { get; set; }
  public string Username { get; set; }
  public string FirstName {get; set;}
  public byte[] PasswordHash { get; set; }
}
```

As you can see, this model is not too complicated, but could get that way quickly if the number of database fields needed to expand. This is only a partial example of what a user profile would look like. Before moving on, think about what other fields would be needed and how they might need to be tested.

Now that you have a data transfer object, you need to be able to read that model into the application from the database. As mentioned in Chapter 3, *Setting up a JavaScript Environment*, in the *What to Know Before Getting Started* section, we prefer to use a repository pattern for this. As a quick recap, the repository pattern is a simple pattern that helps us deal with create, read, update, and delete operations on a data source.

We are only going to use as much of the `FakeRepository` as is needed. For now that means that we will only implement `Get` and `GetAll`.

```
public class FakeRepository<T> : IRepository<T> where T : class
{
    public IList<T> DataSet { get; set; } = new List<T>();
    public T Get(Func<T, bool> predicate)
    {
        return GetAll().Where(predicate).FirstOrDefault();
    }

    public IQueryable<T> GetAll()
    {
        return DataSet.AsQueryable();
    }
}
```

Now that we are using a `FakeRepository`, we can move on to business layer integration.

Creating a business layer

Using the `UserProfileDto` defined previously, you can now focus on the service needed to log on. As you will be dealing with the `UserProfileDto` and repository, call this the `UserProfileService`. It will house all the interactions in the app with user profile objects.

Right now, you only need to worry about the logon capabilities of the system. You will create a `GetUser` method which will consume a username and return a `UserProfile`. Then you will use the `UserProfile` and a password to authenticate.

First, here is the starting test to create a `UserProfileService`.

```
public class UserProfileServiceTests
{
  [Fact]
  public void ItExists()
  {
    var service = new UserProfileService();
  }
}

public class UserProfileService
{
  public UserProfileService()
  {
  }
}
```

What we normally do at this point is create a new class and a folder structure to support tests related to the `UserProfileService`. Our next test class with be for testing the `GetUserProfile` method, so we will create the folder structure and add that test class.

Folder structure:
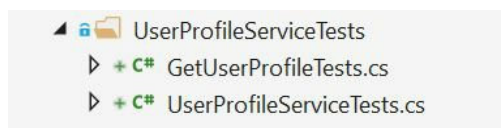
Now write tests for a `GetUserProfile` method.

```
public class GetUserProfileTests
{
  [Fact]
  public void ItReturnsNullForNonExistentUsers()
  {
    // Arrange
    var repository = new FakeRepository<UserProfileDto>();
    var service = new UserProfileService(repository);

    // Act
    var profile = service.GetUserProfile("NonExistantUser@email.com");

    // Assert
    Assert.Null(profile);
  }
```

```
    [Fact]
    public void ItReturnsUserProfileForUsersThatExist()
    {
      // Arrange
      var repository = new FakeRepository<UserProfileDto>();
      var service = new UserProfileService(repository);

      repository.DataSet.Add(new UserProfileDto
      {
        Username = "ExistingUser@email.com"
      });

      // Act
      var profile = service.GetUserProfile("ExistingUser@email.com");

      // Assert
      Assert.NotNull(profile);
      Assert.IsAssignableFrom<UserProfileDto>(profile);
    }
}
```

In this case, we will let you implement the class method that will pass these tests. Remember, we only want to the write a minimal amount of code to pass the tests. You will also want to create tests to verify case insensitivity, if that is something you believe the system needs.

Now that you have a user profile, you need to verify that the password supplied by the user is the correct password. We won't be getting into security concerns too much as part of this book, but you should know that passwords should be a one-way hash. Now, write the test to check the password before you move on to creating a user interface for logging in.

```
public class IsUserPasswordValid
{
  private readonly UserProfileService _service;
  private readonly UserProfileDto _profile;

  public IsUserPasswordValid()
  {
    // Arrange
    var repository = new FakeRepository<UserProfileDto>();
    _service = new UserProfileService(repository);
    _profile = new UserProfileDto
    {
      Username = "ValidUser@email.com",
      // This should be an encryption helper utility. Try to write and
         test a utility to replace this code.
      PasswordHash = SHA512.Create().ComputeHash(Encoding.ASCII.GetBytes("ValidPassword"))
    };

    repository.DataSet.Add(_profile);
  }

  [Fact]
  public void ItReturnsFalseForInvalidPasswords()
  {
    // Act
    var result = _service.IsUserPasswordValid(_profile, "InvalidPassword");

    // Assert
    Assert.False(result);
  }

  [Fact]
  public void ItReturnsTrueForValidPasswords()
  {
    // Act
    var result = _service.IsUserPasswordValid(_profile, "ValidPassword");

    // Assert
```

```
      Assert.True(result);
  }
}

public class UserProfileService
{
  private readonly IRepository<UserProfileDto> _repository;

  public UserProfileService(IRepository<UserProfileDto> repository)
  {
    _repository = repository;
  }

  public object GetUserProfile(string username)
  {
    return _repository.GetAll().FirstOrDefault(u => u.Username == username);
  }

  public bool IsUserPasswordValid(UserProfileDto profile, string password)
  {
    // Now we have the same code in production code as we do in our tests.
    var hash = SHA512.Create().ComputeHash(Encoding.ASCII.GetBytes(password));

    return profile.PasswordHash.SequenceEqual(hash);
  }
}
```

Building a user interface

Now there is enough functionality for you to begin working on your user interface. In a C# web API, the user interface is an API controller. The basic tests needed for an API controller are that it exists and that it inherits correctly from the controller class.

```
public class UserProfileControllerTests
{
  [Fact]
  public void ItExists()
  {
    var controller = new UserProfileController();
  }

  [Fact]
  public void ItIsAController()
  {
    var controller = new UserProfileController();

    Assert.IsAssignableFrom<Controller>(controller);
  }
}

public class UserProfileController : Controller
{
}
```

Next, you need to make sure it has a logon method that accepts a username and password. That same method must also return either a 200 OK or 401 NOT AUTHORIZED, depending on the validity of the user information:

```
public class UserLogon
{
  private readonly UserProfileController _controller;

  public UserLogon()
  {
    // Arrange
    var repository = new FakeRepository<UserProfileDto>();
    var service = new UserProfileService(repository);
    _controller = new UserProfileController(service);

    repository.DataSet.Add(new UserProfileDto
    {
      Username = "TestUser@email.com",
      PasswordHash = SHA512.Create().ComputeHash(Encoding.UTF8.GetBytes("ValidPassword"))
    });
  }

  [Fact]
  public void ItExists()
  {
    // Act
    var response = _controller.LogonUser("TestUser@email.com", "Password");
  }

  [Fact]
  public void ItReturnsAnActionResult()
  {
    // Act
    var response = _controller.LogonUser("TestUser@email.com", "Password");
```

```csharp
    // Assert
    Assert.IsAssignableFrom<IActionResult>(response);
  }

  [Fact]
  public void ItReturnsNotAuthorizedForBadUsername()
  {
    // Act
    var response = (StatusCodeResult) _controller.LogonUser("BadUser@email.com", "ValidPasswor

    // Assert
    Assert.Equal(HttpStatusCode.Unauthorized, (HttpStatusCode)response.StatusCode);
  }

  [Fact]
  public void ItReturnsOkForValidUsernameAndPassword()
  {
    // Act
    var response = (StatusCodeResult)_controller.LogonUser("TestUser@email.com", "ValidPasswor

    // Assert
    Assert.Equal(HttpStatusCode.OK, (HttpStatusCode)response.StatusCode);
  }

  [Fact]
  public void ItReturnsUnauthorizedForInvlalidPassword()
  {
    // Act
    var response = (StatusCodeResult)_controller.LogonUser("TestUser@email.com", "InvalidPassw

    // Assert
    Assert.Equal(HttpStatusCode.Unauthorized, (HttpStatusCode)response.StatusCode);
  }
}

public class UserProfileController : Controller
{
  private readonly UserProfileService _service;

  public UserProfileController(UserProfileService service)
  {
    _service = service;
  }

  public IActionResult LogonUser(string username, string password)
  {
    var user = _service.GetUserProfile(username);

    if (user != null && _service.IsUserPasswordValid(user, password))
    {
      return Ok();
    }

    return Unauthorized();
  }
}
```

One of the downsides to approaching the application in this manner is that now almost all of our layers are concerned with an object that is almost an exact representation of the database. Normally, this is not a real problem. But database tables do change so what if our user profile table needs some touching up in the future? Our entire application will need to be updated at this point. Did you pick up on some of the side-effects of thinking about the application in a back-to-front way? If not, that is okay, but keep an eye open as you explore the two other directional approaches.

Front-to-back

Another way that some developers choose to approach application design and implementation is from a user experience perspective. First, think about how the user would want to interact with the system, then design the system around that concept.

Defining a user interface

To attack the application in this way, first you must determine what you think the best user experience would be. It would probably be best if the user not only got notified whether the logon was accepted, but also received a message explaining to them the current status.

What would you call our controller when testing from this direction? The user is wanting to log on so, you should call it a logon controller.

As before, you need to test that your controller exists. Then test that it properly inherits from the controller.

```
public class LogonControllerTests
{
  [Fact]
  public void ItExists()
  {
    var controller = new LogonController();
  }

  [Fact]
  public void ItIsAnIActionResult()
  {
    // Act
    var controller = new LogonController();

    // Assert
    Assert.IsAssignableFrom<Controller>(controller);
  }
}

public class LogonController : Controller
{
}
```

Now, you can test for your API method. What should it be called? Think again about the user. They are trying to log on so, again, we should probably stick with something simple related to logon. The default post action on this controller should probably be the method used to activate a logon.

```
public class Post
{
  [Fact]
  public void ItExists()
  {
    // Arrange
    var controller = new LogonController();

    // Act
    var response = controller.Post(null);
  }

  [Fact]
  public void ItReturnsAnIActionResult()
  {
    // Arrange
    var controller = new LogonController();

    // Act
```

```csharp
  var response = controller.Post(null);

  // Assert
  Assert.IsAssignableFrom<IActionResult>(response);
}

[Fact]
public void ItReturnsUnauthorizedForInvalidUser()
{
  // Arrange
  var controller = new LogonController();
  var attempt = new LoginAttempt
  {
    Username = "InvalidUser@email.com",
    Password = "BadPassword"
  };

  // Act
  var response = (ObjectResult)controller.Post(attempt);

  // Assert
  Assert.NotNull(response.StatusCode);
  Assert.Equal(HttpStatusCode.Unauthorized, (HttpStatusCode)response.StatusCode);
}

[Fact]
public void ItReturnsOkForValidUser()
{
  // Arrange
  var controller = new LogonController();
  var attempt = new LoginAttempt
  {
    Username = "ValidUser@email.com",
    Password = "ValidPassword"
  };

  // Act
  var response = (ObjectResult)controller.Post(attempt);

  // Assert
  Assert.NotNull(response.StatusCode);
  Assert.Equal(HttpStatusCode.OK, (HttpStatusCode)response.StatusCode);
}

[Fact]
public void ItReturnsUnauthorizedForInvalidPassword()
{
  // Arrange
  var controller = new LogonController();
  var attempt = new LoginAttempt
  {
    Username = "ValidUser@email.com",
    Password = "InvalidPassword"
  };

  // Act
  var response = (ObjectResult)controller.Post(attempt);

  // Assert
  Assert.NotNull(response.StatusCode);
  Assert.Equal(HttpStatusCode.Unauthorized, (HttpStatusCode)response.StatusCode);
}

[Fact]
public void ItReturnsSuccessfulLogonMessageWhenSuccessful()
{
  // Arrange
  var controller = new LogonController();
  var attempt = new LoginAttempt
  {
    Username = "ValidUser@email.com",
    Password = "ValidPassword"
  };
```

```
      // Act
      var response = (ObjectResult)controller.Post(attempt);

      // Assert
      Assert.Equal("Logon Successful", response.Value);
    }

    [Fact]
    public void ItReturnsUnauthorizedLogonMessageWhenUnauthorized()
    {
      // Arrange
      var controller = new LogonController();
      var attempt = new LoginAttempt
      {
        Username = "InvalidUser@email.com",
        Password = "Password"
      };

      // Act
      var response = (ObjectResult)controller.Post(attempt);

      // Assert
      Assert.Equal("Username or Password invalid", response.Value);
    }
}

public class LoginAttempt
{
  public string Username { get; set; }
  public string Password { get; set; }
}

public class LogonController : Controller
{
  [HttpPost]
  public IActionResult Post(LoginAttempt attempt)
  {
    if (attempt != null && attempt.Username == "ValidUser@email.com" && attempt.Password == "\
    {
      return Ok("Logon Successful");
    }

    return new ObjectResult("Username or Password invalid") {
      StatusCode = (int?)HttpStatusCode.Unauthorized
    };
  }
}
```

With the front-to-back directional approach, you don't yet have any of your dependencies defined so you have no choice but to hardcode decisions. You can push those decisions back slightly, though.

Creating a business layer

Create an interface and move your valid user login into a fake logon service for that interface.

```csharp
public class LogonController : Controller
{
  private readonly ILogonService _service;

  public LogonController(ILogonService service)
  {
    _service = service;
  }

  public IActionResult Post(LoginAttempt attempt)
  {
    return _service.IsLogonValid(attempt) ?
      Ok("Logon Successful") :
      new ObjectResult("Username or Password invalid") {
        StatusCode = (int?)HttpStatusCode.Unauthorized
      };
  }
}

public interface ILogonService
{
  bool IsLogonValid(LoginAttempt attempt);
}

class FakeLogonService : ILogonService
{
  public bool IsLogonValid(LoginAttempt attempt)
  {
    return attempt != null &&
      attempt.Username == "ValidUser@email.com" &&
      attempt.Password == "ValidPassword";
  }
}
```

If you are following along, you will need to update all the controller references in the tests to use this new fake logon service.

Now that you have an interface defined, you can write tests to create a service layer.

```csharp
public class IsValidLogon
{
  private readonly LogonService _service;

  public IsValidLogon()
  {
    var repository = new FakeRepository<UserLogonDto>();
    _service = new LogonService(repository);
    var userLogon = new UserLogonDto
    {
      Username = "ValidUser@email.com",
      PasswordHash =  SHA512.Create().ComputeHash(Encoding.ASCII.GetBytes("ValidPassword"))
    };

    repository.DataSet.Add(userLogon);
  }

  [Fact]
  public void ItExists()
  {
```

```csharp
      var repository = new FakeRepository<UserLogonDto>();
      var service = new LogonService(repository);
      var attempt = new LoginAttempt();

      service.IsLogonValid(attempt);
    }

    [Fact]
    public void ItReturnsTrueForValidAttempt()
    {
      // Arrange
      var attempt = new LoginAttempt
      {
        Username = "ValidUser@email.com",
        Password = "ValidPassword"
      };

      // Act
      var result = _service.IsLogonValid(attempt);

      // Assert
      Assert.True(result);
    }

    [Fact]
    public void ItReturnsFalseForInvalidUsername()
    {
      // Arrange
      var attempt = new LoginAttempt
      {
        Username = "InvalidUser@email.com",
        Password = "ValidPassword"
      };

      // Act
      var result = _service.IsLogonValid(attempt);

      // Assert
      Assert.False(result);
    }

    [Fact]
    public void ItReturnsFalseForInvalidPassword()
    {
      // Arrange
      var attempt = new LoginAttempt
      {
        Username = "ValidUser@email.com",
        Password = "InvalidPassword"
      };

      // Act
      var result = _service.IsLogonValid(attempt);

      // Assert
      Assert.False(result);
    }
}

public class LogonService : ILogonService
{
  private readonly IRepository<UserLogonDto> _repository;

  public LogonService(IRepository<UserLogonDto> repository)
  {
    _repository = repository;
  }

  public bool IsLogonValid(LoginAttempt attempt)
  {
    attempt = attempt ?? new LoginAttempt();

    var user = _repository.GetAll().FirstOrDefault(u => u.Username == attempt.Username);
```

```
      var hash = SHA512.Create().ComputeHash(Encoding.ASCII.GetBytes(attempt.Password ?? ""));

      return user != null && user.PasswordHash.SequenceEqual(hash);
  }
}

public class UserLogonDto : IIdentity
{
  public int Id { get; set; }
  public string Username { get; set; }
  public byte[] PasswordHash { get; set; }
}
```

Building a data source

Now that you have a service, you can focus on the data layer. Believe it or not, this part is not really any different from what we did at this stage for the back-to-front approach.

We have managed to do one thing differently. We have created a contract for our data interaction. The rest of the table, if we are using a relational data store, could be anything and we don't care. We only care about the username and password hash. We only have ID because the `FakeRepository` requires it.

> *There are ways to program the repository that do not require this feature. We are not going to recreate the table from the previous example. It is the same table.*

Inside out

The last directional approach that we are going to cover in this chapter is the inside-out approach. With the inside-out approach, you begin, not with the UI or the data source, but instead with the business rules defined in the requirements.

Defining a business layer

Looking back at our requirements, we can build tests and logic that are a one-to-one match for our requirements such as:

- Given a registered speaker
- And given an invalid username
- When attempting login
- Then an INVALID_USERNAME_OR_PASSWORD error occurs

```
public class LoginTests
{
  [Fact]
  public void GivenAnInvalidUsername()
  {
    // Arrange/Given
    var username = "InvalidUser@email.com";

    // Act/When
    var exception = Record.Exception(() => Account.Logon(username));

    // Assert/Then
    Assert.IsAssignableFrom<InvalidUsernameOrPasswordException>(exception);
    Assert.Equal("Invalid Username or Password", exception.Message);
  }
}
```

```
public class InvalidUsernameOrPasswordException: Exception
{
  public InvalidUsernameOrPasswordException() : base("Invalid Username or Password")
  {
  }
}
```

```
public class Account
{
  public object Logon(string username)
  {
    throw new InvalidUsernameOrPasswordException();
  }
}
```

Some significant changes are made by the next requirement in order to provide some latitude going forward.

- Given a registered speaker
- And given a valid username
- And given a valid password
- When attempting login
- Then the user is granted access to the application

```
public class LoginTests
{
  private readonly string _accessKey;
  private readonly Account _account;

  public LoginTests()
  {
```

```csharp
      _accessKey = "GrantedAccessKey";
      var repository = new FakeRepository<UserCredentials>();
      _account = new AccountTestDouble(repository);

      repository.DataSet.Add(new UserCredentials {
        Username = "ValidUser@email.com"
      });
    }

    [Fact]
    public void GivenAnInvalidUsername()
    {
      // Arrange/Given
      var username = "InvalidUser@email.com";
      var password = "UnimportantPassword";

      // Act/When
      var exception = Record.Exception(() => _account.Logon(username, password));

      // Assert/Then
      Assert.IsAssignableFrom<InvalidUsernameOrPasswordException>(exception);
      Assert.Equal("Invalid Username or Password", exception.Message);
    }

    [Fact]
    public void GivenAValidUsernameAndPassword()
    {
      // Arrange/Given
      var username = "ValidUser@email.com";
      var password = "ValidPassword";

      // Act/When
      var result = _account.Logon(username, password);

      // Assert/Then
      Assert.IsAssignableFrom<string>(result);
      Assert.Equal(_accessKey, result);
    }
}

public class InvalidUsernameOrPasswordException : Exception
{
  public InvalidUsernameOrPasswordException() : base("Invalid Username or Password")
  {
  }
}

public class Account
{
  private readonly IRepository<UserCredentials> _repository;

  public Account(IRepository<UserCredentials> repository)
  {
    _repository = repository;
  }

  public string Logon(string username, string password)
  {
    var uc =_repository.GetAll().FirstOrDefault(u => u.Username == username);

    if (uc == null)
    {
      throw new InvalidUsernameOrPasswordException();
    }

    return GenerateAccessKey(uc);
  }

  protected virtual string GenerateAccessKey(UserCredentials userCredentials)
  {
    // Here we would need to actually generate an access token
    return "DefaultKey";
  }
```

```
  }

public class AccountTestDouble : Account
{
  public AccountTestDouble(IRepository<UserCredentials> repository) : base(repository) { }

  protected override string GenerateAccessKey(UserCredentials userCredentials)
  {
    return "GrantedAccessKey";
  }
}

public class UserCredentials : IIdentity
{
  public int Id { get; set; }
  public string Username { get; set; }
}
```

Now for the last requirement that we were provided:

- Given a registered speaker
- And given a valid username
- And given an invalid password
- When attempting login
- Then an INVALID_USERNAME_OR_PASSWORD error occurs

```
[Fact]
public void GivenAnInvalidPassword()
{
  // Arrange/Given
  var username = "ValidUser@email.com";
  var password = "InvalidPassword";

  // Act/When
  var exception = Record.Exception(() => _account.Logon(username, password));

  // Assert/Then
  Assert.IsAssignableFrom<InvalidUsernameOrPasswordException>(exception);
  Assert.Equal("Invalid Username or Password", exception.Message);
}
```

This last test was quite simple and closely resembles the first test we wrote for inside-out development. One thing to note, but which we are not showing here, is that we had to extend our UserCredentials class with the password hash property.

Creating the user interface and data layers from this point is almost exactly like what we have shown in the earlier examples, so we will not show them here.

The tasks left for this example are abstracting the business layer behind an interface, using the business object in the UI, and creating the appropriate data configuration for the data layer.

Summary

In this chapter, we've defined the Speaker Meet application in more detail. Architectural choices were discussed and a path has been set. Epics, features, and user stories have been covered in enough detail that we're now ready to take the next steps with the Speaker Meet application.

In Chapter 7, *Test Driving C# Applications,* we'll focus on test driving the C# API. Topics such as *fakes, stubs,* and *mocks* will be introduced to help you navigate the testing world.

Test-Driving C# Applications

The two most important features for the Speaker Meet application were determined to be the speaker listing and the ability to see an individual speaker's details. The speaker listing and speaker details will deliver the most value for our Minimum Viable Product.

Conference organizers, user group administrators, and the general public would likely care most about finding information on speakers. With that in mind, the speakers epic is where development for the Speaker Meet application begins.

In this chapter, we cover:

- Speaker Meet requirements
- API, service, and repository tests
- The speaker detail and speaker listing APIs

Reviewing the requirements

In order to get started, the foundation of the speaker section of the Speaker Meet application is laid by defining the initial set of requirements. These will help eliminate ambiguity and develop a common understanding of the requirements, as well as defining a common vocabulary used throughout the project.

The abstract is where a projects, purpose and value can be presented. Any project, before it can be approved to be worked on, must prove the value that it can provide to the company. This is true whether you are working for a Fortune 500 company or a startup with two people.

A data dictionary is important because it provides a common, ubiquitous language for the project. The term, ubiquitous language, is from Domain Driven Design and denotes *a shared or common language*. The idea is that the shared jargon of the business and development team is solidified in a codex that can be viewed and used by all.

Last, and certainly not least, the requirements must be presented in an agreed upon format. The specific format is less important than the format agreement. Regardless of the format, good requirements provide a context of interaction, the interaction taking place, and the expected results given the context and specific action.

Speaker listing

The speakers section of the Speaker Meet website contains a listing of all speakers in the system. The listing of speakers will present value to multiple groups including conference and user group organizers as well as conference and user group attendees. From a user interaction perspective, the speaker listing allows entry to the speaker details. The speaker details are where the real value is delivered in the form of availability, upcoming engagements, and contact information for a given speaker.

Initially, the speaker listing will aid organizers by providing quick access to speaker discovery. Organizers will be able to find speakers they know of and discover speakers of whom they are unaware. Once found or discovered, the organizer will be able to view details for specific speakers and, eventually, organizers will be able to contact the speakers using the available contact information.

Attendees will benefit from the speaker list in a similar fashion to the organizers. Attendees have one important difference, however: they are looking for the events a speaker is already attached to as a presenter. This information, similar to the contact information, will be available in the speaker details.

API

The API is the main gateway into the core system of the Speaker Meet application. The speaker listing API should return a listing of speaker summary ViewModels. These ViewModels contain only the information necessary for this portion of the application. The ViewModels represent the speaker, but should not necessarily be direct copies of the speaker objects persisted to a database.

The `SpeakerSummary` ViewModel will be defined based on the requirements of the system. This ViewModel will grow to contain only the properties required for its limited use.

To get started, a new method will need to be added to an API. For the first new piece of functionality to be added, a new method `GetAll` will need to be created in the `SpeakerController`. But first, a test must be created.

API tests

To revisit, code in the `SpeakerController` may not be written without a failing unit test. To begin, a new test file should be created named `GetAll`. This is where all the tests associated with the `GetAll` method of the `SpeakerController` will be contained.

> *There is duplication in how testing the `SpeakerController` is set up. Try to come up with ways that this duplication can be minimized.*

The first such test should be the standard `ItExists` test. Building on the example from previous chapters, the `SpeakerController` accepts an `ISpeakerService` in the constructor. The same method of providing a `Moq` object can be applied here as well.

```
[Fact]
public void ItExists()
{
  // Arrange
  var speakerServiceMock = new Mock<ISpeakerService>();
  var controller = new SpeakerController(speakerServiceMock.Object);

  // Act
  controller.GetAll();
}
```

Comparing this first test with the first test written for the `Search` method in the `SpeakerController`, you may notice there's a bit of duplication happening already. Remember, duplication should be avoided. Don't forget the acronym, **DRY** (**Don't Repeat Yourself**).

In order to make this first test pass, a void `GetAll` method should be added to the `SpeakerController`. This will allow the application to compile, thereby passing this test. Remember, a failure to compile is a failing test.

```
public void GetAll()
{
}
```

Next, ensure that the `GetAll` method of `SpeakerController` returns an `OkObjectResult` by creating a new test. Don't worry about the type of the result itself. That will be covered by the next test.

```
[Fact]
public void ItReturnsOkObjectResult()
{
  // Arrange
  var speakerServiceMock = new Mock<ISpeakerService>();
  var controller = new SpeakerController(speakerServiceMock.Object);

  // Act
  var result = controller.GetAll();

  // Assert
  Assert.NotNull(result);
  Assert.IsType<OkObjectResult>(result);
}
```

In order to get this test to pass, the method should return an `IActionResult` instead of `void`. The method should also be changed to return `Ok()` in order to make the test pass. The method does not need to return anything else in order to make the test pass as written. Do not write more code than is required to make the test pass.

```
public IActionResult GetAll()
{
  return Ok();
}
```

Now, determine that the method returns a collection of `SpeakerSummary`.

```
[Fact]
public void ItReturnsCollectionOfSpeakerSummary()
{
  // Arrange
  var speakerServiceMock = new Mock<ISpeakerService>();
  var controller = new SpeakerController(speakerServiceMock.Object);

  // Act
  var result = controller.GetAll() as OkObjectResult;

  // Assert
  Assert.NotNull(result);
  Assert.NotNull(result.Value);
  Assert.IsAssignableFrom<IEnumerable<SpeakerSummary>>(result.Value);
}
```

Create a `SpeakerSummary` class to fulfill the requirement defined by this test. Give some thought to where the new `SpeakerSummary` lives. This is a ViewModel that will need to be accessed by the tests, but should not be available to the other layers of the application. More about proper separation in a future chapter.

Modify the `GetAll` method of the `SpeakerController` to return a set of `SpeakerSummary` objects as the return value.

```
public IActionResult GetAll()
{
  return Ok(new List<SpeakerSummary>());
}
```

Moq

In previous chapters, Moq was used to provide a stand-in set of functionalities for the item under test. The results provided for the mocked instance were required, but the implementation was not vital to what was being tested.

Like the examples in previous chapters, the logic for `GetAll` should not be found in the controller itself. Instead, the logic will be contained within the business layer, specifically the `SpeakerService` implementation of `ISpeakerService`. When the `GetAll` method in `SpeakerController` is called it is expected that the `GetAll` method of the `SpeakerService` will be called.

The `GetAll` method does not exist within the `SpeakerService`, so the following test should fail.

```
[Fact]
public void ItCallsGetAllServiceOnce()
{
  // Arrange
  var speakerServiceMock = new Mock<ISpeakerService>();
  var controller = new SpeakerController(_speakerServiceMock.Object);

  // Act
  controller.GetAll();

  // Assert
  speakerServiceMock.Verify(mock => mock.GetAll(), Times.Once());
}
```

Creating the previous test has forced the creation of a new method signature in the `ISpeakerService` interface. The following method signature should be added to the `ISpeakerService` interface.

```
IEnumerable<SpeakerSummary> GetAll();
```

To get the application to compile, `GetAll` will also need to be added to the `SpeakerService` class. For now, this should throw an exception.

```
public IEnumerable<SpeakerSummary> GetAll()
{
  throw new NotImplementedException();
}
```

To get the `ItCallsGetAllServiceOnce` test to pass, make sure that the `GetAll` method of the `SpeakerService` is called. The return value from the call is not yet needed for the test to pass, so simply calling the method is all that is required.

```
public IActionResult GetAll()
{
  _speakerService.GetAll();

  return Ok(new List<SpeakerSummary>());
}
```

Note that this will make the test pass, but it is not exactly the correct solution yet. A new test is required to force the code to do something with the return value of the service. Moving on, it's

time to do something with the result of the `SpeakerService.GetAll` call.

```
[Fact]
public void GivenSpeakerServiceThenResultsReturned()
{
  // Arrange
  var speakers = new List<SpeakerSummary> { new SpeakerSummary
  {
    Name = "Speaker"
  } };

  var speakerServiceMock = new Mock<ISpeakerService>();
  speakerServiceMock.Setup(x => x.GetAll()).Returns(() => _speakers);

  var controller = new SpeakerController(speakerServiceMock.Object);

  // Act
  var result = controller.GetAll() as OkObjectResult;
  var speakers = ((IEnumerable<SpeakerSummary>)result.Value).ToList();

  // Assert
  Assert.Equal(_speakers, speakers);
}
```

Don't forget to refactor the tests as well as the code. For readability, the `Arrange` methods have been included in the previous examples. Likely, these would be extracted and defined as *fields* and assigned in the constructor.

```
private readonly SpeakerController _controller;
private static Mock<ISpeakerService> _speakerServiceMock;
private readonly List<SpeakerSummary> _speakers;

public GetAll()
{
  _speakers = new List<SpeakerSummary> { new SpeakerSummary
  {
    Name = "test"
  } };

  _speakerServiceMock = new Mock<ISpeakerService>();
  _speakerServiceMock.Setup(x => x.GetAll()).Returns(() => _speakers);

  _controller = new SpeakerController(_speakerServiceMock.Object);
}
```

Testing exception cases

In the event that a speaker is requested that does not exist, it would be best to return a friendly error message to the consumer of the API.

```
[Fact]
public void GivenSpeakerNotFoundExceptionThenNotFoundObjectResult()
{
  // Arrange
  // Act
  var result = _controller.Get(-1);

  // Assert
  Assert.IsAssignableFrom<NotFoundObjectResult>(result);
}
```

Create a new exception class named `SpeakerNotFoundException`. This will be the specific exception returned by the `Moq` call below. Like the `SpeakerSummary` class file before, give some thought to where the `SpeakerNotFoundException` class file should be saved.

```
public class SpeakerNotFoundException : Exception
{
}
```

"Throwing" a new exception when a specific ID is supplied requires a little bit of setup in `Moq`. This is similar to what was already defined by the `x.Get(It.IsAny<int>)` definition.

```
_speakerServiceMock.Setup(x => x.Get(-1)).Returns(() => throw new SpeakerNotFoundException());
```

Make sure this is added after the previous setup, as `Moq` will process the last value first. Avoid a false positive by understanding how `Moq` will evaluate what has been set up within its context.

Next, modify the `Get` method of the controller to catch the exception and return the proper response code.

```
public IActionResult Get(int id)
{
  try
  {
    var speaker = _speakerService.Get(id);
    return Ok(speaker);
  }
  catch (SpeakerNotFoundException)
  {
    return NotFound();
  }
}
```

The initial requirements stated that a friendly error message be returned to the client. Create a test that ensures a friendly message is returned to the consumer in the event a speaker is not found with the supplied ID.

```
[Fact]
public void GivenSpeakerNotFoundExceptionThenMessageReturned()
{
  // Arrange
```

```
  // Act
  var result = _controller.Get(-1) as NotFoundObjectResult;

  // Assert
  Assert.NotNull(result);
  Assert.Equal("Speaker Not Found", result.Value);
}
```

In order to make this test pass, the `SpeakerNotFoundException` class must be modified in order to return a friendly error message.

```
public class SpeakerNotFoundException : Exception
{
  public SpeakerNotFoundException() : base("Speaker Not Found")
  {
  }
}
```

And finally, modify the `Get` method in the controller to return the message.

```
public IActionResult Get(int id)
{
  try
  {
    var speaker = _speakerService.Get(id);
    return Ok(speaker);
  }
  catch (SpeakerNotFoundException ex)
  {
    return NotFound(ex.Message);
  }
}
```

Service

The business logic for the `GetAll` method should be housed in the `SpeakerService`. As before, in order to write a line of code a test must first be written.

Service tests

To build on the previous example, start with an `ItExists` test.

```
[Fact]
public void ItHasGetAllMethod()
{
  var speakerService = new SpeakerService();
  speakerService.GetAll();
}
```

Since this method was previously added to the `SpeakerService`, although with a `NotImplementedException`, it would be best to see this test fail for the proper reason. Delete the `GetAll` method from the `SpeakerService` so that the application will fail to compile. Now, add the method back to see that the application once again compiles, and therefore this test passes. This time, have the method return `null` instead of throwing a new `NotImplementedException`.

```
public IEnumerable<SpeakerSummary> GetAll()
{
  return null;
}
```

Now, ensure that the `GetAll` method returns a collection of `SpeakerSummary` by creating a new test.

```
[Fact]
public void ItReturnsCollectionOfSpeakerSummary()
{
  // Arrange
  // Act
  var speakers = _speakerService.GetAll();

  // Assert
  Assert.NotNull(speakers);
  Assert.IsAssignableFrom<IEnumerable<SpeakerSummary>>(speakers);
}
```

Modify the `GetAll` method of the `SpeakerService` in order to make this test pass. The minimum amount of code required to make this test pass involves returning a `new List` of `SpeakerSummary` objects. Do not add more code than is required to make this test pass.

```
public IEnumerable<SpeakerSummary> GetAll()
{
  return new List<SpeakerSummary>();
}
```

Building on the examples from a previous chapter, use the hardcoded data from before. Extract `hardCodedSpeakers` into a field in order to use the data in both the `Search` method as well as the `GetAll` method:

```
public readonly List<Speaker> HardCodedSpeakers = new List<Speaker>
{
  new Speaker {Name = "Josh"},
  new Speaker {Name = "Joshua"},
  new Speaker {Name = "Joseph"},
  new Speaker {Name = "Bill"}
};
```

Note that the field was made public. This will allow tests to use this data for comparison for *Asserts*. Don't worry, this field and the hardcoded data contained therein will be short-lived. Once these are no longer needed they can be safely deleted.

Now, create a test to ensure that all of the data contained in `HardCodedSpeakers` is returned by the `GetAll` method in the `SpeakerService`. Start by verifying that the same number of speakers in the hardcoded data is returned by the method.

```
[Fact]
public void ItReturnsAllSpeakers()
{
  // Arrange
  // Act
  var speakers = _speakerService.GetAll();

  // Assert
  Assert.NotNull(speakers);
  Assert.IsAssignableFrom<IEnumerable<SpeakerSummary>>(speakers);
  Assert.Equal(_speakerService.HardCodedSpeakers.Count, speakers.Count());
}
```

To get this to pass, simply iterate over the hardcoded values and return a new `SpeakerSummary` for each entry. As the test is not yet checking the values of the speakers returned, all that is required is that the proper count of `SpeakerSummary` objects is returned.

```
public IEnumerable<SpeakerSummary> GetAll()
{
  return HardCodedSpeakers.Select(speaker => new SpeakerSummary());
}
```

Now, ensure that the speakers are properly converted to `SpeakerSummary` objects. First, check that the `Name` properties are the same.

```
[Fact]
public void ItReturnsAllSpeakersWithName()
{
  // Arrange
  // Act
  var speakers = _speakerService.GetAll().ToList();

  // Assert
  Assert.NotNull(speakers);
  Assert.IsAssignableFrom<IEnumerable<SpeakerSummary>>(speakers);

  for (var i = 0; i < speakers.Count; i++)
  {
    Assert.NotNull(_speakerService.HardCodedSpeakers[i].Name);
    Assert.Equal(_speakerService.HardCodedSpeakers[i].Name, speakers[i].Name);
  }
}
```

And now, make this test pass by assigning the `Name` within the `GetAll` method.

```
public IEnumerable<SpeakerSummary> GetAll()
{
  return HardCodedSpeakers.Select(speaker => new SpeakerSummary
  {
    Name = speaker.Name
  });
}
```

Continue to build up the `SpeakerSummary` object with the required properties. The `Name` property has been added. Now, add an ID and ensure that it is being assigned and returned properly.

```
[Fact]
public void ItReturnsAllSpeakersWithId()
{
  // Arrange
  // Act
  var speakers = _speakerService.GetAll().ToList();

  // Assert
  Assert.NotNull(speakers);
  Assert.IsAssignableFrom<IEnumerable<SpeakerSummary>>(speakers);

  for (var i = 0; i < speakers.Count; i++)
  {
    Assert.NotNull(_speakerService.HardCodedSpeakers[i].Id);
    Assert.Equal(_speakerService.HardCodedSpeakers[i].Id, speakers[i].Id);
  }
}
```

In order to make this pass, an ID will need to be mapped in the `GetAll` method of the `SpeakerService`, and an ID property added to the `Speaker` and `SpeakerSummary` objects.

```
public IEnumerable<SpeakerSummary> GetAll()
{
  return HardCodedSpeakers.Select(speaker => new SpeakerSummary
  {
    Id = speaker.Id,
    Name = speaker.Name
  });
}
```

Next, add a `Location` to be returned by the `GetAll` method. This, too, will require the `Speaker` and `SpeakerSummary` objects to be modified. Give the new Location property in the `HardCodedSpeakers` collection distinct values to ensure that the values are being returned properly.

```
[Fact]
public void ItReturnsAllSpeakersWithLocation()
{
  // Arrange
  // Act
  var speakers = _speakerService.GetAll().ToList();

  // Assert
  Assert.NotNull(speakers);
  Assert.IsAssignableFrom<IEnumerable<SpeakerSummary>>(speakers);

  for (var i = 0; i < speakers.Count; i++)
  {
    Assert.NotNull(_speakerService.HardCodedSpeakers[i].Location);
    Assert.Equal(_speakerService.HardCodedSpeakers[i].Location, speakers[i].Location);
  }
}
```

Add some locations to the hardcoded data.

```
public readonly List<Speaker> HardCodedSpeakers = new List<Speaker>
{
  new Speaker {Id = 1, Name = "Josh", Location = "Tampa, FL"},
  new Speaker {Id = 2, Name = "Joshua", Location = "Louisville, KY"},
  new Speaker {Id = 3, Name = "Joseph", Location = "Las Vegas, NV"},
  new Speaker {Id = 4, Name = "Bill", Location = "New York, NY"},
};
```

Finally, map the location to the `SpeakerSummary` ViewModel.

```
public IEnumerable<SpeakerSummary> GetAll()
{
  return HardCodedSpeakers.Select(speaker => new SpeakerSummary
```

```
  {
    Id = speaker.Id,
    Name = speaker.Name,
    Location = speaker.Location,
  });
}
```

As has been discussed before, tests should have a single action. That does not preclude them
from having multiple asserts. In order to minimize duplication, the property tests should be
collapsed.

Clean tests

A test suite should be well maintained. This is the first consumer of the application and provides the most comprehensive documentation of the functionality of the system. To clean up the tests that were just created, it is time to do some refactoring.

Collapse the `SpeakerSummary` properties into single act, with multiple asserts. This will help to make the test suite smaller, easier to read and maintain, and quite possibly it will execute execute more quickly. A test suite that executes quickly is far more likely to be run often by the developers.

Rename `ItReturnsAllSpeakersWithName` to `ItReturnsAllSpeakersWithProperties` and collapse the `ID` and `Location` tests into this one.

```
[Fact]
public void ItReturnsAllSpeakersWithProperties()
{
  // Arrange
  // Act
  var speakers = _speakerService.GetAll().ToList();

  // Assert
  Assert.NotNull(speakers);
  Assert.IsAssignableFrom<IEnumerable<SpeakerSummary>>(speakers);

  for (var i = 0; i < speakers.Count; i++)
  {
    Assert.NotNull(_speakerService.HardCodedSpeakers[i].Name);
    Assert.Equal(_speakerService.HardCodedSpeakers[i].Name, speakers[i].Name);
    Assert.NotNull(_speakerService.HardCodedSpeakers[i].Id);
    Assert.Equal(_speakerService.HardCodedSpeakers[i].Id, speakers[i].Id);
    Assert.NotNull(_speakerService.HardCodedSpeakers[i].Location);
    Assert.Equal(_speakerService.HardCodedSpeakers[i].Location, speakers[i].Location);
  }
}
```

Repository

In a previous chapter, the data was hard-coded within the `SpeakerController` class. The data has since moved to a hardcoded collection in the `SpeakerService`. Ultimately the data will be persisted in a database. For now, moving the data out of the `SpeakerService` will be enough.

A repository layer will be used to separate the data access layer from the rest of the application. To achieve this, a repository must be introduced. In order for a repository to be created, a need must be established. Start slowly by requiring the `SpeakerService` to accept an `IRepository`.

```
[Fact]
public void ItAcceptsIRepository()
{
  // Arrange
  IRepository fakeRepository = new FakeRepository();

  // Act
  var service = new SpeakerService(fakeRepository);

  // Assert
  Assert.NotNull(service);
}
```

This, of course, will cause the application to fail to compile. Create an `IRepository` interface, a `FakeRepository` class, and modify the `SpeakerService` to accept an `IRepository`.

```
public SpeakerService(IRepository repository)
{
}
```

The IRepository interface

The `IRepository` interface will be where the method signatures for interacting with the data access layer will be defined. This interface will be grown slowly, guided by tests. In Chapter 8, *Abstract Away Problems*, more details will be provided and additional concepts will be introduced. For now, the interface will merely be a contract for the `FakeRepository` used for the `SpeakerService` tests.

FakeRepository

Now that the `FakeRepository` has been created, the `HardCodedSpeakers` can be moved into the `FakeRepository`. First, several iterative tests need to be created.

Interacting with a `FakeRepository` of your own creation allows you to substitute values and create additional functionality for testing purposes.

```
[Fact]
public void ItCallsRepository()
{
  // Arrange
  FakeRepository fakeRepository = new FakeRepository();
  var service = new SpeakerService(fakeRepository);

  // Act
  var speakers = service.GetAll();

  // Assert
  Assert.True(fakeRepository.GetAllCalled);
}
```

By introducing a public field, the same functionality seen with `Moq` can be applied here in the `FakeRepository`.

```
public bool GetAllCalled { get; private set; }

public void GetAll()
{
  GetAllCalled = true;
}
```

Now ensure that the `FakeRepository` returns the `HardCodedSpeakers` when `GetAll` is called by modifying the existing tests for `ItReturnsAllSpeakers` and `ItReturnsAllSpeakersWithProperties`.

```
[Fact]
public void ItReturnsAllSpeakers()
{
  // Arrange
  // Act
  var speakers = _speakerService.GetAll();

  // Assert
  Assert.NotNull(speakers);
  Assert.IsAssignableFrom<IEnumerable<SpeakerSummary>>(speakers);
  Assert.Equal(_fakeRepository.HardCodedSpeakers.Count, speakers.Count());
}
[Fact]
public void ItReturnsAllSpeakersWithProperties()
{
  // Arrange
  // Act
  var speakers = _speakerService.GetAll().ToList();

  // Assert
  Assert.NotNull(speakers);
  Assert.IsAssignableFrom<IEnumerable<SpeakerSummary>>(speakers);

  for (var i = 0; i < speakers.Count; i++)
```

```
  {
    Assert.NotNull(_fakeRepository.HardCodedSpeakers[i].Name);
    Assert.Equal(_fakeRepository.HardCodedSpeakers[i].Name, speakers[i].Name);
    Assert.NotNull(_fakeRepository.HardCodedSpeakers[i].Id);
    Assert.Equal(_fakeRepository.HardCodedSpeakers[i].Id, speakers[i].Id);
    Assert.NotNull(_fakeRepository.HardCodedSpeakers[i].Location);
    Assert.Equal(_fakeRepository.HardCodedSpeakers[i].Location, speakers[i].Location);
  }
}
```

It may seem like a lot of effort has been expended to just kick the can down the road. This has all been necessary effort to successfully work towards a truly functional and maintainable application. However, there's still more work to be done.

Using factories with the FakeRepository

So far this has been a relatively straight-forward exercise. The `Speaker` class represents the shape of the object that will be persisted to the database. The `HardCodedSpeakers` collection represents the entire set of *speakers* from a database.

It's not entirely ideal to have or maintain a set of hardcoded data, whether it's in a test file or not. It would be far more flexible to provide a way for the test writer to define the data with which to test.

Using a factory to create speakers and add them to the `FakeRepository` provides a much cleaner and easier-to-maintain way of managing the state of the tests that require specific data scenarios.

```
public static class SpeakerFactory
{
  public static Speaker Create(FakeRepository fakeRepository, int id = 1, string name = "Joshu
  {
    var speaker = new Speaker
    {
      Id = id,
      Name = name,
      Location = location
    };

    fakeRepository.Speakers.Add(speaker);

    return speaker;
  }
}
```

Note that default values defined for id, name, and location have been provided. This allows the user to supply specific values if they want, or proceed without the need for supplying them.

The `FakeRepository` must also be modified to remove the `HardCodedSpeakers` and expose a public collection of speakers.

```
public class FakeRepository : IRepository
{
  public List<Speaker> Speakers = new List<Speaker>();
  public bool GetAllCalled { get; private set; }

  public IEnumerable<Speaker> GetAll()
  {
    GetAllCalled = true;

    return Speakers;
  }
}
```

Now, for each test a specific set of data can be provided with which to test. All that is required is that the factory be called to create one or more speakers to add to the `FakeRepository`.

```
public GetAll()
{
  _fakeRepository = new FakeRepository();
  SpeakerFactory.Create(_fakeRepository);
  _speakerService = new SpeakerService(_fakeRepository);
}
```

If you have been following along with the same solution from previous chapters, you may need to modify the Search tests as well.

```
public Search()
{
  var fakeRepository = new FakeRepository();
  SpeakerFactory.Create(fakeRepository);
  SpeakerFactory.Create(fakeRepository, name:"Josh");
  SpeakerFactory.Create(fakeRepository, name:"Joseph");
  SpeakerFactory.Create(fakeRepository, name:"Bill");
  _speakerService = new SpeakerService(fakeRepository);
}
```

Soft delete

It was decided that it would be useful to be able to "soft delete" a speaker from the system. A "soft delete" allows for the record to be marked as deleted without physically deleting the record. This will help maintain referential integrity while achieving the desired result.

First, add an extension method to the SpeakerFactory called `IsDeleted` that will set the speaker to be deleted.

```
public static Speaker IsDeleted(this Speaker speaker)
{
  speaker.IsDeleted = true;
  return speaker;
}
```

Now, create a test to ensure that this speaker is not returned when `GetAll` is called.

```
[Fact]
public void GivenSpeakerIsDeletedSpeakerIsNotReturned()
{
  // Arrange
  var fakeRepository = new FakeRepository();
  SpeakerFactory.Create(fakeRepository).IsDeleted();
  var speakerService = new SpeakerService(fakeRepository);

  // Act
  var speakers = speakerService.GetAll().ToList();

  // Assert
  Assert.NotNull(speakers);
  Assert.IsAssignableFrom<IEnumerable<SpeakerSummary>>(speakers);
  Assert.Equal(0, speakers.Count);
}
```

Finally, modify the code to guarantee the "deleted" speaker is not returned.

```
public IEnumerable<SpeakerSummary> GetAll()
{
  return _repository.GetAll()
    .Where(x => !x.IsDeleted)
    .Select(speaker => new SpeakerSummary
     {
       Id = speaker.Id,
       Name = speaker.Name,
       Location = speaker.Location
     });
}
```

Speaker details

Next up we come to the speaker details. We've chosen to continue in the back-end application as we'll tie the entire program together in up coming chapters.

As stated earlier, this is where the real value is delivered for the first set of requirements. User groups and conference organizers will be able to contact a speaker using the information provided in the details view.

API

To return the details of an individual speaker, a new endpoint is needed. A new method `Get` is required which will take an integer ID and return a `SpeakerDetail` ViewModel.

API tests

To get started, add a new test class named `Get`. Now, add a test to check that the `Get` method exists.

```
[Fact]
public void ItExists()
{
  // Arrange
  var speakerServiceMock = new Mock<ISpeakerService>();
  var controller = new SpeakerController(speakerServiceMock.Object);

  // Act
  var result = controller.Get();
}
```

Make this test pass by adding a `Get` method to the `SpeakerController`. Note that, in the following example, the `Arrange` test setup has been moved to the constructor of the test class.

```
[Fact]
public void ItExists()
{
  // Arrange
  // Act
  _controller.Get();
}
```

Next, ensure that the `Get` method accepts an integer.

```
[Fact]
public void ItAcceptsInteger()
{
  // Arrange
  // Act
  _controller.Get(1);
}
```

In order to make this test pass, an integer parameter will need to be added to the `Get` method. At this time, it is safe to delete the `ItExists` method. This test would need to be modified to accommodate the change, and its existence would be verified with the new test.

```
public void Get(int id)
{
}
```

Now that the tests confirm that the `Get` method accepts an integer, now confirm that it returns an `Ok` result.

```
[Fact]
public void ItReturnsOkObjectResult()
{
  // Arrange
  // Act
  var result = _controller.Get(1);

  // Assert
  Assert.IsType<OkObjectResult>(result);
}
```

Now, ensure that the result is a `SpeakerDetail`.

```
[Fact]
public void ItReturnsSpeakerDetail()
{
  // Arrange
  // Act
  var result = _controller.Get(1) as OkObjectResult;

  // Assert
  Assert.NotNull(result);
  Assert.NotNull(result.Value);
  Assert.IsType<SpeakerDetail>(result.Value);
}
```

In order to get this test to pass, a `SpeakerDetail` object is required. Create an empty object with no properties, as none are yet required by the tests.

```
public IActionResult Get(int id)
{
  return Ok(new SpeakerDetail());
}
```

Just like with the `GetAll` method, the logic for this action should reside in the *Service.* Create a test to check that the `Get` method in the `SpeakerService` is called using `Moq`.

```
[Fact]
public void ItCallsGetServiceOnce()
{
  // Arrange
  // Act
  _controller.Get(1);

  // Assert
  _speakerServiceMock.Verify(mock => mock.Get(), Times.Once());
}
```

To get the application to compile a `Get` method, a signature will need to be added to the `IService` interface.

```
void Get();
```

The `SpeakerService` will need to be modified in order to get the application to compile.

```
public void Get()
{
  throw new NotImplementedException();
}
```

To make this test pass, simply call the `Get` method of the `SpeakerService`.

```
public IActionResult Get(int id)
{
  _speakerService.Get();

  return Ok(new SpeakerDetail());
}
```

The method signature of the `Get` method in the `ISpeakerService` will need to be modified to return a `SpeakerDetail` instead of `void`.

```
SpeakerDetail Get();
```

Now ensure that the ID passed into the `Get` method in the `SpeakerController` is what is provided to the `Get` method in the `SpeakerService`.

```
[Fact]
public void ItCallsGetServiceWithProvidedId()
{
    // Arrange
    const int id = 1;

    // Act
    _controller.Get(id);

    // Assert
    _speakerServiceMock.Verify(mock => mock.Get(id),Times.Once());
}
```

This will require modifications to the `ISpeakerService` interface as well as the `SpeakerService` class.

```
SpeakerDetail Get(int id);

...

public SpeakerDetail Get(int id)
{
    throw new NotImplementedException();
}
```

Now return the result of the `Get` method of the `SpeakerService`.

```
[Fact]
public void GivenSpeakerServiceThenResultIsReturned()
{
    // Arrange
    // Act
    var result = _controller.Get(1) as OkObjectResult;

    // Assert
    Assert.NotNull(result);
    var speaker = ((SpeakerDetail)result.Value);
    Assert.Equal(_speaker, speaker);
}
```

In order to make this test pass, simply return the result of the `Get` method.

```
public IActionResult Get(int id)
{
    var speaker = _speakerService.Get();

    return Ok(speaker);
}
```

Here is what the final results of the `SpeakerController` currently look like:

```
using Microsoft.AspNetCore.Mvc;
using SpeakerMeet.Api.Services;

namespace SpeakerMeet.Api.Controllers
{
    [Route("api/[controller]")]
    public class SpeakerController : Controller
    {
        private readonly ISpeakerService _speakerService;

        public SpeakerController(ISpeakerService speakerService)
        {
            _speakerService = speakerService;
```

```csharp
    }

    [Route("search")]
    public IActionResult Search(string searchString)
    {
      var speakers = _speakerService.Search(searchString);

      return Ok(speakers);
    }

    public IActionResult GetAll()
    {
      var speakers = _speakerService.GetAll();

      return Ok(speakers);
    }

    public IActionResult Get(int id)
    {
      var speaker = _speakerService.Get(id);

      return Ok(speaker);
    }
  }
}
```

Service

Now that the controller is calling the `Get` method of the `Moq` service, it's time to implement this method in the `SpeakerService`.

Service tests

The `Get` method was declared as a result of previous tests. Create a new `ItExists` test and delete the implementation to see it fail.

```
[Fact]
public void ItHasGetMethod()
{
  // Act
  // Arrange
  _speakerService.Get();
}
```

Make this test pass by implementing the `Get` method.

```
public void Get()
{
}
```

Now ensure the `Get` method accepts an integer.

```
[Fact]
public void ItAcceptsAnInteger()
{
  // Act
  // Arrange
  _speakerService.Get(1);
}
```

Modify the `Get` method to accept an integer.

```
public SpeakerDetail Get(int id)
{
}
```

Test that the `Get` method returns a `SpeakerDetail` object.

```
[Fact]
public void ItReturnsSpeakerDetail()
{
  // Arrange
  // Act
  var speaker = _speakerService.Get(1);

  // Assert
  Assert.NotNull(speaker);
  Assert.IsType<SpeakerDetail>(speaker);
}
```

To make this test pass, simply return a new `SpeakerDetail` object.

```
public SpeakerDetail Get(int id)
{
  return new SpeakerDetail();
}
```

Verify that the `SpeakerDetail` returned contains an ID.

```
[Fact]
public void GivenSpeakerReturnsId()
{
  // Arrange
  // Act
  var speaker = _speakerService.Get(1);

  // Assert
  Assert.Equal(1, speaker.Id);
}
```

Now make the test pass.

```
public SpeakerDetail Get(int id)
{
  return new SpeakerDetail
  {
    Id = 1,
    Name = "Joshua"
  };
}
```

Confirm that the `SpeakerDetail` contains a name.

```
[Fact]
public void GivenSpeakerReturnsName()
{
  // Arrange
  // Act
  var speaker = _speakerService.Get(1);

  // Assert
  Assert.Equal("Joshua", speaker.Name);
}
```

And make the test pass.

```
public SpeakerDetail Get(int id)
{
  return new SpeakerDetail
  {
    Id = 1,
    Name = "Joshua"
  };
}
```

Finally, ensure that `Location` is returned.

```
[Fact]
public void GivenSpeakerReturnsLocation()
{
  // Arrange
  // Act
  var speaker = _speakerService.Get(1);

  // Assert
  Assert.Equal("Tampa, FL", speaker.Location);
}
```

And make the test pass by returning the location.

```
public SpeakerDetail Get(int id)
{
  return new SpeakerDetail
  {
    Id = 1,
    Name = "Joshua",
```

```
    Location = "Tampa, FL"
  };
}
```

Clean the tests

Don't forget to clean and refactor the tests. Collapse the property tests.

```
[Fact]
public void GivenSpeakerReturnsSpeakerWithProperties()
{
  // Arrange
  // Act
  var speaker = _speakerService.Get(1);

  // Assert
  Assert.Equal(1, speaker.Id);
  Assert.Equal("Joshua", speaker.Name);
}
```

More from the repository

Now, verify that the repository is called.

```
[Fact]
public void ItCallsRepository()
{
  // Arrange
  var fakeRepository = new FakeRepository();
  var service = new SpeakerService(fakeRepository);

  // Act
  service.Get(-1);

  // Assert
  Assert.True(fakeRepository.GetCalled);
}
```

Now ensure the test passes by implementing the necessary modification.

```
public SpeakerDetail Get(int id)
{
  _repository.Get();

  return new SpeakerDetail
  {
    Id = 1,
    Name = "Joshua"
  };
}
```

Additional factory work

As before, it would be ideal if the values weren't hardcoded. Use the factory to create a speaker and have the repository return the designated speaker.

```
[Fact]
public void ItReturnsSpeakerFromRepository()
{
  // Arrange
  var fakeRepository = new FakeRepository();
  var expectedSpeaker = SpeakerFactory.Create(fakeRepository, 2, "Bill");
  var service = new SpeakerService(fakeRepository);

  // Act
  var actualSpeaker = service.Get(expectedSpeaker.Id);

  // Assert
  Assert.True(fakeRepository.GetCalled);
  Assert.Equal(expectedSpeaker.Id, actualSpeaker.Id);
  Assert.Equal(expectedSpeaker.Name, actualSpeaker.Name);
}
```

To get this to pass requires a modification to `IRepository`, `FakeRepository`, and `Service`.

`IRepository`:

```
        Speaker Get(int id);
```

`FakeRepository`:

```
public Speaker Get(int id)
{
  GetCalled = true;

  return Speakers.Find(x => x.Id == id);
}
```

`Service`:

```
public SpeakerDetail Get(int id)
{
  var speaker = _repository.Get(id);

  return new SpeakerDetail
  {
    Id = speaker.Id,
    Name = speaker.Name
  };
}
```

All previous tests to `ItReturnsSpeakerFromRepository` can now be deleted. These were all yak shaving in order to get to this point.

Now, to ensure that this will work with numerous values, convert the last test to a set of theories.

```
[Theory]
[InlineData(1, "Joshua")]
[InlineData(2, "Bill")]
```

```
[InlineData(3, "Suzie")]
public void ItReturnsSpeakerFromRepository(int id, string name)
{
  // Arrange
  var expectedSpeaker = SpeakerFactory.Create(_fakeRepository, id, name);
  var service = new SpeakerService(_fakeRepository);

  // Act
  var actualSpeaker = service.Get(expectedSpeaker.Id);

  // Assert
  Assert.True(_fakeRepository.GetCalled);
  Assert.Equal(expectedSpeaker.Id, actualSpeaker.Id);
  Assert.Equal(expectedSpeaker.Name, actualSpeaker.Name);
}
```

All the tests should pass. If for some reason a failing test is encountered, do not proceed until the failing test is resolved.

Testing exception cases

Testing exception cases is a very important step. In this case, the business has defined a case where we will return a SPEAKER NOT FOUND error if the speaker does not exist. It is also important for the developer to consider any significant edge cases the business has missed. Discuss them with the business if you can and get them added to the spec.

Now test that the speaker must exist.

```
[Fact]
public void GivenSpeakerNotFoundThenSpeakerNotFoundException()
{
  // Arrange
  var service = new SpeakerService(_fakeRepository);

  // Act
  var exception = Record.Exception(() => service.Get(-1));

  // Assert
  Assert.IsAssignableFrom<SpeakerNotFoundException>(exception);
}
```

And make it pass.

```
public SpeakerDetail Get(int id)
{
  var speaker = _repository.Get(id);

  if (speaker == null)
  {
    throw new SpeakerNotFoundException();
  }

  return new SpeakerDetail
  {
    Id = speaker.Id,
    Name = speaker.Name
  };
}
```

Now, verify that the speaker is not deleted. If it is deleted, throw the same `SpeakerNotFoundException`.

```
[Fact]
public void GivenSpeakerIsDeletedThenSpeakerNotException()
{
  // Arrange
  var expectedSpeaker = SpeakerFactory.Create(_fakeRepository).IsDeleted();
  var service = new SpeakerService(_fakeRepository);

  // Act
  var exception = Record.Exception(() => service.Get(expectedSpeaker.Id));

  // Assert
  Assert.IsAssignableFrom<SpeakerNotFoundException>(exception);
}
```

The simplest, most effective way to make this test pass is to throw an exception if the speaker found has been deleted. Make the necessary change to the `Get` method.

```
public SpeakerDetail Get(int id)
{
  var speaker = _repository.Get(id);

  if (speaker == null || speaker.IsDeleted)
  {
    throw new SpeakerNotFoundException();
  }

  return new SpeakerDetail
  {
    Id = speaker.Id,
    Name = speaker.Name
   };
}
```

Summary

Now, you should feel fairly comfortable with the requirements surrounding the Speaker Meet application and have had a decent introduction to the API, Service, and Repository layers for the Speaker section of the back-end application. Mocks and Fakes continue to play a role in the Test-Driving of the program.

In Chapter 8, *Abstract Away Problems,* more will be discussed with respect to abstractions. The models for `SpeakerSummary` and `SpeakerDetail` will be grown to include more properties. Additional details will be provided on how best to increase the functionality, and with it the complexity, of the application.

Abstract Away Problems

These days, it is quite easy to find resources on the internet to integrate into your application. Many provide functionality that would be perfectly suited to any number of applications. After all, why spend time reinventing the wheel when someone else has already done the bulk of the work for you?

In this chapter, we will gain an understanding of:

- Abstracting a Gravatar service
- Extending the repository pattern
- Using a generic repository and Entity Framework

Abstracting away problems

There is an abundance of utilities and libraries these days to help make a full-featured application. It can be quite easy to integrate these third-party systems within your application. At times, however, you may need to replace one third-party library with another. Alternatively, you may find yourself relying on the implementation that a third-party system provides, only to find that the implementation has changed with a later update. How can you avoid these potential problems?

Creating a dependency on code that is outside your control can create problems for you in the future. If a change is introduced in a library that you depend on, it could potentially break your system. Or, if your requirements change and the system no longer fits your specific needs you may have to rewrite large portions of your application.

Don't depend directly on any third-party system. Abstract away the details so that your application depends only on an interface that you define. If you define the interface and expose only the functionality that you need, it can become trivial to make changes when they are required. Changes could include minor updates or replacing whole libraries. You want these changes to have minimal impact on the rest of your application.

Don't rely on third-party implementations; focus on test driving your code.

While developing an application with Test-Driven Development in mind, it can often be tempting to test third-party software. While it is important to ensure that any third-party library or utility works well when integrated into your system, it is best to focus on the behavior of your system. Ensure that your system behaves well with the functionality that you wish to expose.

This means that you should handle the *happy path* as well as any possible *exceptions* that may be thrown. Gracefully recovering from an error that crops up will allow your application to continue to function in the event that a third-party service is not functioning as you expect.

Gravatar

The Speaker Meet application uses Gravatar to display speaker, community, and conference avatar images. Gravatar is an online service that associates an email address with an image. Users can create an account and add an image that they wish to be shown by any service that requests their image. The image is retrieved from the Gravatar service by creating an MD5 hash of the user's email address and requesting an image from Gravatar by supplying the hashed value. By relying on the hashed value, the user's email address is not exposed.

The Gravatar service allows the consumer to supply optional parameters to the HTTP call in order to request a specific size, rating, or default image if none is found. Some of these options include:

- **s**: The requested size of the image; by default, this is 80 x 80 pixel
- **d**: The default image if none is found; options include 404, **mm** (**mystery-man**) , identicon, and so on
- **f**: Force default; always return the default icon, even if an image is found
- **r**: Rating; users can label their image as G, PG, R, and X

By supplying these values, you have some control over the size and types of image you wish to display within your application. The Speaker Meet application relies on the default offerings from Gravatar.

Starting with an interface

Looking at the Gravatar site, it appears that there a number of options available. In order to shield the rest of the application, the functionality of Gravatar will be exposed through a class contained within the Speaker Meet application. This functionality will first be defined by an interface.

The desired interface might look something like this:

```
public interface IGravatarService
{
  string GetGravatar(string emailAddress);
  string GetGravatar(string emailAddress, int size);
  string GetGravatar(string emailAddress, int size, string rating);
  string GetGravatar(string emailAddress, int size, string rating,
  string imageType);
}
```

To get started, you must first write some tests. Remember, you should not write a line of production code without a failing unit test.

Implementing a test version of the interface

In order to create an interface named `IGravatarService`, there first must be a need within the application. Create a test within a `SpeakerServiceTests Get` class entitled `ItTakesGravatarService`:

```
[Fact]
public void ItTakesGravatarService()
{
  // Arrange
  var fakeGravatarService = new FakeGravatarService();
  var service = new SpeakerService(_fakeRepository, fakeGravatarService);
}
```

This will cause a compilation error. Create an `IGravatarService` and modify the constructor of the `SpeakerService` so that this is a parameter.

Interface:

```
public interface IGravatarService
{
}
```

`SpeakerService` method:

```
public SpeakerService(IRepository repository, IGravatarService gravatarService)
{
  _repository = repository;
}
```

In order to get the tests to compile, create a `FakeGravatarService` that can be supplied to the `SpeakerService` under test. Remember, you're not testing the `FakeGravatarService`, merely that the `SpeakerService` accepts an `IGravatarService` instance.

Now, ensure that the `FakeGravatarServiceGetGravatar` method is called when an individual *Speaker* is requested.

```
[Fact]
public void ItCallsGravatarService()
{
  // Arrange
  var expectedSpeaker = SpeakerFactory.Create(_fakeRepository);
  var service = new SpeakerService(_fakeRepository, _fakeGravatarService);

  // Act
  service.Get(expectedSpeaker.Id);

  // Assert
  Assert.True(_fakeGravatarService.GetGravatarCalled);
}
```

Modify the interface to add a `GetGravatar` method:

```
public interface IGravatarService
{
  void GetGravatar();
}
```

And implement this method in the `FakeGravatarService`. This is similar to the `GetCalled` check of the `FakeRepository` from Chapter 7, *Test Driving C# Applications*:

```
public class FakeGravatarService : IGravatarService
{
  public bool GetGravatarCalled { get; set; }

  public void GetGravatar()
  {
    GetGravatarCalled = true;
  }
}
```

Next, ensure that the `GetGravatar` function is executed when the `SpeakerService Get(id)` is called:

```
private readonly IRepository _repository;
private readonly IGravatarService _gravatarService;

public SpeakerService(IRepository repository, IGravatarService gravatarService)
{
  _repository = repository;
  _gravatarService = gravatarService;
}

public Models.SpeakerDetail Get(int id)
{
  var speaker = _repository.Get(id);

  if (speaker == null || speaker.IsDeleted)
  {
    throw new SpeakerNotFoundException();
  }

  var gravatar = _gravatarService.GetGravatar();

  return new Models.SpeakerDetail
  {
    Id = speaker.Id,
    Name = speaker.Name,
    Location = speaker.Location
  };
}
```

The test should now pass. However, the `FakeGravatarService` isn't providing any real value at the moment. The `GetGravatar` method should be executed with a provided email address:

```
[Fact]
public void ItCallsGravatarServiceWithEmail()
{
  // Arrange
  var expectedSpeaker = SpeakerFactory.Create(_fakeRepository, emailAddress: "example@test.com
  var service = new SpeakerService(_fakeRepository, _fakeGravatarService);

  // Act
  service.Get(expectedSpeaker.Id);

  // Assert
  Assert.True(_fakeGravatarService.WithEmailCalled);
  Assert.Equal(expectedSpeaker.EmailAddress, _fakeGravatarService.CalledWith);
}
```

You will need to modify the `SpeakerFactory` to accept an email address and the `Speaker` model

class to house an email address property.

Modify the `GetGravatar` method in the `FakeGravatarService` and the `IGravatarService` interface to accept a string `emailAddress`. Make sure you set the `CalledWith` property when the `GetGravatar` is executed:

```
public string CalledWith { get; set; }

public void GetGravatar(string emailAddress)
{
  GetGravatarCalled = true;
  CalledWith = emailAddress;
}
```

And ensure the `GetGravatar` method is called with the speaker's email address:

```
public Models.SpeakerDetail Get(int id)
{
  var speaker = _repository.Get(id);

  if (speaker == null || speaker.IsDeleted)
  {
    throw new SpeakerNotFoundException();
  }

  var gravatar = _gravatarService.GetGravatar(speaker.EmailAddress);

  return new Models.SpeakerDetail
  {
    Id = speaker.Id,
    Name = speaker.Name,
    Location = speaker.Location,
  };
}
```

Finally, set the return value of the `GetGravatar` method to a new property on the `SpeakerDetail` object `Gravatar`:

```
[Fact]
public void GivenGravatarServiceThenItSetsGravatar()
{
  // Arrange
  var expectedSpeaker = SpeakerFactory.Create(_fakeRepository);
  var service = new SpeakerService(_fakeRepository,
   _fakeGravatarService);

  // Act
  var actualSpeaker = service.Get(expectedSpeaker.Id);
  var expectedGravatar =
   _fakeGravatarService.GetGravatar(expectedSpeaker.EmailAddress);

  // Assert
  Assert.True(_fakeGravatarService.WithEmailCalled);
  Assert.Equal(expectedSpeaker.Id, actualSpeaker.Id);
  Assert.Equal(expectedSpeaker.Name, actualSpeaker.Name);
  Assert.Equal(expectedGravatar, actualSpeaker.Gravatar);
}
```

You will need to modify the `FakeGravatarService` and its interface, and the `SpeakerService Get` method to return a string, and the `SpeakerDetail` class to add a `Gravatar` property:

```
public string GetGravatar(string emailAddress)
{
  WithEmailCalled = true;
  CalledWith = emailAddress;
```

```
    return System.Reflection.MethodBase.GetCurrentMethod().Name;
}
```

The return value of the `GetGravatar` method doesn't matter, so long as it is a known value. Remember, you're not testing that the `FakeGravatarService` returns a valid Gravatar image URL, just that the method returns something and that the return value is set to the `Gravatar` property on the `SpeakerDetail` object.

Implementing the production version of the interface

So far, an `IGravatarService` interface has been created with one method, `GetGravatar`. There are a number of options available to interact with Gravatar. You could choose to write your own methods to communicate directly with its public API. The Speaker Meet application uses one of the available `NuGet` packages, `GravatarHelper.NetStandard`.

Install the latest version of `GravatarHelper.NetStandard` through `NuGet` in order to follow along.

While reviewing the Gravatar website, it appears that they offer a variety of optional parameters. To grow the `IGravatarService` interface and its implementation, create a new test class, `GetGravatar`:

```
public class GetGravatar
{
}
```

Now test that the `GravatarService` exists:

```
[Fact]
public void ItExists()
{
  var gravatarService = new GravatarService();
}
```

Make this test pass by creating a `GravatarService` class in the same location as the `SpeakerService`:

```
namespace SpeakerMeet.Api.Services
{
  public class GravatarService
  {}
}
```

Now, ensure that the `GravatarService` implements the `IGravatarInterface`:

```
[Fact]
public void ItImplementsIGravatarInterface()
{
  // Arrange
  // Act
  var gravatarService = new GravatarService();

  // Assert
  Assert.IsAssignableFrom<IGravatarService>(gravatarService);
}
```

From the previous set of tests, a `GetGravatar` method has already been defined within the interface. Make the test pass by implementing the interface:

```
public class GravatarService : IGravatarService
{
  public string GetGravatar(string emailAddress)
```

```
  {
    throw new System.NotImplementedException();
  }
}
```

Verify that the `GetGravatar` method exists with a new test:

```
[Fact]
public void ItHasGetGravatarMethod()
{
  // Arrange
  IGravatarService gravatarService = new GravatarService();

  // Act
  gravatarService.GetGravatar("example@test.com");
}
```

Allow this test to pass by returning an empty string:

```
public string GetGravatar(string emailAddress)
{
  return string.Empty;
}
```

The following tests are classified as *integration tests* as they're testing how the Speaker Meet application interacts with a third-party system. Decorate the class as such:

```
[Trait("Category", "Integration")]
public class GetGravatar
```

Many test runners will allow you to conditionally run or exclude these tests based on trait categories. Once the integration tests are defined and known to run successfully, you may choose to ignore or disable them on change or only run them before check-in.

Now, test that the Gravatar service returns a known value when an email address is supplied. If you have a Gravatar account, feel free to supply your own email address and test for your Gravatar URL:

```
[Fact]
public void GivenEmailAddressThenGravatarReturned()
{
  // Arrange
  IGravatarService gravatarService = new GravatarService();

  // Act
  var actual = gravatarService.GetGravatar("example@test.com");

  // Assert
  Assert.Equal("http://www.gravatar.com/avatar/29e3f53ee49fae541ee0f48fb712c231", actual);
}
```

Now, make this test pass by calling the static method supplied by the `GravatarHelper`:

```
public string GetGravatar(string emailAddress)
{
  return Gravatar.GetGravatarImageUrl(emailAddress);
}
```

The test should now pass. You can see how the implementation has been hidden from the rest of the application. The interface was designed out of necessity through a series of tests in the `SpeakerService`.

So, why not just call the `GravatarHelper` methods directly from the `SpeakerService` and elsewhere? Remember, you shouldn't rely on third-party implementations. If the `GravatarHelper` is changed or swapped out for something else entirely, then any class that is calling it directly may need to change. By using an interface and a façade, the only class that would potentially need to change is the `GravatarService`.

Future planning

Future planning can be bad. If you're writing code now in anticipation of future problems, you could be wasting effort. Don't write code you don't need. This could add complexity that can slow development.

Remember the term **YAGNI** (**you ain't gonna need it**) as this applies to any code written without an immediate need. The additional `GravatarService` methods previously, could be used as an illustration of exactly that. Of the examples provided so far, none require the additional methods that were just created. If for some reason the implementation of the `GravatarHelper` changes, the code that has already been written may need to change. If it is not currently being used, this is a waste of effort.

So, where does future planning start and good abstraction end? Abstract away third-party systems. Only expose methods and functionalities that are an immediate need. Minimize the pain of change by shielding the rest of the application from the details of any third-party system. That includes things such as the .NET Framework and ORMs such as Entity Framework.

Abstracting the data layer

The data layer abstraction has already begun with the implementation of a repository pattern. In this section, we will work to create a valid abstraction for connecting to an Entity Framework. After we can communicate with the Entity Framework, we will then focus on making the repository more generic and able to work with multiple data models.

Extending the repository pattern

The first step in creating a valid data layer abstraction is to make sure CRUD has been handled. **CRUD** (**Create**, **Read**, **Update**, and **Delete**) are the basic operations that can be performed on any dataset. The `IRepository` does not yet provide access to all of these capabilities so we will begin by extending it.

First create a folder to contain the tests for a `SpeakerRepository`. The folder should be named in line with the folders containing the `SpeakerService` tests and `SpeakerController` tests. As usual, we start with a failing test. In this case, the test is failing to compile:

```
[Trait("Category", "SpeakerRepository")]
public class Class
{
  [Fact]
  public void ItExists()
  {
    var repo = new SpeakerRepository();
  }
}
```

Create the `SpeakerRepository` and the test should pass:

```
public class SpeakerRepository
{
}
```

The `SpeakerRepository` needs to inherit from `IRepository` so either transform the existence test into a test for type, or create a new test:

```
[Fact]
public void ItIsARepository()
{
  // Arrange / Act
  var repo = new SpeakerRepository();

  // Assert
  Assert.IsAssignableFrom<IRepository<Speaker>>(repo);
}
```

Now, make the test pass by inheriting from `IRepository`. We don't have tests for functionality at this point, so leave the repository methods as not implemented:

```
public class SpeakerRepository : IRepository<Speaker>
{
  public Speaker Get(int id)
  {
    throw new System.NotImplementedException();
  }

  public IQueryable<Speaker> GetAll()
  {
    throw new System.NotImplementedException();
```

```
    }
}
```

The Get method

Now that `SpeakerRepository` properly inherits from `IRepository`, the two methods currently defined by `IRepository` need to be implemented. As we did for `SpeakerService`, we need to create a new test class specifically for the `Get` method. Again, while it might seem to be overkill at this point, creating a file per method will help with organization as the test suite grows:

```
[Trait("Category", "SpeakerRepository")]
public class Get
{
}
```

Now that the class exists, the first test method that can be written is a simple exists method:

```
[Fact]
public void ItHasGetMethod()
{
  // Arrange
  var repo = new SpeakerRepository();

  // Act
  var result = repo.Get(0);
}
```

Initially this test will fail because the stub implementation provided by Visual Studio just throws a `NotImplementedException`. To fix this, we have to return something; so what should be returned? There is no test to explain what result is expected so we must go with something that will compile but is almost certainly incorrect. In this situation the correct, incorrect value to return is probably `null`:

```
public Speaker Get(int id)
{
  return null;
}
```

The GetAll method

The test now passes. Let us pause here and get the same amount of testing around the `GetAll` method enforced by the interface. As previously, create a new class for `GetAll`:

```
[Trait("Category", "SpeakerRepository")]
public class GetAll
{
}
```

Create an exists method which will just ensure the method doesn't throw when called:

```
[Fact]
public void ItHasGetAllMethod()
{
  // Arrange
  var repo = new SpeakerRepository();

  // Act
  var result = repo.GetAll();
}
```

The Create method

It will probably be easier to test the repository pattern if all the repository methods are assumed to exist. Unfortunately, a repository presents a chicken and egg scenario. How can we test `Get` or `GetAll` without some way of creating entries in the repository? At the same time, how can we test `Create` or `Delete` without some way of retrieving entries from the repository?

Next create a new class for `Create`:

```
[Trait("Category", "SpeakerRepository")]
public class Create
{
}
```

As before, write a method to check for existence. In this test, we need to be sure to test against the repository, not the `Create` class implementation. This will force us to add to the interface:

```
[Fact]
public void ItHasCreateMethod()
{
  // Arrange
  IRepository<Speaker> repo = new SpeakerRepository();

  // Act
  var result = repo.Create(new Speaker());
}
```

You may have noticed, in this test, that we receive a result from the `Create` method. This may not be obvious since we received values from the `Get` and `GetAll` methods, but we are choosing to break **CQRS** (**Command Query Responsibility Separation**) in favor of a more RESTful approach. In **REST** (**Representational State Transfer**), because it must remain stateless and cannot provide information about an action that has already finished, a service will generally return either the object created or a way to retrieve that object in the future.

In this case, you might think that we are now providing a leaky abstraction of the web. It could be interpreted that way. I prefer to look at this choice as opening options instead of limiting them. It will be easier to hide a CQRS implementation behind a RESTful interface than it would be to work things the other way around.

Now, to pass the currently failing test, a method definition will need to be added to the `IRepository` interface, a method implementation will need to be added to the `SpeakerRepository`, and the `SpeakerRepository` implementation will need to be amended to not throw:

```
public interface IRepository<T>
{
  T Get(int id);
  IQueryable<T> GetAll();
  T Create(T item);
}
```

`Speaker` `Save` method:

```
public Speaker Save(Speaker speaker)
{
 return null;
}
```

A stub implementation will also have to be added to the `FakeRepository` that was defined in , *Test Driving C# Applications*.

The Delete method

Next, we will add the `Delete` method. Just as before, create a new test class:

```
[Trait("Category","SpeakerRepository")]
public class Delete
{
}
```

Just like for the `Create` method, we need to treat the `SpeakerRepository` as an `IRepository`. Create a `Delete` exists method:

```
[Fact]
public void ItHasDeleteMethod()
{
  // Arrange
  IRepository<Speaker> repo = new SpeakerRepository();
  var speaker = new Speaker();

  // Act
  repo.Delete(speaker);
}
```

Implementing this method will be slightly easier as it is a void method and we are not expecting a result. There are some decisions that will have to be made later regarding how the method should behave when it is passed to a speaker that does not exist. For now, we can just assume that nothing happens and the method is successful.

As before, modify the `IRepository` to contain a `Delete` method:

```
public interface IRepository<T>
{
  T Get(int id);
  IQueryable<T> GetAll();
  T Create(T item);
  void Delete(T item);
}
```

And now create the stub method in the `SpeakerRepository` and the `FakeRepository`:

```
public void Delete(Speaker speaker)
{
}
```

The Update method

There is one last method required for a valid repository pattern and any useful system. We need the ability to update the models we are working with. As with the last two methods, this one does not yet exist in the repository, so let's add it.

As before, begin by creating a test class for it:

```
[Trait("Category", "SpeakerRepository")]
public class Update
{
}
```

Just like the others, create an ItExists test referencing the IRepository:

```
[Fact]
public void ItHasUpdateMethod()
{
  // Arrange
  IRepository<Speaker> repo = new SpeakerRepository();
  var speaker = new Speaker();

  // Act
  var result = repo.Update(speaker);
}
```

As before, we are just stubbing functionality here so we don't yet have an actual speaker to update. This test, and the others, will almost certainly have to change as we begin testing for actual functionality. For now, they will suffice to ensure the interface and class have the appropriate methods.

As before, add the method to the interface and then to the SpeakerRepository and the FakeRepository:

```
public interface IRepository
{
  TGet(int id);
  IQueryable<T> GetAll();
  T Create(T item);
  T Update(T item);
  void Delete(T item);
}
```

Speaker Update method:

```
public Speaker Update(Speaker speaker)
{
  return null;
}
```

Ensuring functionality

Now that all the methods have been defined, we can begin writing tests for functionality. We will begin with `Create` and work our way down to `Delete`.

Creating a speaker

The chicken and egg scenario mentioned earlier has us in a predicament. We can't read a speaker from the repository if no speakers have been created. We also can't verify that a speaker has in fact been created unless we can retrieve a speaker from the repository.

One way to solve this problem is by using a special kind of test double that exposes the internal functionality of a class for the purposes of asserting on that information. For `Create`, we will use this approach. In the `Create.cs` file, let's add a test that assumes the testable class already exists:

```
[Fact]
public void ItAddsASpeakerToTheRepository()
{
  // Arrange
  var repo = new TestableSpeakerRepository();

  // Act
  var result = repo.Create(new Speaker());

  // Assert
  Assert.Equal(1, repo.SpeakersCollection.Count);
}
```

To get past the compilation error, the testable class must be created. Create this class in the same file, for now, to make working with it more efficient:

```
public class TestableSpeakerRepository : SpeakerRepository
{
}
```

Now the class is created and the initial compilation error is resolved, but a new error has risen:

```
// Assert
Assert.Equal(1, repo.SpeakersCollection.Count);
```

This error is slightly more difficult to resolve. In reality, we want a collection of speakers to exist in the real repository. However, we have had no reason to expose that collection. No collection has actually been created as a result. Now, in this test we are asking whether a collection exists. The test requires that a collection exists for the `TestableSpeakerRepository` but we know we need one for the real `SpeakerRepository`. Let's play devil's advocate and actually do the thing we know is not quite right:

```
public class TestableSpeakerRepository : SpeakerRepository
{
  public IQueryable<Speaker> SpeakersCollection { get; set; }
}
```

This change doesn't quite make the test pass; when writing the test we hastily accessed the `Count` property on Speakers. The `Count` property is only on a list, but to limit the exposure of an interface until we actually can require it with tests, we should really be using an `IQueryable`. We can quickly update the test to reflect this choice:

```
// Assert
```

```
Assert.Equal(1, repo.SpeakersCollection.Count);
```

Now, execute the tests and it finally fails with an actual message. The solution to this failure is to add an entry to the speakers collection when `Create` is called. The problem is that Speakers is in a different class from `Create`. So, we must have a collection in `SpeakerRepository` as well:

```
protected readonly IList<Speaker> Speakers = new List<Speaker>();

public Speaker Create(Speaker speaker)
{
  Speakers.Add(speaker);

  return speaker;
}
```

What's important to note is the scope and type of `_speakers`. It is an `IList`, because we need to add an item to it; an `IQueryable` will not do. It is also protected; `_speakers` must be hidden from the outside world but also must be accessible from the testable class. The scope operator that gives us this functionality is protected.

We must also make changes in the testable class in order to make this test pass:

```
internal class TestableSpeakerRepository : SpeakerRepository
{
  public IList<Speaker> SpeakersCollection => Speakers;
}
```

Continuing with `Create`, we now need to verify that, when a new speaker is created, it receives a unique ID:

```
[Fact]
public void ItAssignsUniqueIdsToEachSpeaker()
{
  // Arrange
  var repo = new TestableSpeakerRepository();

  // Act
  var speaker1 = repo.Create(new Speaker());
  var speaker2 = repo.Create(new Speaker());

  // Assert
  Assert.NotEqual(speaker1.Id, speaker2.Id);
}
```

To make this test pass, we must come up with some kind of ID generation system. There are many options, but one of the simplest is to create a private field and increment the value each time `Create` is called:

```
private int _currentId = 0;

public Speaker Create(Speaker speaker)
{
  speaker.Id = ++_currentId;

  Speakers.Add(speaker);

  return speaker;
}
```

With that test passing, we can now turn our attention to a leak in the abstraction. We are simply placing the passed-in object into the dataset. This could cause an issue in an application that needs to be fixed.

The repository should isolate its object from the rest of the application by passing and storing clones, instead of directly accessing and providing objects:

```
[Fact]
public void ItReturnsANewSpeaker()
{
  // Arrange
  var repo = new TestableSpeakerRepository();
  var speaker = new Speaker { Id = 0 };

  // Act
  var result = repo.Create(speaker);

  // Assert
  Assert.Equal(0, speaker.Id);
}
```

To make this test pass, we will need some cloning mechanism. To make this test pass as soon as possible, we can simply use a new object and object initializer:

```
public Speaker Create(Speaker speaker)
{
  var newSpeaker = new Speaker
  {
    Id = ++_currentId,
    Name = speaker.Name,
    Location = speaker.Location,
    IsDeleted = speaker.IsDeleted
  };

  Speakers.Add(newSpeaker);

  return newSpeaker;
}
```

Now, we must handle the other direction for reference passing. The value stored in the speakers collection should not directly be handed back to us from the `Create` method:

```
[Fact]
public void ItProtectsAgainstObjectChangesAfterCreation()
{
  // Arrange
  var repo = new TestableSpeakerRepository();
  var speaker = repo.Create(new Speaker());

  // Act
  speaker.Name = "test name";

  // Audit
  var result = repo.SpeakersCollection.First();

  // Assert
  Assert.NotEqual("test name", result.Name);
}
```

Notice the extra auditing step. Sometimes, you will need to take an action and then assert on a deeply nested value or a distant value. In those cases, you can keep a clean single-step action by adding an audit step.

To make this test pass, we must take a similar action to what we are already doing at the top of the `Create` method:

```
public Speaker Create(Speaker speaker)
{
  var newSpeaker = new Speaker
```

```
  {
    Id = ++_currentId,
    Name = speaker.Name,
    Location = speaker.Location,
    IsDeleted = speaker.IsDeleted
  };

  Speakers.Add(newSpeaker);

  var returnableSpeaker = new Speaker
  {
    Id = newSpeaker.Id,
    Name = newSpeaker.Name,
    Location = newSpeaker.Location,
    IsDeleted = newSpeaker.IsDeleted
  };

  return returnableSpeaker;
}
```

That completes the functionality required for the `Create` method. Now we should really do some long overdue refactoring. Firstly, let's focus on the tests and reduce the duplicated calls to create a new repository.

Create a constructor and a private `repo` field:

```
private readonly TestableSpeakerRepository _repo;

public Create()
{
  _repo = new TestableSpeakerRepository();
}
```

Then replace all repository references in the tests with `_repo`. After reducing the number of repositories being created in the tests, the tests look pretty good. Now we can focus on the `SpeakerRepository` class.

One of the immediate standouts in the `SpeakerRepository` is the code we are using to clone a speaker. The same code has essentially been typed twice in the same method. Let's abstract this to a private function inside the repository for now. We may end up making a more sophisticated solution later on, but for now that should be good enough.

At the bottom of the class, after all the public methods, we can create a `CloneSpeaker` method:

```
private Speaker CloneSpeaker(Speaker speaker)
{
  return new Speaker
  {
    Id = speaker.Id,
    Name = speaker.Name,
    Location = speaker.Location,
    IsDeleted = speaker.IsDeleted
  };
}
```

Then we use the `CloneSpeaker` method in `Create`:

```
public Speaker Create(Speaker speaker)
{
  var newSpeaker = CloneSpeaker(speaker);

  newSpeaker.Id = ++_currentId;

  Speakers.Add(newSpeaker);
```

```
    return CloneSpeaker(newSpeaker);
}
```

Getting a single speaker

With the existence of `Create`, we can now very easily assert on the retrieval of an existing or non-existing speaker. According to Uncle Bob's Transformation Priority Premise, it is easier and simpler to test a singular item rather than a plural item, so while it doesn't completely fulfil the intent of the premise, we will test the retrieval of a singular speaker next.

We already have an exists test, so what will the next test be? The simplest test would be the retrieval of a single speaker, but if we are trying to avoid the gold standard, the most appropriate test would be examining what happens when a speaker does not exist.

For non-existing speakers, we have a few immediately apparent options. We could throw an error stating that the requested speaker is not in the system. Another option would be returning a `null` object. And the last would be simply returning `null`.

Throwing an error is probably the most straightforward, so let's examine this option first:

```
[Fact]
public void ItThrowsWhenSpeakerIsNotFound()
{
  // Arrange
  var repo = new SpeakerRepository();

  // Act
  var result = Record.Exception(() => repo.Get(-1));

  // Assert
  Assert.IsType<SpeakerNotFoundException>(result.GetBaseException());
}
```

To make this test pass, first we must make it compile. The `SpeakerNotFoundException` in this case is not the same as the one we are using in the `SpeakerService`. So, it will need to be created:

```
public class SpeakerNotFoundException : Exception
{
  public SpeakerNotFoundException()
  {
  }
}
```

Now that the test is correctly compiling and failing, we can add the appropriate code to the repository to make the test pass:

```
public Speaker Get(int id)
{
  if (id == -1)
  {
    throw new SpeakerNotFoundException();
  }

  return null;
}
```

Another guideline for tests, to help you know you are on the right path and not digging yourself

into a hole, is again from Uncle Bob, *A*s the tests get more specific the code gets more generic. If we look at the code we just wrote, it seems more specific then generic. Let's refactor it to maintain a trend towards generics:

```
public Speaker Get(int id)
{
  if (id > -1)
  {
    return null;
  }

  throw new SpeakerNotFoundException();
}
```

The change here is subtle, but important. Thinking about the operation of this method in production, the default case really is to throw. There is only a small subset in the set containing all the integers that we actually have a speaker for, so the generic case is to throw. The specific case is actually to find a speaker.

One issue with the method of throwing when a speaker is not found is that it brings a possibly unexpected and abrupt end to the application flow. The entire logic path the application took to get to this method is now destroyed and the exception must be handled. Even when we handle the exception, C# uses extra CPU cycles on the first-chance exception error handling process. Sometimes throwing is definitely the right decision; however, exceptions should be reserved for truly exceptional events. As discussed earlier, it is far more likely that a `Get` could be called with an invalid ID than with a valid one. So, in this case it is not necessarily a properly exceptional event for an invalid speaker to be requested.

Let's explore the alternative of a NullObject. First, we need to revert our code to where it was when we started working on the `Get` method. We are using source control so we can very simply revert our code. If you are not using source control, I would suggest you start. Here is the state the `Get` method should be in before we begin again:

```
public Speaker Get(int id)
{
  return null;
}
```

We can just delete the test that asserts an exception was thrown.

The NullObject pattern is a simple pattern with a slightly more complicated implementation. Basically, you create an object that inherits from the class needed, but it does absolutely nothing in just the right way.

Thinking about this from our eventual website usage, a speaker null object would possibly have a name such as "Mr. Unknown" and would speak at conferences like "Mid-Nowhere Tech Fest." We could create a funny profile picture and fill out harmless user information that would let the user know they had requested a non-existent speaker. All of that information could be determined and filled out later. The important part of a null object is that it represents a fully capable object, but does absolutely nothing in the right way to not cause harm within your application:

```
[Fact]
public void ItReturnsANullSpeakerWhenNotFound()
{
```

```
  // Arrange
  var repo = new SpeakerRepository();

  // Act
  var result = repo.Get(-1);

  // Assert
  Assert.IsType<NullSpeaker>(result);
}
```

To fix the compilation error, create the `NullSpeaker` class and have it inherit from `Speaker`:

```
public class NullSpeaker : Speaker
{
}
```

Making the test pass is fairly simple. In this case we don't have to worry about existing tests that could break from returning the null object:

```
public Speaker Get(int id)
{
  return new NullSpeaker();
}
```

Superficially, the null object pattern seems like a pretty good solution. In reality, this is not always the case. It is very difficult to properly do nothing within a system. In the case of getting a speaker, the null object pattern would probably work just fine. We have one more option to explore though.

The last option is to simply return `null`. Well, simply may not be the best word choice. `Null` can cause a lot of trouble within a system. It will wreak havoc if you do not handle the null that was received and a distant part of the system tries to use the null as if it weren't null. Earlier, when we were designing the service, we decided to expect null as a possible result from calling the repository; thus, in this case the range of effects should be short and null should not cause significant negative side-effects within the system. Let's revert the code one more time and explore simply returning `null`:

```
[Fact]
public void ItReturnsNullWhenNotFound()
{
  // Arrange
  var repo = new SpeakerRepository();

  // Act
  var result = repo.Get(-1);

  // Assert
  Assert.Null(result);
}
```

There is nothing to do in the `SpeakerRepository` because we are already returning `null`. Normally, this test would not be written here. We would wait to write this test until we had a test that was returning a valid speaker. The reason we want to delay this test is because we can't fail it. You will not see this test go red and that is generally a problem. Either the test is not needed because it is extraneous and covered elsewhere or the test is flawed because it can't fail.

The test we should write before this one is a test that should retrieve a valid value. For the moment, ignore this test and we will come back and watch it fail after the next test:

```
[Fact(Skip = "Can't fail")]
public void ItReturnsNullWhenNotFound()
{
  // Arrange
  var repo = new SpeakerRepository();

  // Act
  var result = repo.Get(-1);

  // Assert
  Assert.Null(result);
}
```

So, if we are about to test the retrieval of an existing speaker, which we called the gold standard before, why did we avoid it in the first place? The reason we are going to test the gold standard first now is solely because of implementation choices. If we were going to use either as an exception or a null object, we would have done those first. The problem right now is that our code is already returning `null` so testing for null would not have done us any good.

To test for an existing speaker, a speaker must first exist. In the arrange section of our test, we will need to make sure we create a speaker:

```
[Fact]
public void ItReturnsASpeakerWhenFound()
{
  // Arrange
  var repo = new SpeakerRepository();
  var speaker = repo.Create(new Speaker {Name = "Test Speaker"});

  // Act
  var result = repo.Get(speaker.Id);

  // Assert
  Assert.NotNull(result);
}
```

The first step is to simply assert that the result is not null. This of course fails because all we are doing right now is returning `null`. Let's fix that:

```
public Speaker Get(int id)
{
  return new Speaker();
}
```

Even though we are now forced to go down the gold-standard path, we can still avoid it as much as possible. One way to do this is by playing devil's advocate, which we can do by returning a speaker, just not the right speaker.

We must now modify the test to make sure the values we care about are correct:

```
// Assert
Assert.NotNull(result);
Assert.Equal("Test Speaker", result.Name);
```

Now to modify the code. We could continue to play devil's advocate, but at this point that would only cause more unnecessary work in the tests:

```
public Speaker Get(int id)
{
  return Speakers.SingleOrDefault(s => s.Id == id);
}
```

At this point, we are left with a conundrum. We can't call `Single` or `First` because they will throw an exception if the requested speaker is not found. We do have a system rule to return `null` when the speaker is not found though. The system is behaving correctly by default. We could simply put our test back into play and accept that the null test just won't be properly verifiable. Another option is to purposely write code to return a new speaker in the event that the speakers collection is missing the requested speaker. Taking the second option would allow us to see the null test fail, but we would just be removing it to make the test pass.

In this case, the second option is probably the most appropriate. Many of you will probably think that there is no reason to add the code that forces a non-null; many others will probably think that the test is not needed as it doesn't seem to provide value. Both groups are correct and incorrect at the same time. They are correct: putting in code just to remove it two seconds later is stupid. Adding a test that can't really fail is stupid too.

However, we need the test because at some point in the future a requirement could come in that would cause a null to be accidentally impossible and we need a record of the business requirement stating that the value should be null. We also need to see every test fail. In this particular case, we could probably get away with skipping that part, but if we get into a habit of skipping test failures it will quickly bite you.

So, let's make the appropriate change to force the null test to fail and then fix our "error:"

```
public Speaker Get(int id)
{
  return Speakers.SingleOrDefault(s => s.Id == id) ?? new Speaker();
}
```

We can now remove the skip attribute on the null test and then come back here and remove the null coalescing operator.

We have one more test needed for `Get`. We must ensure the pointer for the returned speaker is not the same as the speaker in the repository:

```
[Fact]
public void ItProtectsAgainstObjectChanges()
{
  // Arrange
  var repo = new SpeakerRepository();
  var speaker = repo.Create(new Speaker { Name = "Test Speaker" });
  var retrievedSpeaker = repo.Get(speaker.Id);
  retrievedSpeaker.Name = "New Speaker Name";

  // Act
  var result = repo.Get(speaker.Id);

  // Assert
  Assert.NotEqual(retrievedSpeaker.Name, result.Name);
}
```

This test is significantly harder to follow, but necessary. We first have to create a speaker. Then we retrieve the speaker we created, update the retrieved speaker's name, and retrieve the speaker again. Finally, we verify that the modified speaker is not the same as the retrieved speaker. They should not be the same because we have not saved the data, so no updates should have occurred.

The fix for this issue has already been created and just needs to be implemented:

```
public Speaker Get(int id)
{
  var speaker = Speakers.SingleOrDefault(s => s.Id == id);

  if (speaker != null)
  {
    speaker = CloneSpeaker(speaker);
  }

  return speaker;
}
```

Lastly, we should refactor. Follow the same steps as you did for create and extract the repository creation into the constructor. The simplicity of this test class means it doesn't require any further refactoring.

Getting multiple speakers

Getting many speakers is much easier than getting a single speaker. We are not going to have to worry about not finding speakers. We are not going to have to handle any error conditions. All we have to test for is retrieving the correct number of speakers, and ensuring data safety the same as we have done for the other methods so far.

Let's start with retrieving all the speakers when there are no speakers:

```
[Fact]
public void ItReturnsNoSpeakersWhenThereAreNoSpeakers()
{
  // Arrange
  var repo = new SpeakerRepository();

  // Act
  var result = repo.GetAll();

  // Assert
  Assert.NotNull(result);
}
```

This is an easy fix in the repository:

```
public IQueryable<Speaker> GetAll()
{
  return new List<Speaker>().AsQueryable();
}
```

Remember, we are playing devil's advocate; this is the correct response for its purpose.

Now, we need to assert the type and size of the method's response:

```
// Assert
Assert.NotNull(result);
Assert.IsAssignableFrom<IQueryable<Speaker>>(result);
Assert.Equal(0, result.Count());
```

I know we have already mentioned the single assert rule and how it doesn't mean what it sounds like it means. This test is still following the rule because we are asserting that `GetAll` returns a non-null empty collection of speakers.

Next, we test for a single speaker present in the repository:

```
[Fact]
public void ItReturnsASingleSpeakerWhenOnlyOneSpeakerExists()
{
  // Arrange
  var repo = new SpeakerRepository();
  repo.Create(new Speaker { Name = "Test Speaker"});

  // Act
  var result = repo.GetAll();

  // Assert
  Assert.Equal(1, result.Count());
}
```

Again, making the appropriate adjustment in the repository is fairly simple:

```
public IQueryable<Speaker> GetAll()
{
  return Speakers.AsQueryable();
}
```

Next, we want to close off this line of testing by making sure that the speaker returned is the speaker we created. We expect this to pass right way because we already know that is the case. This assertion is important to ensure the future integrity of the repository.

```
// Act
var result = repo.GetAll().ToList();

// Assert
Assert.Single(result);
Assert.Equal("Test Speaker", result.First().Name);
```

Notice the change to the action. Executing a count and retrieving the first element in the collection both cause an enumeration. Enumerations equal CPU cycles. To reduce test time and production execution time, we want to reduce the number of enumerations. Converting the IQueryable to a list will enforce a single enumeration.

Let's do one final test to ensure the proper return values and check that multiple speakers come back correctly:

```
[Fact]
public void ItReturnsManySpeakersWhenManySpeakersExists()
{
  // Arrange
  var repo = new SpeakerRepository();
  repo.Create(new Speaker());
  repo.Create(new Speaker());
  repo.Create(new Speaker());

  // Act
  var result = repo.GetAll().ToList();

  // Assert
  Assert.Equal(3, result.Count);
}
```

This test is just a sanity check and passes right away. Our next test is going to check data integrity:

```
[Fact]
public void ItProtectsAgainstObjectChanges()
{
  // Arrange
  var repo = new SpeakerRepository();
  repo.Create(new Speaker {Name = "Test Name"});
  var speakers = repo.GetAll().ToList();
  speakers.First().Name = "New Name";

  // Act
  var result = repo.GetAll();

  // Assert
  Assert.NotEqual(speakers.First().Name, result.First().Name);
}
```

The update in the repository is similar to what we have done for the other methods with one exception. We are operating on an entire collection, instead of just a single item:

```
public IQueryable<Speaker> GetAll()
{
  return Speakers.Select(CloneSpeaker).AsQueryable();
}
```

Some of you may have seen this syntax before. What we have done is to pass the function pointer of the `CloneSpeaker` method in as the Lambda expression required by the `Select` method in linq. The preceding line is exactly the same, functionally, as the following:

```
public IQueryable<Speaker> GetAll()
{
  return Speakers.Select(CloneSpeaker).AsQueryable();
}
```

Time to clean up. We have the same clean-up steps as before. The only real refactoring needed is extracting the repository creation.

Updating a speaker

We can now create and retrieve speakers. We have also ensured that we cannot update them by accident. So, let's make sure we can update them on purpose:

```
[Fact]
public void ItUpdatesASpeaker()
{
  // Arrange
  var repo = new SpeakerRepository();
  var speaker = repo.Create(new Speaker {Name = "Test Name"});
  speaker.Name = "New Name";

  // Act
  var result = repo.Update(speaker);

  // Assert
  Assert.Equal(speaker.Name, result.Name);
}
```

Playing devil's advocate again, this is a simple update to the repository:

```
public Speaker Update(Speaker speaker)
{
  return speaker;
}
```

This is clearly the wrong solution; let's write another, more specific test:

```
[Fact]
public void ItUpdatesASpeakerInTheRepository()
{
  // Arrange
  var repo = new SpeakerRepository();
  var speaker = repo.Create(new Speaker {Name = "Test Name"});
  speaker.Name = "New Name";

  // Act
  var updatedSpeaker = repo.Update(speaker);

  // Audit
  var result = repo.Get(speaker.Id);

  // Assert
  Assert.Equal("New Name", result.Name);
}
```

Making this test pass is a little tricky, but not too bad:

```
public Speaker Update(Speaker speaker)
{
  var oldSpeaker = Speakers.FirstOrDefault(s => s.Id == speaker.Id);
  var index = Speakers.IndexOf(oldSpeaker);
  Speakers[index] = speaker;

  return speaker;
}
```

However, making this change causes the exists test to fail. Looking at that test, it is failing for a good reason and should not really pass given our current understanding of how `Update` is

supposed to work. Let's make a small but significant update to that test:

```
[Fact]
public void ItHasUpdateMethod()
{
  // Arrange
  IRepository<Speaker> repo = new SpeakerRepository();
  var speaker = repo.Create(new Speaker());

  // Act
  var result = repo.Update(speaker);
}
```

Next up on our testing timeline, is handling the error that was highlighted by the failure of the exists test. If someone asks to update a speaker that doesn't exist, the repository blows up. This could be the required behavior, but it should blow up with an informative exception, not an index out of bounds exception:

```
[Fact]
public void ItThrowsNotFoundExceptionWhenSpeakerDoesNotExist()
{
  // Arrange
  var repo = new SpeakerRepository();
  var speaker = new Speaker {Id = 5, Name = "Test Name"};

  // Act
  var result = Record.Exception(() => repo.Update(speaker));

  // Assert
  Assert.IsAssignableFrom<SpeakerNotFoundException>(result.GetBaseException());
}
```

A `SpeakerNotFoundException` already exists within the system, but this comes from a different layer so we need to create a new exception:

```
public class SpeakerNotFoundException : Exception
{
  public SpeakerNotFoundException(int id) : base($"Speaker {id} not found.")
  {
  }
}
```

As an exercise, see if you can cover this exception with tests. Being able to work backwards and write tests after the fact, as we will discuss in upcoming chapters, is a valuable skill to have.

Now, let's continue and make this test pass:

```
public Speaker Update(Speaker speaker)
{
  var oldSpeaker = Speakers.FirstOrDefault(s => s.Id == speaker.Id);
  var index = Speakers.IndexOf(oldSpeaker);

  if (index == -1)
  {
    throw new SpeakerNotFoundException(speaker.Id);
  }

  Speakers[index] = speaker;

  return speaker;
}
```

As with the other tests, we must now ensure data integrity:

```
[Fact]
public void ItProtectsAgainstObjectChanges()
{
  // Arrange
  var repo = new SpeakerRepository();
  var speaker = repo.Create(new Speaker {Name = "Test Name"});
  speaker.Name = "New Name";
  var updatedSpeaker = repo.Update(speaker);

  // Act
  updatedSpeaker.Name = "Updated Name";

  // Audit
  var result = repo.Get(updatedSpeaker.Id);

  // Assert
  Assert.NotEqual("Updated Name", result.Name);
}
```

We are bordering on test complexity that is too high. If these tests get much more complex, we will want to consider rethinking our approach. This test confirms what we thought: the returned speaker is not protected against change. Let's fix that:

```
public Speaker Update(Speaker speaker)
{
  var oldSpeaker = Speakers.FirstOrDefault(s => s.Id == speaker.Id);
  var index = Speakers.IndexOf(oldSpeaker);

  if (index == -1)
  {
    throw new SpeakerNotFoundException(speaker.Id);
  }

  Speakers[index] = speaker;

  return CloneSpeaker(speaker);
}
```

A fairly simple solution: just wrap the return with a `CloneSpeaker` call. Do you see another potential data integrity issue? What are we doing to protect against further changes to the speaker that was passed? Let's write a test to ensure that changes after the fact to the speaker that was passed in don't affect the repository:

```
[Fact]
public void ItProtectsAgainstOriginalObjectChanges()
{
  // Arrange
  var repo = new SpeakerRepository();
  var speaker = repo.Create(new Speaker { Name = "Test Name" });
  speaker.Name = "New Name";
  var updatedSpeaker = repo.Update(speaker);

  // Act
  speaker.Name = "Updated Name";

  // Audit
  var result = repo.Get(updatedSpeaker.Id);

  // Assert
  Assert.NotEqual("Updated Name", result.Name);
}
```

This test looks almost identical to the previous test. The only significant change is the action. Instead of changing the value on the `updatedSpeaker`, we are now changing the value on the original speaker. To make this test pass, the change to the repository is similar to the previous fix:

```
public Speaker Update(Speaker speaker)
{
  var oldSpeaker = Speakers.FirstOrDefault(s => s.Id == speaker.Id);
  var index = Speakers.IndexOf(oldSpeaker);

  if (index == -1)
  {
    throw new SpeakerNotFoundException(speaker.Id);
  }

  Speakers[index] = CloneSpeaker(speaker);

  return CloneSpeaker(speaker);
}
```

All that is left to refactor and clean up are the tests and repository.

Deleting a speaker

We are finally at the last method in this repository. For Speaker Meet, we don't really want to delete the speakers; we might flag a speaker as deleted, but we don't really want to remove them from the dataset. So, our `Delete` method will be more like an update.

We don't have any strange behavior or constraints for `Delete`, so we should start with the failure case. What should happen if we ask to delete a user that doesn't exist? Should we throw an exception? In this case, we can actually just assume that the job is done. If there is no speaker with the given ID then the deletion of that speaker could be considered a success.

Well, now that we have considered the failure case, we see that it does have special behavior that would cause the failure case to simply pass straight away. Let's take a look at the success case and see if it would cause a failing test:

```
[Fact]
public void ItMarksTheGivenSpeakerAsDeleted()
{
  // Arrange
  var repo = new SpeakerRepository();
  var speaker = repo.Create(new Speaker {Name = "Test Name"});

  // Act
  repo.Delete(speaker);

  // Audit
  var result = repo.Get(speaker.Id);

  // Assert
  Assert.True(result.IsDeleted);
}
```

Making this test pass should be fairly easy: we can just call into the local update method after changing the `isDeleted` flag to `true`:

```
public void Delete(Speaker speaker)
{
  speaker.IsDeleted = true;

  Update(speaker);
}
```

Don't forget to fix any other tests that fail as a result of adding this code, after making sure the failure is for a valid reason. If tests fail while we are writing more tests we need to check the requirements and make sure they are not in disagreement with each other. We never want to break existing requirements unintentionally.

We now have to deal with what happens if the speaker didn't exist in the context. As decided earlier, we are okay with just ignoring this request as a failure to delete a non-existing speaker results in the same situation as successfully deleting an existing speaker:

```
[Fact]
public void ItDoesNothingWhenDeletingANonexistingSpeaker()
{
```

```
  // Arrange
  var repo = new SpeakerRepository();
  var speaker = new Speaker();

  // Act
  var result = Record.Exception(() => repo.Delete(speaker));

  // Assert
  Assert.Null(result);
}
```

To pass this one we have to do something that is normally not recommended or preferred. We have to swallow an exception. Before taking a step like this and ignoring an exception, make sure you only ignore a specific exception and make sure you have thought it through thoroughly:

```
public void Delete(Speaker speaker)
{
  speaker.IsDeleted = true;

  try
  {
    Update(speaker);
  }
  catch (SpeakerNotFoundException ex)
  {
    // We can assume non-existing speakers are deleted
  }
}
```

And for the last test for this method, we must make sure that we don't accidently pollute the object that was passed in:

```
[Fact]
public void ItProtectsAgainstPassedObjectChanges()
{
  // Arrange
  var repo = new SpeakerRepository();
  var speaker = repo.Create(new Speaker {Name = "Test Name"});

  // Act
  repo.Delete(speaker);

  // Assert
  Assert.False(speaker.IsDeleted);
}
```

We can use the same method we have been using to ensure data integrity:

```
speaker = CloneSpeaker(speaker);
speaker.IsDeleted = true;
```

Genericizing the repository

This repository is great, but do we really want to repeat this logic and all these tests for every single data model that we need to be retrieved from some data source? The answer is no; if you find yourself doing the same thing over and over as a developer, you are doing something wrong.

So, how can we protect ourselves from that drudgery?

One way is to use generics. Let's refactor the `SpeakerRepository` to use generics, this will also involve refactoring many of the tests to make them apply to the `GenericRepository` instead of the concrete `SpeakerRepository`.

Step one – abstract interface

In the `IRepository`, everywhere we use `Speaker` we need to replace it with C# generics:

```
public interface IRepository<T>
{
  T Create(T item);
  T Get(int id);
  IQueryable<T> GetAll();
  T Update(T item);
  void Delete(T item);
}
```

This change will cause a break in the `SpeakerRepository` that we need to fix. Right now, we are chasing the compiler and leaning on it to tell us what we are breaking. Once the tests pass again, we will know we are back to good:

```
public class SpeakerRepository : IRepository<Speaker>
```

Now we have to make some changes in the SpeakerService as it is no longer able to reference a simple IRepository but instead needs an `IRepository<Speaker>`:

```
public class SpeakerService : ISpeakerService
{
  private readonly IRepository<Speaker> _repository;

  public SpeakerService(IRepository<Speaker> repository)
  {
    _repository = repository;
  }
…
```

We must update the fake repository to use the correct generic repository type.

```
public class FakeRepository : IRepository<Speaker>
```

And lastly, we have several tests that need to be updated. No secret code there; just update `IRepository` to `IRepository<Speaker>`.

Step two – abstract the concrete class

Now that the interface is generic, we can start to work on the `SpeakerRepository`. First let's rename it to `InMemorySpeakerRepository`. Now, we want to start using generics. Create a new class, `InMemoryRepository<T>` and have the speaker repository inherit from it:

```
public abstract class InMemoryRepository<T> : IRepository<T>
{
  public abstract T Create(T speaker);
  public abstract T Get(int id);
  public abstract IQueryable<T> GetAll();
  public abstract T Update(T speaker);
  public abstract void Delete(T speaker);
}
```

In order to move slow and have the tests passing as much as possible, we are using abstract and will have to have each method in the speaker repository override the base class methods. This gives us the ability to move each method individually into the abstract as we discover what needs to be done, instead of trying to tackle the whole problem all at once.

Inherit from the in-memory repository and add override to each inherited method:

```
public class InMemorySpeakerRepository : InMemoryRepository<Speaker>
public overrideSpeaker Create(Speaker speaker)
public override Speaker Get(int id)
public override IQueryable<Speaker> GetAll()
public override Speaker Update(Speaker speaker)
public override void Delete(Speaker speaker)
```

Converting Create to a generic method

We will begin our generic method journey with `Create`. The first step is to copy the body of the `Create` method from the speaker repository to the generic repository. In order to do this, we must change the abstract `Create` method to a virtual `Create` method:

```
public virtual T Create(T speaker)
{
  var newSpeaker = CloneSpeaker(speaker);
  newSpeaker.Id = ++CurrentId;
  Speakers.Add(newSpeaker);

  return CloneSpeaker(newSpeaker);
}
```

Instantly, we run into trouble. We don't have a generic clone method. For now let's fake it until we make it.

Add the following protected abstract method to the generic repository:

```
protected abstract T CloneEntity(T entity);
```

Now, we can make similar changes in the code we copied from the speaker repository's `Create` method:

```
protected readonly IList<T> Entities = new List<T>();
protected int CurrentId;

public virtual T Create(T entity)
{
  var newSpeaker = CloneEntity(entity);
  newSpeaker.Id = ++CurrentId;
  Entities.Add(newSpeaker);

  return CloneEntity(newSpeaker);
}
```

Its's looking much better now, but we still have one issue. We can't expect `Id` to exist on just any object. The compiler is quite mad about this right now. There are several solutions, but we are going to create a simple data model interface and place a constraint on `T` that it must inherit from that interface. The only thing in the interface will be a property of `Id` with an integer type:

```
public interface IIdentity
{
  int Id { get; set; }
}

public abstract class InMemoryRepository<T> : IRepository<T> where T: IIdentity
```

This causes a break in the speaker repository. We must also have Speaker inherit from `IIdentity`. Now, we have relocated all of the logic in the `Create` method into the generic repository. Delete

the `create` method in the speaker repository.

Many tests fail because we need to repoint the other methods in the speaker repository to use the backing objects from the generic repository. Go through the speaker repository and update all the references to Speakers to Entities, instead.

We also need to make an adjustment to the `TestableSpeakerRepository` class:

```
internal class TestableSpeakerRepository : InMemorySpeakerRepository
{
  public IQueryable<Speaker> SpeakersCollection => Entities;
}
```

Converting Get to a generic method

Next on the list is the `Get` method. Just like with the `Create` method, copy all the contents into the generic repository and fix any errors that occur:

```
public virtual T Get(int id)
{
  var entity = Entities.SingleOrDefault(e => e.Id == id);

  if (entity != null)
  {
    entity = CloneEntity(entity);
  }

  return entity;
}
```

This method turns out to be an easy one to copy. Now, delete the existing method in the speaker repository. No tests break this time so we can move on to the next method.

Converting GetAll to a generic method

`GetAll` is the easiest of the methods to convert. It doesn't even reference speakers textually:

```
public virtual IQueryable<T> GetAll()
{
  return Entities.Select(CloneEntity).AsQueryable();
}
```

Delete the existing method in the speaker repository and move on to the next method, update.

Converting Update to a generic method

The process for update is the same as with the other methods. Copy the body of the existing code, rename any speaker references to entity references, and then delete the existing method:

```
public virtual T Update(T entity)
{
  var oldEntity = Entities.FirstOrDefault(s => s.Id == entity.Id);
  var index = Entities.IndexOf(oldEntity);

  if (index == -1)
  {
    throw new EntityNotFoundException(entity.Id);
  }

  Entities[index] = CloneEntity(entity);

  return CloneEntity(entity);
}
```

Converting Delete to a generic method

`Delete` is a different story: every object is likely to have different requirements for deleting it. Some will actually be deleted and others, such as Speaker, will merely be flagged as deleted. For this and many other reasons, we choose to leave the `Delete` implementation up to concrete repositories and the generic repository will throw a not implemented exception instead.

Let's write a `Delete` test class just for this functionality. It should be short and quick:

```
public class Delete
{
  [Fact]
  public void ItThrowsNotImplementException()
  {
    // Arrange
    var repo = new InMemoryRepository<TestEntity>();

    // Act
    var result = Record.Exception(() => repo.Delete(new TestEntity()));

    // Assert
    Assert.IsAssignableFrom<NotImplementedException>
    (result.GetBaseException());
     Assert.Equal("Delete is not avaliable for TestEntity",
      result.Message);
  }
}

public class TestEntity : IIdentity
{
  public int Id { get; set; }
}
```

Writing this test also makes us change the generic repository from an abstract class to a normal class. Changing the class type makes us change the `CloneEntity` method from abstract to virtual:

```
protected virtual T CloneEntity(T entity)
{
  return entity;
}
```

Now, we can write the method that will pass the test:

```
public virtual void Delete(T speaker)
{
  throw new NotImplementedException($"Delete is not avaliable for {typeof(T).Name}");
}
```

Step three – reorient the tests to use the generic repository

We began this process when we dealt with the `Delete` method. But let's continue with the other methods. All the methods that were moved into the generic repository can have most of their tests moved as well.

We will be leaving the data integrity tests because they are directly tied to the functionality in the speaker repository. We will leave all the delete tests for the same reason.

InMemoryRepository Create tests

To implement the rest of the functionality for the `InMemoryRepository`, we will start with the tests for the `Create` method:

```
public class Create
{
  private readonly TestableEntityRepository _repo;

  public Create()
  {
    _repo = new TestableEntityRepository();
  }

  [Fact]
  public void ItExists()
  {
    // Act
    var result = _repo.Create(new TestEntity());
  }

  [Fact]
  public void ItAddsAEntityToTheRepository()
  {
    // Act
    var result = _repo.Create(new TestEntity());

    // Assert
    Assert.Equal(1, _repo.EntityCollection.Count());
  }

  [Fact]
  public void ItAssignsUniqueIdsToEachEntity()
  {
    // Act
    var entity1 = _repo.Create(new TestEntity());
    var entity2 = _repo.Create(new TestEntity());

    // Assert
    Assert.NotEqual(entity1.Id, entity2.Id);
  }
}

internal class TestableEntityRepository : InMemoryRepository<TestEntity>
{
  public IQueryable<TestEntity> EntityCollection => Entities;
}
```

InMemoryRepository Get tests

Now that we can create using the `InMemoryRepository`, we should be able to accurately test the `Get` method:

```csharp
public class Get
{
  private readonly InMemoryRepository<TestEntity> _repo;

  public Get()
  {
    _repo = new InMemoryRepository<TestEntity>();
  }

  [Fact]
  public void ItExists()
  {
    // Act
    var result = _repo.Get(0);
  }

  [Fact]
  public void ItReturnsAnEntityWhenFound()
  {
    // Arrange
    var entity = _repo.Create(new TestEntity() { Name = "Test Entity" });

    // Act
    var result = _repo.Get(entity.Id);

    // Assert
    Assert.NotNull(result);
    Assert.Equal("Test Entity", result.Name);
  }

  [Fact]
  public void ItReturnsNullWhenNotFound()
  {
    // Act
    var result = _repo.Get(-1);

    // Assert
    Assert.Null(result);
  }
}
```

InMemoryRepository GetAll tests

With getting a single object working, lets test getting all the objects:

```csharp
public class GetAll
{
  private readonly InMemoryRepository<TestEntity> _repo;

  public GetAll()
  {
    _repo = new InMemoryRepository<TestEntity>();
  }

  [Fact]
  public void ItExists()
  {
    // Act
    var result = _repo.GetAll();
  }

  [Fact]
  public void ItReturnsNoEntitiesWhenThereAreNoEntities()
  {
    // Act
    var result = _repo.GetAll();

    // Assert
    Assert.NotNull(result);
    Assert.IsAssignableFrom<IQueryable<TestEntity>>(result);
    Assert.Equal(0, result.Count());
  }

  [Fact]
  public void ItReturnsASingleEntityWhenOnlyOneEntityExists()
  {
    // Arrange
    _repo.Create(new TestEntity { Name = "Test Entity" });

    // Act
    var result = _repo.GetAll().ToList();

    // Assert
    Assert.Equal(1, result.Count);
    Assert.Equal("Test Entity", result.First().Name);
  }

  [Fact]
  public void ItReturnsManyEntitiesWhenManyEntitiesExist()
  {
    // Arrange
    _repo.Create(new TestEntity());
    _repo.Create(new TestEntity());
    _repo.Create(new TestEntity());

    // Act
    var result = _repo.GetAll().ToList();

    // Assert
    Assert.Equal(3, result.Count);
  }
}
```

InMemoryRepository Update tests

Last but not least, we will add the ability to update records using the `InMemoryRepository`:

```
public class Update
{
  private readonly InMemoryRepository<TestEntity> _repo;

  public Update()
  {
    _repo = new InMemoryRepository<TestEntity>();
  }

  [Fact]
  public void ItExists()
  {
    // Arrange
    var entity = _repo.Create(new TestEntity());
    // Act
    var result = _repo.Update(entity);
  }

  [Fact]
  public void ItUpdatesAnEntity()
  {
    // Arrange
    var entity = _repo.Create(new TestEntity(){ Name = "Test Name" });
    entity.Name = "New Name";
    // Act
    var result = _repo.Update(entity);
    // Assert
    Assert.Equal(entity.Name, result.Name);
  }

  [Fact]
  public void ItUpdatesAnEntityInTheRepository()
  {
    // Arrange
    var entity = _repo.Create(new TestEntity() { Name = "Test Name" });
    entity.Name = "New Name";
    // Act
    var updatedEntity = _repo.Update(entity);
    // Audit
    var result = _repo.Get(entity.Id);
    // Assert
    Assert.Equal("New Name", result.Name);
  }

  [Fact]
  public void ItThrowsNotFoundExceptionWhenEntityDoesNotExist()
  {
    // Arrange
    var entity = new TestEntity { Id = 5, Name = "Test Name" };
    // Act
    var result = Record.Exception(() => _repo.Update(entity));
    // Assert
    Assert.IsAssignableFrom<EntityNotFoundException>(result.GetBaseException());
  }
}
```

We now have a generic repository that can be used with any data model as long as that data model has an ID. We also have a way to ensure data integrity if we need to, by inheriting and creating a method to clone the data object that we need to protect.

Entity Framework

Object-Relation Mapping (**ORM**) frameworks such as Entity Framework help increase productivity and optimize code reuse and maintainability. However, you should not tightly couple your application to the ORM. Abstract away ORMs such as Entity Framework just as you would any third-party library or system.

Speaker Meet uses Entity Framework through a generic repository; in order to get started, add a NuGet reference to Entity Framework – Sql Server.

```
Microsoft.EntityFrameworkCore.SqlServer
```

Next, add a connection string to the `appsettings.json`:

```
"ConnectionStrings": {"DefaultConnection":
"Server=.;Database=SpeakerMeetBook;Trusted_Connection=True;MultipleAct
 iveResultSets=true"}
```

DbContext

The latest version of Entity Framework, EF Core 2.0, has added DbContext pooling, which helps improve performance by saving some of the cost of initializing a new instance of the `DbContext` with each request.

Modify the `startup.cs` ConfigureServices to reference the connection string:

```
var connectionString =
  Configuration.GetConnectionString("DefaultConnection");
services.AddDbContextPool<SpeakerMeetContext>(options =>
  options.UseSqlServer(connectionString));
```

Add a `DbContext` to your application. Create a new file named `SpeakerMeetContext` with options as follows:

```
using Microsoft.EntityFrameworkCore;

namespace SpeakerMeet.Api.Entities
{
  public class SpeakerMeetContext : DbContext
  {
    public SpeakerMeetContext(DbContextOptions<SpeakerMeetContext> options) : base(options)
    { }

    public virtual DbSet<Speaker> Speakers { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      modelBuilder.Entity<Speaker>().ToTable("Speaker");
    }
  }
}
```

Models

Entity Framework models may contain additional information that you might not necessarily want to expose to other parts of the system or application consumers. Create a new model named `Speaker`. This will be the model used by Entity Framework and the generic repository. The services or business layer will be responsible for converting the Entity Framework models to **Data Transfer Objects** (**DTOs**) or ViewModels used by the rest of the application:

```
using System.ComponentModel.DataAnnotations;

namespace SpeakerMeet.Api.Entities
{
  public class Speaker
  {
    public int Id { get; set; }

    [Required]
    [StringLength(50)]
    public string Name { get; set; }

    [Required]
    [StringLength(50)]
    public string Location { get; set; }

    [Required]
    [StringLength(255)]
    public string EmailAddress { get; set; }

    public bool IsDeleted { get; set; }
  }
}
```

Generic repository

In order to see the Entity Framework specific Generic repository run you may want to add the following class to your application:

```csharp
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;

namespace SpeakerMeet.Api.Repository
{
  public class Repository<T> : IRepository<T> where T : class
  {
    private readonly DbSet<T> _dbSet;
    protected readonly DbContext Context;

    public Repository(DbContext context)
    {
      Context = context;
      _dbSet = context.Set<T>();
    }

    public T Create(T entity)
    {
      throw new NotImplementedException();
    }

    public T Get(int id)
    {
      return _dbSet.Find(id);
    }

    public IQueryable<T> GetAll()
    {
      return _dbSet;
    }

    public T Update(T speaker)
    {
      throw new NotImplementedException();
    }

    public void Delete(T entity)
    {
      throw new NotImplementedException();
    }
  }
}
```

Dependency Injection

In order to wire everything up, the Speaker Meet application leverages the built in **Dependency Injection** (**DI**) container. Dependency Injection allows the system to be loosely coupled. There are ways to achieve this without the use of a container (poor man's DI, and so on) which will be covered in a later chapter. The tests themselves do not rely on DI, instead opting to instantiate classes as needed.

Wire it all up

In order to configure the DI container, add the following to ConfigureServices in `startup.cs`:

```
services.AddSingleton(typeof(DbContext), typeof(SpeakerMeetContext));
services.AddScoped(typeof(IRepository<>), typeof(Repository<>));
services.AddTransient<ISpeakerService, SpeakerService>();
services.AddTransient<IGravatarService, GravatarService>();
```

This will avoid the need to instantiate the `SpeakerService` from within the `SpeakerController`. The DI container handles this for you.

If you've created your database and populated the speaker table, you now should be able to run the application and hit the `SpeakerController` endpoints. Give it a shot!

- `http://localhost:41436/api/speaker/`
- `http://localhost:41436/api/speaker/1`
- `http://localhost:41436/api/speaker/search?searchString=test`

Postman

There are a variety of tools available for manually exercising an API. Postman is just one such tool and is a favorite in the industry. Postman offers a lot of functionality to help you with your API development and testing. It's worth a look if you're interested.

To install Postman, simply visit the website (http://www.getpostman.com) and follow the instructions. With the Speaker Meet application running, enter the URL (example: `http://localhost:41436/api/speaker/search`) into the box, add params (example: `[{"key":"searchString","value":"te","description":""}]`), and hit Send. The response shows as JSON by default in the body of the message.

This can be an extremely powerful tool. As the complexity and feature set of the Speaker Meet API grows, more complex messages can be sent with `POST`, `PUT`, `PATCH`, and `DELETE`. These will be covered in a later chapter.

Summary

This chapter was all about abstraction: when and how to use it. You should now see why it's so important to create abstractions between code you write yourself and that of a third party. You should also have a good idea of how those abstractions can be achieved.

In this chapter, we abstracted a Gravatar service, extended the repository pattern, and used a generic repository for Entity Framework.

In Chapter 9, *Testing JavaScript Applications*, we'll focus on testing JavaScript applications. We'll walk through creating a React application and discuss different approaches to testing a JavaScript application.

Testing JavaScript Applications

To get started testing in JavaScript, we will need to create a ReactJS application and configure it for testing using the Mocha, Chai, Enzyme, and Sinon libraries.

These steps were discussed in detail in Chapter 3, *Setting up the JavaScript Environment*, so here, we will simply walk through the steps and not explain them in detail.

The goals for this chapter are:

- Create the Speaker Meet React application
- Talk through our plan of attack for testing the application:
  - What is our approach?
  - What parts of the app can we even test?
  - What part of the app do we start with?
- Write tests and complete a couple of features for the application:
  - Speaker listing
  - Speaker detail

Once this chapter is finished, you should be capable of unit-testing any React-based application.

Creating a React app

For the application in this book, to maintain compatibility, you will want to use Node.js version 8.5.0, NPM version 5.4.2, and create-react-app version 1.4.0.

Execute the following commands to install and execute the app:

```
>npm install
>npm test
>npm start
```

All three commands should run successfully. After running `npm test`, you will need to exit the test run by hitting `<q>`. After running `npm start`, you will need to exit the server by hitting *Ctrl + C*.

Ejecting the app

Assuming the previous step went without a hitch, we can proceed to eject the React app. Again, as it has already been explained in detail in Chapter 3, *Setting up the JavaScript Environment,* we will only do a short review here.

There is only a single command to eject the application. After ejection, we will want to rerun the commands in the previous section to ensure that the application still works as expected.

Execute the following command to eject:

```
>npm run eject
```

Configuring Mocha, Chai, Enzyme, and Sinon

Now, we are ready to add the testing facilities that we would like to use for this app. As before, the addition of these utilities has been covered in detail in a previous chapter. So, we will only be providing the commands to execute and the versions of the packages to install.

Execute the following commands to install the libraries we are going to use:

```
>npm install mocha@3.5.3
>npm install chai@4.1.2
>npm install enzyme@2.9.1
>npm install sinon@3.2.1
```

There are also a few other libraries we will be using as part of our Redux workflow:

```
>npm install nock@9.0.1
>npm install react-router-dom@4.2.2
>npm install redux@3.7.2
>npm install redux-mock-store@1.3.0
>npm install redux-thunk@2.2.0
```

Including the version in the install command will ensure that you are using the same version of the libraries that we are and will reduce the number of potential issues.

To use the libraries we have just installed, we will also need to install an extra preset for babel:

```
>npm install babel-preset-es2015@6.24.1
```

Update your babel config in `package.json` to remove react-app and include `react` and `es2015`.

```
"babel": {
  "presets": [
    "react",
    "es2015"
  ]
},
```

As described in Chapter 3, *Setting up the JavaScript Environment*, delete the test configuration section from `package.json`. Then, update the test script to:

```
"test": "mocha --require ./scripts/test.js --compilers babel-core/register ./src/**/*.spec.js'
```

And add a test watch script:

```
"test:watch": "npm test -- -w"
```

We are now ready to update the test execution file `test.js` in the scripts folder so it's compatible with Mocha. Change all the contents of the file to:

```
'use strict';
```

```
import jsdom from 'jsdom';
global.document = jsdom.jsdom('<html><body></body></html>');
global.window = document.defaultView;
global.navigator = window.navigator;

function noop() {
  return {};
}

// prevent mocha tests from breaking when trying to require a css file
require.extensions['.css'] = noop;
require.extensions['.svg'] = noop;
```

The last step before we can use our new testing libraries is to update the `App.test.js` file to match the conventions used with Mocha and Chai. So, change the filename to `App.spec.js` and update the contents to match the code shown here:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { expect } from 'chai';

import App from './App';

describe('(Component) App', () => {
  it('renders without crashing', () => {
      const div = document.createElement('div');
      ReactDOM.render(<App />, div);
    });
});
```

Now, as before, execute the test script and start the application to make sure nothing broke during our transformation to Mocha.

```
>npm test
>npm run test:watch
>npm start
```

All three of those commands should work. If you have an issue, check all the steps we have just discussed and look to Chapter 3, *Setting Up a JavaScript Environment,* for a more detailed explanation.

The plan

Now that our testing config has been updated and is working correctly, we can begin thinking about test-driving our first feature.

In earlier chapters, we discussed where to start testing and decided that if possible an inside-out approach is preferred. To keep with that approach, we want to determine the different parts of our React app so that we can target the purest business logic we can.

Right off the bat, regardless of any other architectural choices, we can identify the React component and a service representing communication with our data source. We are planning to use Redux in this app so that makes up the missing piece and connects our component with our data.

Which one of these is the business logic though? Out of those base options, what would we even test? Let's examine each one a little more closely and see what we could test that would be considered a unit test.

Considering the React component

Generally, we want to avoid unit-testing third-party libraries. So, let's separate the third-party aspects of a React component from the parts that we would potentially unit-test.

The third-party aspects include any inherited features and functionality; this includes to some degree any life cycle methods and the JSX. So, what's left? The answer to this question depends on whether the component in question is a presentational component or a container component.

Presentational components are almost pure HTML and view mechanisms. There is almost no traditionally unit-testable behavior. Certainly, there is no real business logic.

Container components are where the real action happens in a React application. These components can manipulate data and make business decisions that can control the flow of the application. So, let's keep container components in the list of possible places to start our unit-testing efforts.

Looking at Redux testability

Redux is a third-party library, that controls data flow throughout the application and manages quite a bit of the normal data shuffling that we may want to unit test. Because it is third-party though, on the surface there doesn't seem to be too much that we can unit test. Let's take a closer look at the aspects of the Redux data flow to determine if there really is nothing to test or if we still need to unit test parts of Redux.

The store

The Redux store is where all the data lives after it has been acquired by the application. Typically, there is only one store for each application using Redux. The store is almost completely contained within the Redux library and we have very few direct interactions with it. For this reason, there doesn't seem to be much we would or could test for the store and it falls squarely in the realm of third-party code that we must simply trust.

Actions

Actions in Redux represent an event carrying a data packet. The event is usually a command to either retrieve or update data within the data source which should be reflected by the store. Because actions are just a key with some data attached, there doesn't seem to be much to test here.

Reducers

If there is anything to test within the Redux interactions, it is likely in the reducers. Reducers receive the actions and determine what to do, if anything, based on the actions requested and the data provided as part of those actions.

Typically, the reducer is going to simply call the API service once the appropriate service call is determined. It is possible that a reducer might also map the received data into a format that is more appropriate to the service call that must be made.

So, if the reducer is, in all reality, just going to call the service, what would we test for the reducer? Other than ensuring that the appropriate service method is called with the appropriate data there doesn't seem to be much. For completeness, we would want to test those things, but they do not represent the core of our business logic.

In conclusion, it doesn't appear that much is testable in Redux and what is testable doesn't represent the core of our business logic.

Unit-testing an API service

Lastly, let's look at the API service. Normally, the service in a front-end application behaves much like the repository in a back-end application. The service's main function is to abstract data interactions with some data source. Those interactions don't necessarily contain any definable business logic. The real logic, if any, for a service exists on the server and doesn't need to be tested as part of a front-end application. At least it doesn't need to be tested the way you might think it does.

So, if the service doesn't contain any business logic, and Redux doesn't contain much business logic, and the components don't contain much business logic, what do we test and how can it be unit-tested?

The short answer is that we are not off the hook for testing, but we will have to jump through some hoops to do any testing because it is difficult to remove ourselves from integration testing. In a typical front-end application, unlike in C#, there is no clear division between our code and their code. So, we will have to make some concessions and write quite a bit of code to abstract parts of third-party code to allow us to test what we need to be testing.

So, where does this leave us when it comes to a testing direction? Unfortunately, there doesn't seem to be a clear winner. For the purposes of this application, we will work from the data source up so that we have a clear understanding of the data manipulations available to us while we write user interface aspects of the application.

Speaker listing

Following the functionality in our C# backend, we will start by testing a listing of the speakers available. We are not yet ready to connect to the backend and, for any of the tests we will write here as unit tests, we will need to mock the behaviors that the backend would normally present.

For the moment, we are not going to concern ourselves with any kind of authentication. So, the important functionality we will be looking to implement is that when no speakers exist we should let the user know, and when speakers do exist we should list them.

The way that we will produce both situations is through a mock API. As strange as it may seem, most of our business logic will be in the mock API. Because it will be crucial to all of the other tests we will write, we must unit test the mock API as if it were production code.

A mock API service

To begin testing the mock API service, let's create a new services folder and add a `mockSpeakerService.spec.js` file.

Inside that file, we need to import our assertion library, create our initial describe, and write an existence test.

```
import { expect } from 'chai';

describe('Mock Speaker Service', () => {
  it('exits', () => {
    expect(MockSpeakerService).to.exist;
  });
});
```

Start the npm test script with watch. The test we just wrote should fail. To make the test pass, we must create a `MockSpeakerService` object. Let's play devil's advocate a little and create an object in this file, but only enough of an object to make the test pass.

```
let MockSpeakerService = {};
```

This line passes the currently failing test, but clearly isn't what we are after. It does, however, force us to write more robust tests. The next test we can write is one that proves that the `MockSpeakerService` can be constructed. This test should ensure that we have defined the `MockSpeakerService` as a class.

```
it('can be constructed', () => {
  // arrange
  let service = new MockSpeakerService();

  // assert
  expect(service).to.be.an.instanceof(MockSpeakerService);
});
```

This test fails, stating that `MockSpeakerService` is not a constructor. The way to fix this is to change `MockSpeakerService` into a class.

```
class MockSpeakerService {
}
```

Now that we have a class that can be instantiated, the next test we write can start to test actual functionality. So, what functionality are we going to test? Looking at the requirements, we can see that the first scenario involves requesting all the speakers and receiving no speakers. That's a reasonably simple scenario to test. What would we call the function in the `MockSpeakerService` that would get all the speakers? Because we are trying to get all the speakers, a simple name that would not be redundant and fits the repository pattern we discussed in the C# backend is simply `getAll`. Let's create a nested describe and an existence test for a `getAll` class method.

```
describe('Get All', () => {
  it('exists', () => {
    // arrange
    let service = new MockSpeakerService();
```

```
      // assert
      expect(service.getAll).to.exist;
    });
  });
```

As per usual, this test should fail and it should fail with `expected undefined to exist`. Making this test pass is relatively simple, just add a `getAll` method to the `MockSpeakerService` class.

```
class MockSpeakerService {
  getAll() {
  }
}
```

The next thing we need to decide is the result we should expect when there are no speakers. Looking back at the backend, we should be receiving an empty array when no speakers are present. Looking at the requirements, the system should present a `NO_SPEAKERS_AVAILABLE` message. Should the service be responsible for displaying that message? In this case, the answer is no. The react component should be responsible for displaying the `NO_SPEAKERS_AVAILABLE` message when we get to that portion of the code. For now, we should expect, when no speakers exist, to receive an empty data set.

Because we are extending the context of the test, let's create another describe for that context extension.

```
describe('No Speakers Exist', () => {
  it('returns an empty array', () => {
    // arrange
    let service = new MockSpeakerService();

    // act
    let promise = service.getAll();

    // assert
    return promise.then((result) => {
      expect(result).to.have.lengthOf(0);
    });
  });
});
```

Notice the syntax we used for this test. We return the promise and make our assertions inside the then function. This is because we want our test to operate on asynchronous code from our service. The majority of backend operations will need to be asynchronous and one convention for dealing with that asynchronicity is to use promises. Asynchronous tests, that is, tests dealing with promises, in Mocha require that the promise be returned from the test so that Mocha can know to wait for the promise to resolve before closing out the test.

And now, to make the test pass, all we need to do is return a promise that resolves with an empty array from the `getAll` method. We are going to use a zero delay `setTimeout` here which will set us up to implement some kind of delay for development purposes later on. The reason we want a delay is so that we can test the operation of the UI in the event of a slow network response.

```
getAll() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(Object.assign([], this._speakers));
    }, 0);
  });
}
```

Now we have the first scenario passing and enough code to warrant a refactoring. We are declaring the service variable in multiple places and we don't have a context that represents a baseline instantiation of that variable. Let's create a describe to wrap all the post instantiation tests and add a `beforeEach` to initialize a service variable scoped to that describe and available to all the tests within it.

Here are the tests after the refactoring:

```
describe('Mock Speaker Service', () => {
   it('exits', () => {
     expect(MockSpeakerService).to.exist;
   });

   it('can be constructed', () => {
     // arrange
     let service = new MockSpeakerService();

     // assert
     expect(service).to.be.an.instanceof(MockSpeakerService);
   });

   describe('After Initialization', () => {
     let service = null;

     beforeEach(() => {
       service = new MockSpeakerService();
     });

     describe('Get All', () => {
       it('exists', () => {
         // assert
         expect(service.getAll).to.exist;
       });

       describe('No Speakers Exist', () => {
         it('returns an empty array', () => {
           // act
           let promise = service.getAll();

           // assert
           return promise.then((result) => {
             expect(result).to.have.lengthOf(0);
           });
         });
       });
     });
   });
});
```

The next scenario, the speaker listing, is for when speakers do exist. The first test for this scenario will need to add at least one speaker to the mock API. Let's create a new describe inside `GetAll` but separate from `No Speakers Exist`.

```
describe('Speaker Listing', () => {
   it('returns speakers', () => {
     // arrange
     service.create({});

     // act
     let promise = service.getAll();

     // assert
     return promise.then((result) => {
       expect(result).to.have.lengthOf(1);
     });
   });
});
```

We have added, as part of the setup for this test, a reference to a `create` method. This method does not yet exist and our test can't pass without it. So, we need to temporarily ignore this test and write tests for `create`. We can ignore this test by skipping it.

```
it.skip('returns speakers', () => {
```

Now, we can write a new describe block inside the `After Initialization` block for `create`.

```
describe('Create', () => {
   it('exists', () => {
     expect(service.create).to.exist;
   });
 });
```

And to make the test pass we add the `create` method to the mock service class.

```
class MockSpeakerService {
   create() {

   }
…
```

We could, from this point, write a few tests to add validation logic to the `create` method. However, we don't currently have any scenarios that reference a `create` method on the API. Since this method exists only for testing purposes, we are going to leave it alone with just an exists test. Let's move back to our scenario test.

Now that `create` exists, we should receive the failure that the test is expecting, which is that we expected a length of 1 but instead we have a length of 0. Remove skip from the test and verify.

To make this test pass, we essentially have to implement the basic logic for create and make a modification to `getAll`.

```
class MockSpeakerService {
   constructor() {
     this._speakers = [];
   }

   create(speaker) {
     this._speakers.push(speaker);
   }

   getAll() {
     return new Promise((resolve, reject) => {
       setTimeout(() => {
         resolve(Object.assign([], this._speakers));
       }, 0);
     });
   }
 }
```

We can consider the current tests sufficient to move forward and start testing our data flow.

The Get All Speakers action

To begin testing with Redux, there are a few testing entry points we could start with. We could begin by testing actions, reducers, or even interactions with the store. The store tests would be more integration tests and we want to concentrate on unit tests in this chapter. That leaves actions and reducers. Either is a fine place to start, but we will start with actions because they are extremely simple and uncomplicated as a concept for testing.

The action that we need right now is one to request the retrieval of speaker information; in essence, a get all speakers action. As stated earlier, actions can be extremely simple; however, we have an issue in that our get all speakers service call is asynchronous. Actions were not really designed to handle asynchronous calls. For that reason, let's start with something a little bit simpler and we will come back to this problem after we understand how to test a normal action.

Testing a standard action

We will need an action to notify Redux that we have the speakers after they have been loaded. There is no reason why we can't start there. So, let's write a test for the successful retrieval of speakers.

```
import { expect } from 'chai';

 describe('Speaker Actions', () => {
   describe('Sync Actions', () => {
     describe('Get Speakers Success', () => {
       it('exists', () => {
         expect(getSpeakersSuccess).to.exist;
       });
     });
   });
 });
```

Running this test should fail. To make the test pass, define a function named `getSpeakersSuccess`.

```
function getSpeakersSuccess() {
 }
```

Because of the simplicity of a typical action, our next test will essentially test the functionality of the action. We could break this into multiple tests, but all we are really doing is asserting on the structure of the data returned. Concerning the single assert rule, we are still only asserting one thing.

```
it('is created with correct data', () => {
   // arrange
   const speakers = [{
     id: 'test-speaker',
     firstName: 'Test',
     lastName: 'Speaker'
   }];

   // act
   const result = getSpeakersSuccess(speakers);

   // assert
   expect(result.type).to.equal(GET_SPEAKERS_SUCCESS);
   expect(result.speakers).to.have.lengthOf(1);
   expect(result.speakers).to.deep.equal(speakers);
 });
```

To make this test pass, we need to make significant changes to our current implementation of the `getSpeakersSuccess` function.

```
const GET_SPEAKERS_SUCCESS = 'GET_SPEAKERS_SUCCESS';

 function getSpeakersSuccess(speakers) {
   return { type: GET_SPEAKERS_SUCCESS, speakers: speakers };
 }
```

In Redux, actions have an expected format. They must contain a type property and usually contain some data structure. In the case of `getSpeakersSuccess`, our type is a constant, `GET_SPEAKERS_SUCCESS`, and the data is an array of speakers passed into the action. To

make them available to the application, let's move the action and the constant into their own files. We need a `speakerActions` file and an `actionTypes` file,

`src/actions/speakerActions.js`:

```
import * as types from '../reducers/actionTypes';

export function getSpeakersSuccess(speakers) {
  return { type: types.GET_SPEAKERS_SUCCESS, speakers: speakers };
}
```

`src/reducers/actionTypes.js`:

```
export const GET_SPEAKERS_SUCCESS = 'GET_SPEAKERS_SUCCESS';
```

Add import statements to the test and all the tests should pass. For a typical action, this is the format for testing. The placement of the action types in the reducers folder is for dependency inversion reasons. From a SOLID standpoint, the reducers are defining a contract of interaction, which is represented by the action types. The actions are fulfilling that contract.

Testing a thunk

Because the `getSpeakersSuccess` action is intended to be the resulting action of a successful service call, we need a special kind of action to represent the service call itself. Redux does not inherently support asynchronous actions, as stated before. So, we need some other way to accomplish communication with the backend. Thankfully, Redux does support middleware and much middleware has been designed to add asynchronous capability to Redux. We are going to use `redux-thunk` for simplicity.

To start the next test, we need to first import `redux-thunk` and `redux-mock-store` to our speaker action tests.

```
import thunk from 'redux-thunk';
 import configureMockStore from 'redux-mock-store';
```

Then we can test the getting speakers.

```
describe('Async Actions', () => {
   describe('Get Speakers', () => {
     it('exists', () => {
       expect(speakerActions.getSpeakers).to.exist;
     });
   });
 });
```

As usual, we start with a test for existence. And, as usual, it is fairly easy to make this test pass. In the speaker actions file, add a definition for the `getSpeakers` function and export it.

```
export function getSpeakers() {
 }
```

The next test is slightly more complicated than the tests we have been working on, so we will explain it in rather more detail.

The first thing we will need to do is configure a mock store and add the thunk middleware. We need to do this because to properly test a thunk we will have to pretend that Redux is actually running so that we can dispatch our new action and retrieve the results. So, let's add our mock store configuration to the `Async Actions describe`:

```
const middleware = [thunk];
 let mockStore;

 beforeEach(() => {
   mockStore = configureMockStore(middleware);
 });
```

Now that we have a store available to us, we are ready to begin writing the test.

```
it('creates GET_SPEAKERS_SUCCESS when loading speakers', () => {
   // arrange
   const speaker = {
     id: 'test-speaker',
     firstName: 'Test',
```

```
    lastName: 'Speaker'
  };

  const expectedActions = speakerActions.getSpeakersSuccess([speaker]);
  const store = mockStore({
    speakers: []
  });
```

In the arrange, we are configuring a bare minimum speaker. Then, we call the action we previously tested to build the proper data structure. Finally, we define a mock store and its initial state.

```
  // act
  return store.dispatch(speakerActions.getSpeakers()).then(() => {
    const actions = store.getActions();

    // assert
    expect(actions[0].type).to.equal(types.GET_SPEAKERS_SUCCESS);
  });
});
```

Now, when testing asynchronously in Mocha, we can return a promise and Mocha will automatically know that test is asynchronous. Our assertions, for asynchronous tests, go in the resolve or the reject function of the promise. In the case of the get speaker action, we are going to assume a successful server interaction and test the resolved promise.

Because we are not returning anything from our `getSpeakers` action, the `mockStore` throws an error stating that the action may not be an undefined. To move the test forward, we must return something. To move in the direction of using a `thunk`, we need to return a function.

```
export function getSpeakers() {
  return function(dispatch) {
  };
};
```

Adding the return of a function that does nothing else moves the test failure message forward and now presents us with a failure to read the property `then` of undefined. So, now we need to return a promise from our action. We already have the service endpoint built in the mock API service, so let's call that now.

```
export function getSpeakers() {
  return function(dispatch) {
    return new MockSpeakerService().getAll().then(speakers => {
      dispatch(getSpeakersSuccess(speakers))
    }).catch(err => {
      throw(err);
    });
  };
}
```

Now the test passes and we have written our first test dealing with thunks. As you can see, both the test and the code to pass the test are fairly easy to write.

The Get All Speakers reducer

Now that we have tested the actions related to getting all the speakers, it's time to move on to testing the reducers. As usual, let's begin with an exists test.

```
describe('Speaker Reducers', () => {
   describe('Speakers Reducer', () => {
      it('exists', () => {
         expect(speakersReducer).to.exist;
      });
   });
});
```

To make this test pass, all we need to do is define a function named `speakersReducer`.

```
function speakersReducer() {
 }
```

Our next test will check the functionality of the reducer.

```
it('Loads Speakers', () => {
   // arrange
   const initialState = [];

   const speaker = {
     id: 'test-speaker',
     firstName: 'Test',
     lastName: 'Speaker'
   };
   const action = actions.getSpeakersSuccess([speaker]);

   // act
   const newState = speakersReducer(initialState, action);

   // assert
   expect(newState).to.have.lengthOf(1);
   expect(newState[0]).to.deep.equal(speaker);
 });
```

This test is larger than we normally prefer, so let's walk through it. In the arrange, we configure the initial state and create an action result consisting of an array of a single speaker. When a reducer is called, the previous state of the application and the result of an action are passed to it. In this case, we start with an empty array and the modification is the addition of a single speaker.

Next, in the *Act* section of the test, we call the reducer passing in the `initialState` and the result of our action call. The reducer returns a new state for us to use in the application.

Lastly, in the assert, we expect that the new state consists of a single speaker and that the speaker has the same data as the speaker we created for the action.

To make the test pass we need to handle the action being passed into the reducer.

```
function speakersReducer(state = [], action) {
   switch(action.type) {
     case types.GET_SPEAKERS_SUCCESS:
```

```
      return action.speakers;
    default:
      return state;
  }
}
```

Because, in an application using Redux, reducers are called for every action, we need to determine what to do for any action that is not the action we want to handle. The proper response in those cases is to simply return the state with no modification.

For the action type that we do want to handle, in this case we are returning the actions speakers array. In other reducers, we might combine the initial state with the actions result, but for get speakers success we want to replace the state with the value we receive.

The last step, now that all our tests are passing, is to extract the speaker reducer from the test file and move it to `speakerReducer.js`

The Speaker listing component

Another piece of the application that we can test is the components. There are two types of component in a typical React + Redux application. We have container and presentational components.

Container components don't typically hold any real HTML in them. The render function for a container component simply references a single presentational component.

Presentational components don't typically have any business logic in them. They receive properties and display those properties.

In our journey from the back-end to the front-end, we have been covering the retrieval and updating of data. Next, let's look at the container component that will use this data.

Our container component is going to be a simple one. Let's start with the typical existence test.

```
import { expect } from 'chai';
import { SpeakersPage } from './SpeakersPage';

 describe('Speakers Page', () => {
   it('exists', () => {
     expect(SpeakersPage).to.exist;
   });
 });
```

Simple and straightforward; now to make it pass.

```
export class SpeakersPage {
 }
```

Next is the render function of the component.

```
import React from 'react';
import Enzyme, { mount, shallow } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
import { SpeakersPage } from './SpeakersPage';

 describe('Render', () => {
   beforeEach(() => {
      Enzyme.configure({ adapter: new Adapter() });
   });

   it('renders', () => {
     // arrange
     const props = {
       speakers: [{
         id: 'test-speaker',
         firstName:'Test',
         lastName:'Speaker'
       }]
     };

     // act
     const component = shallow(<SpeakersPage { ...props } />);

     // assert
```

```
    expect(component.find('SpeakerList')).to.exist;
    expect(component.find('SpeakerList').props().speakers)
      .to.deep.equal(props.speakers);
  });
});
```

This test introduces some new concepts. Starting at the act portion of the test. We are using Enzyme's shallow render. A shallow render will render the React component but not the component's children. In this case, we are expecting that a `SpeakerList` component exists and that this component is rendering it. The Enzyme adapter configuration is shown here, but it can also be moved to `test.js` after the tests pass.

We are also checking the props to make sure we pass the speakers into the presentational component. To make this test pass, we must make modifications to the `SpeakersPage` component, but we must also create a `SpeakerList` component. Let's do that now.

```
export class SpeakersPage extends Component {
  render() {
    return (
      <SpeakerList speakers={this.props.speakers} />
    );
  }
}
```

And then in a new file, we need to add the `SpeakerList`.

```
export const SpeakerList = ({speakers}) => {}
```

You may have noticed that our container component doesn't have any logic. In fact, all it does is render the `SpeakerList` component. If that is all it does, why is it a container component? The reason is that this component is going to be a Redux-connected component. We want to keep the Redux code in our business logic and out of our display components. So, we are treating this as a higher order component and just using it to pass data through to the presentational components. Later, when we get to the speaker detail component you will see a container component with a little business logic.

For now, our `SpeakerList` component looks a little anemic and doesn't really work as part of a React Redux app. Time to test our presentational components.

```
describe('Speaker List', () => {
  it('exists', () => {
    expect(SpeakerList).to.exist;
  });
});
```

Because of the last test, this test will automatically pass. Normally we would not write this test if we followed to progression what we just did. In reality, what we should have done is ignore the previous test, create this test, and then create the `SpeakerList` component. After which, we could have re-enabled the previous test and gotten it to pass.

The next step is to test that a message of no speakers available is rendered when the speakers array is empty.

```
function setup(props) {
  const componentProps = {
    speakers: props.speakers || []
  };
```

```
      return shallow(<SpeakerList {...componentProps} />);
    }

  describe('On Render', () => {
    describe('No Speakers Exist', () => {
      it('renders no speakers message', () => {
        // arrange
        const speakers = [];

        // arrange
        const component = setup({speakers});

        // assert
        expect(component.find('#no-speakers').text())
          .to.equal('No Speakers Available.');
      });
    });
  });
```

For this test, we created a helper function to initialize the component with the props that we need. To make the test pass we just need to return a `div` with the correct text.

```
export const SpeakerList = ({speakers}) => {
    return (
      <div>
      <h1>Speakers</h1>
      <div id="no-speakers">No Speakers Available.</div>
      </div>
    );
  };
```

While we are only testing for the `no-speakers div`, we can have decoration that we decide not to test. In this case, we want a header on the page. Our tests should pass regardless.

So, now we are ready to test for when speakers do exist.

```
describe('Speakers Exist', () => {
    it('renders a table when speakers exist', () => {
      // arrange
      const speakers = [{
        id: 'test-speaker-1',
        firstName: 'Test',
        lastName: 'Speaker 1'
      }, {
        id: 'test-speaker-2',
        firstName: 'Test',
        lastName: 'Speaker 2'
      }];

      // act
      const component = setup({speakers});

      // assert
      expect(component.find('.speakers')
        .children()).to.have.lengthOf(2);
      expect(component.find('.speakers')
        .childAt(0).type().name).to.equal('SpeakerListRow');
    });
  });
```

In this test, we check for two things. We want the correct number of speaker rows to display and we want them to be rendered by a new `SpeakerListRow` component.

```
export const SpeakerList = ({speakers}) => {
    let contents = <div>Error!</div>;
```

```
      if(speakers.length === 0) {
        contents = <div id="no-speakers">No Speakers Available.</div>;
      } else {
        contents = (
          <table className="table">
            <thead>
              <tr>
                <th>Name</th>
                <th></th>
              </tr>
            </thead>
            <tbody className="speakers">
              {
                speakers.map(speaker =>
                  <SpeakerListRow key={speaker.id} speaker={speaker} />)
              }
            </tbody>
          </table>
        );
      }

      return (
        <div>
        <h1>Speakers</h1>
        { contents }
        </div>
      );
    };
```

The component code has changed significantly because of our latest test. We had to add some logic, and we also added a default error case if somehow the content were to make it through without being assigned.

There is one more component to make the code work correctly for this section. We are not going to test that component in this book, though. The component has no logic inside it and is left as an exercise to you to create.

In order to create that component, it would be nice if the application ran. Right now, we have not wired up Redux so the application won't render anything. Let's walk through the configuration we are using for Redux now.

Inside `index.js`, we need to add a few items to let Redux work. Your index should look similar to this:

```
import React from 'react';
import ReactDOM from 'react-dom';
import {BrowserRouter} from 'react-router-dom';
import {Provider} from 'react-redux';
import registerServiceWorker from './registerServiceWorker';
import configureStore from './store/configureStore';
import { getSpeakers } from './actions/speakerActions';
import 'bootstrap/dist/css/bootstrap.min.css';
import './index.css';
import App from './components/App.js';

const store = configureStore();
store.dispatch(getSpeakers());

ReactDOM.render(
  <Provider store={store}>
    <BrowserRouter>
      <App/>
    </BrowserRouter>
  </Provider>,
  document.getElementById('root')
);
```

```
registerServiceWorker();
```

The two parts that we have added are the Redux store including an initial call to dispatch the load speakers action, and markup to add the Redux provider.

Where your other routes are defined, you will need to add routes for the speaker section. We are placing the Routes in App.js.

```
<Route exact path='/speakers/:id' component={SpeakerDetailPage}/>
<Route exact path='/speakers' component={SpeakersPage}/>
```

Lastly, we have to convert our component to a Redux component. Add the following lines to the bottom of your speaker's page component file.

```
import { connect } from 'react-redux';

function mapStateToProps(state) {
   return {
     speakers: state.speakers
   };
 }

 function mapDispatchToProps(dispatch) {
   return bindActionCreators(speakerActions, dispatch);
 }

 export default connect(mapStateToProps, mapDispatchToProps)(SpeakersPage);
```

Starting at the bottom of the code sample, the connect function is provided by Redux and will wire up all the Redux functionality into our component. The two functions passed in, mapStateToProps and mapDispatchToProps, are passed in as a way to populate state and provide actions for our component to execute.

Inside mapDispatchToProps we are calling bindActionCreators; this is another Redux-provided function and will give us an object containing all the actions. By returning that object directly from mapDispatchToProps, we are adding the actions directly to props. We could also create our own object containing an actions property and then assign the result of the bindActionCreators to that property.

Anywhere inside the application that references SpeakersPage can now be changed to just SpeakersPage, which will grab our new default export. Do not make this change in the tests. Inside the tests we still want the named import.

With those things done, we should be able to run the application and navigate to the speakers route. If you have not added a link to the speakers route, now would be a good time so that you don't have to type the route directly in the URL every time.

Once you arrive at the speakers route, you should see that there are no speakers and we receive our message. We need some way to populate the speakers so that we can test the listing. We will cover a way to populate speakers in the next section. For now, in the mock API modify the constructor to contain a couple of speakers. Modifying the service in this way will cause a few tests to break, so after you have visually verified that everything is looking good, be sure to remove or at least comment out the code you added.

Speaker detail

Now that we have our speakers listing nicely, it would be nice to be able to view a bit more information about a specific speaker. Let's look at the tests involved in retrieving and viewing a speakers-detailed information.

Adding to the mock API Service

In the mock API, we need to add a call to get the details for a specific speaker. We can assume that the speaker has an ID field that we can use to gather that information. As usual, let's start our tests with a simple exists check. We will need to add a new describe inside the `After Initialization` describe for getting a speaker by ID.

```
describe('Get Speaker By Id', () => {
  it('exists', () => {
    // assert
    expect(service.getById).to.exist;
  });
});
```

To make this test pass, we need to add a method to the mock API.

```
getById() {
}
```

Now, we can write a test to verify the functionality we expect when a matching speaker cannot be found. The functionality we want in this case is for a `SPEAKER_NOT_FOUND` message to be shown once we get to the user interface. At the mock API level, we could assume that a 404 will be sent from the server. We can respond from the mock API with an error containing the `SPEAKER_NOT_FOUND` type. This is similar to the way an action would be used.

Let's create another describe for our speaker not found scenario.

```
describe('Speaker Does Not Exist', () => {
  it('SPEAKER_NOT_FOUND error is generated', () => {
    // act
    const promise = service.getById('fake-speaker');

    // assert
    return promise.catch((error) => {
      expect(error.type).to.equal(errorTypes.SPEAKER_NOT_FOUND);
    });
  });
});
```

You may have noticed that we snuck in `errorTypes`. The `errorTypes` are in their own folder, but build exactly like `actionTypes`.

To make this test pass, we must add a rejected promise to our mock API.

```
getById(id) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
        reject({ type: errorTypes.SPEAKER_NOT_FOUND });
    }, 0);
  });
}
```

We don't have any tests that enforce a positive result from this method, so we can reject every time for now.

That brings us to our next test. What happens if the speaker is found? Ideally, the speaker and all the speakers details would be delivered back to the caller. Let's write that test now.

```
describe('Speaker Exists', () => {
   it('returns the speaker', () => {
      // arrange
      const speaker = { id: 'test-speaker' };
      service.create(speaker);

      // act
      let promise = service.getById('test-speaker');

      // assert
      return promise.then((speaker) => {
        expect(speaker).to.not.be.null;
        expect(speaker.id).to.equal('test-speaker');
      });
   });
});
```

To pass this test we will have to add some logic to the production code.

```
getById(id) {
   return new Promise((resolve, reject) => {
      setTimeout(() => {
        let speaker = this._speakers.find(x => x.id === id);
        if(speaker) {
          resolve(Object.assign({}, speaker));
        } else {
          reject({ type: errorTypes.SPEAKER_NOT_FOUND });
        }
      }, 0);
   });
}
```

To make the test pass, we need to first check to see if the speaker exists. If the speaker does exist, we return that speaker. If the speaker does not exist, we reject the promise and provide our error result.

The Get Speaker action

We now have a mock API to call that behaves the way we want it to. Next on our list is creating the actions that will handle the results from our mock API. For the process of getting a speaker, we will need two actions. One of the actions will notify the application about a successful find and provide the found speaker to the reducers. The other action will notify the application about the failure to find the requested speaker.

Let's write a test to confirm its existence. This test should be inside the synchronous tests section of the speaker actions tests. We will also want to create a new describe for the get speaker success action.

```
describe('Find Speaker Success', () => {
  it('exists', () => {
    expect(speakerActions.getSpeakerSuccess).to.exist;
  });
});
```

To make this test pass, we just create the action function.

```
export function getSpeakerSuccess() {
}
```

Now we need to verify the return value of the action. Just like our get all speakers success action, the get speaker success action will receive the found speaker and return an object containing a type and the speaker data. Let's write the test for that now.

```
it('is created with correct data', () => {
  // arrange
  const speaker = {
    id: 'test-speaker',
    firstName: 'Test',
    lastName: 'Speaker'
  };

  // act
  const result = speakerActions.getSpeakerSuccess(speaker);

  // assert
  expect(result.type).to.equal(types.GET_SPEAKER_SUCCESS);
  expect(result.speaker).to.deep.equal(speaker);
});
```

This test is fairly straightforward so let's look at the production code to pass it.

```
export function getSpeakerSuccess(speaker) {
  return { type: types.GET_SPEAKER_SUCCESS, speaker: speaker };
}
```

Again, this code is straightforward. Next, let's handle the failure action. We will need to create a new describe for this test as well.

```
describe('Get Speaker Failure', () => {
  it('exists', () => {
    expect(speakerActions.getSpeakerFailure).to.exist;
```

```
    });
  });
```

Nothing new here, you should be starting to get a feel for the flow by now. Let's keep going and make this test pass.

```
export function getSpeakerFailure() {
  }
```

The data we should be getting back for a failure to retrieve a speaker should be the SPEAKER_NOT_FOUND error type. In our next test, we will receive that error and create the action type from it.

```
it('is created with correct data', () => {
  // arrange
  const error = {
    type: errorTypes.SPEAKER_NOT_FOUND
  };

  // act
  const result = speakerActions.getSpeakerFailure(error);

  // assert
  expect(result.type).to.equal(types.GET_SPEAKER_FAILURE);
  expect(result.error).to.deep.equal(error);
});
```

Making this test pass is very similar to the implementation for the other synchronous actions.

```
export function getSpeakerFailure(error) {
    return { type: types.GET_SPEAKER_FAILURE, error: error }
  }
```

Looking at the code, there is one important difference. This code doesn't have speaker data. The reason is because this action will need to be handled by a different reducer, an error reducer. We will create the error reducer and error component shortly. But first, we need to create the asynchronous action that will make the call to the mock API.

In testing the asynchronous action to get speakers, we should start with the failure case. In this case, the failure case is GET_SPEAKER_FAILURE. Here is a test to ensure the correct secondary action is triggered.

```
it('creates GET_SPEAKER_FAILURE when speaker is not found', () => {
  // arrange
  const speakerId= 'not-found-speaker';
  const store = mockStore({
    speaker: {}
  });

  // act
  return (
    store.dispatch(speakerActions.getSpeaker(speakerId)).then(() => {
      const actions = store.getActions();

      // assert
      console.log(actions);
      expect(actions[0].type).to.equal(types.GET_SPEAKER_FAILURE);
    })
  );
});
```

The code to make this test pass is similar to the code we have for getting all the speakers.

```
export function getSpeaker(speakerId) {
   return function(dispatch) {
      return new MockSpeakerService().getById(speakerId).catch(err => {
         dispatch(getSpeakerFailure(err));
      });
   };
}
```

Here, we have called the mock API and we expect it to reject the promise, resulting in the dispatching of the `getSpeakerFailure` action.

Our next test is the successful retrieval of a specific speaker. We do have a problem though. You may have noticed that we are creating a new `MockSpeakerService` for each asynchronous action. This is problematic because it prevents us from pre-populating our mock API with values for the test. Later in the development of this application, the back-end will be ready and we will want to point our front-end code to a real back-end. We can't do that as long as we are directly referencing and creating a mock API service.

There are many solutions for the problem that we are facing. We will explore making a factory to decide what back-end to provide for us. A factory will also allow us to treat the mock API as a singleton. Treating the service as a singleton will allow us to prepopulate the service as part of the test setup.

In the services folder, let's create a new set of tests for creating the factory class and functionality.

```
import { expect } from 'chai';
import { ServiceFactory as factory } from './serviceFactory';

describe('Service Factory', () => {
  it('exits', () => {
    expect(factory).to.exist;
  });
});
```

All we need to make this test pass is a class definition.

```
export class ServiceFactory {
}
```

Now we want a method to create a speaker service. Add a new describe to the factory tests.

```
describe('Create Speaker Service', () => {
   it('exists', () => {
      expect(factory.createSpeakerService).to.exist;
   });
});
```

Notice the way we are using the factory, we are not initializing it. We want the factory to be a class with static methods. Having static functions will give us the singleton ability we want.

```
static createSpeakerService() {
}
```

Next up, we want to ensure that the `createSpeakerService` factory method will provide us with an instance of the mock API.

```
it('returns a speaker service', () => {
   // act
```

```
    let result = factory.createSpeakerService();

    // assert
    expect(result).to.be.an.instanceof(MockSpeakerService);
  });
```

Making this test pass is easy, just return a new mock speaker service from the factory method.

```
static createSpeakerService() {
  return new MockSpeakerService();
}
```

This isn't a singleton though. So, we still have some more work to do here. Let's write one more test in the factory before we swap out all the service calls in the application for factory calls. To verify that something is a singleton, we have to make sure it is the same throughout the application. We can do that by doing reference comparisons on successive calls. Another option is to create the speaker service, add a speaker to it, create a new speaker service, and try to pull the speaker from the second service. If we have done things correctly, the second option is the most thorough. We will do the first option here, but it would be a good exercise to do the second option on your own.

```
it('returns the same speaker service', () => {
  // act
  let service1 = factory.createSpeakerService();
  let service2 = factory.createSpeakerService();

  // assert
  expect(service1).to.equal(service2);
});
```

To pass the test, we must ensure that the same instance of the speaker service is returned every time.

```
export default class ServiceFactory {
  constructor() {
    this._speakerService = null;
  }

  static createSpeakerService() {
    return this._speakerService = this._speakerService ||
                                  new MockSpeakerService();
  }
}
```

The factory will now return the current value or create a new speaker service if the current value is `null`.

The next step is to go to each place where we are directly instantiating a mock speaker service and swap it out with a factory call. We will leave that as an exercise for you to do, but know that going forward we will assume that it has been done.

Now that we have the factory swapped out and it is generating a singleton, we can write the next action test. We want to test a successful retrieval of a speaker.

```
it('creates GET_SPEAKER_SUCCESS when speaker is found', () => {
  // arrange
  const speaker = {
    id: 'test-speaker',
    firstName: 'Test',
    lastName: 'Speaker'
  };
```

```
    const store = mockStore({ speaker: {} });
    const expectedActions = [
      speakerActions.getSpeakerSuccess([speaker.id])
    ];
    let service = factory.createSpeakerService();
    service.create(speaker);

    // act
    return store.dispatch(
      speakerActions.getSpeaker('test-speaker')).then(() => {
        const actions = store.getActions();

        // assert
        expect(actions[0].type).to.equal(types.GET_SPEAKER_SUCCESS);
        expect(actions[0].speaker.id).to.equal('test-speaker');
        expect(actions[0].speaker.firstName).to.equal('Test');
        expect(actions[0].speaker.lastName).to.equal('Speaker');
      });
  });
```

There is a lot going on in this test; let's walk through it. First in the arrange, we create a speaker object to be placed in the service, and used for the assertions. Next, still in the arrange, we create and configure the mock store. Lastly, in the arrange, we create the speaker service and we create our test speaker using the service.

Next, in the act, we dispatch a call to get the test speaker. Remember, this call is asynchronous. So, we must subscribe to then.

When the promise is resolved, we store the actions in a variable and assert that the first action has the correct type and payload.

Now to make this test pass we need to make some modifications to the `getById` method on the service.

```
export function getSpeaker(speakerId) {
   return function(dispatch) {
     return factory.createSpeakerService().getById(speakerId).then(
       speaker => {
         dispatch(getSpeakerSuccess(speaker));
       }).catch(err => {
         dispatch(getSpeakerFailure(err));
       });
   };
}
```

All we have really done here is add a then to handle the resolving of the promise. We now have, for all current intents and purposes, a working speaker service. Let's move on to creating the reducers for handling the get speaker actions.

The Get Speaker reducer

To handle the actions related to getting a speaker, we must create two reducers. The first reducer is extremely similar to the reducer we made for the get speakers actions. The second is going to need to be slightly different and is for handling the error case.

Let's begin with the simplest of the two and create the speaker reducer.

```
describe('Speaker Reducer', () => {
    it('exists', () => {
      expect(speakerReducer).to.exist;
    });
 });
```

Our typical existence test is easily passed.

```
export function speakerReducer() {
  }
```

The next test ensures that the reducer updates state properly, and will close out the tests needed for this reducer.

```
it('gets a speaker', () => {
   // arrange
   const initialState = { id: '', firstName: '', lastName: '' };
   const speaker = { id: 'test-speaker', firstName: 'Test', lastName: 'Speaker'};
   const action = actions.getSpeakerSuccess(speaker);

   // act
   const newState = speakerReducer(initialState, action);

   // assert
   expect(newState).to.deep.equal(speaker);
 });
```

The changes from this test are the inputs to the reducer, and the output of a state. Let's make this test pass by modeling our reducer after the speakers reducer.

```
export function speakerReducer(state = {
   id: '',
   firstName: '',
   lastName: ''
}, action) {
   switch(action.type) {
     case types.GET_SPEAKER_SUCCESS:
       return action.speaker;
     default:
       return state;
   }
 }
```

Similar to the speakers reducer, this reducer simply checks the action type for GET_SPEAKER_SUCCESS and, if found, returns the speaker attached to the action as the new state. Otherwise, we just return the state object we received.

Next up, we need an error reducer.

```
describe('Error Reducer', () => {
  it('exists', () => {
    expect(errorReducer).to.exist;
  });
});
```

Passing this test is just as easy as all the other existence tests.

```
import * as types from './actionTypes';
import * as errors from './errorTypes';

export function errorReducer() {
}
```

The error reducer will have some interesting functionality. In the event that an error is received, we want multiple errors to stack up so we won't be replacing the state. Instead, we will be cloning and adding to the state. However, when an action is received that is not an error we will want to clear the errors and allow normal program execution to continue. We will also want to ignore duplicate errors. First, we will handle the error we know about.

```
it('returns error state', () => {
  // arrange
  const initialState = [];
  const error = { type: errorTypes.SPEAKER_NOT_FOUND };
  const action = actions.getSpeakerFailure(error);

  // act
  const newState = errorReducer(initialState, action);

  // assert
  expect(newState).to.deep.equal([error]);
});
```

Our test is slightly different from the previous reducer test. The main difference is that we are wrapping our expected value in an array. We are doing this to meet the need for having multiple errors potentially stack up and display for the user.

To make this test pass we follow the familiar reducer pattern we have been using.

```
export function errorReducer(state = [], action) {
  switch(action.type) {
    case types.GET_SPEAKER_FAILURE:
      return [...state, action.error];
    default:
      return state;
  }
}
```

For the same reasons as stated previously, notice how we use the rest of the parameter syntax to spread the existing state into a new array, effectively cloning state.

We have two more tests for the error reducer; the first is to ensure duplicate errors are not added. The second test will be to clear the errors when a non-error action is called.

```
it('ignores duplicate errors', () => {
  // arrange
  const error = { type: errorTypes.SPEAKER_NOT_FOUND };
  const initialState = [error];
  const action = actions.getSpeakerFailure(error);

  // act
  const newState = errorReducer(initialState, action);
```

```
    // assert
    expect(newState).to.deep.equal([error]);
  });
```

In the test, to set the condition of having a prepopulated state, all we had to do was modify the `initialState` parameter.

```
export function errorReducer(state = [], action) {
  switch(action.type) {
    case types.GET_SPEAKER_FAILURE:
      let newState = [...state];

      if(newState.every(x => x.type !== action.error.type)) {
        newState.push(action.error);
      }

      return newState;
    default:
      return state;
  }
}
```

All we must do to make this test pass is make sure that the error type is not already present in the state array. There are many ways to do this; we have chosen to use the `every` function as a check that none of the existing errors match. It is likely that this method is not extremely performant, but there should only be a couple errors at most so it shouldn't be a performance issue.

The next test is to clear the error state when a non-error is received.

```
it('clears when a non-error action is received', () => {
  // arrange
  const error = { type: errorTypes.SPEAKER_NOT_FOUND };
  const initialState = [error];
  const action = { type: 'ANY_NON_ERROR' };

  // act
  const newState = errorReducer(initialState, action);

  // assert
  expect(newState).to.deep.equal([]);
});
```

Making this test pass is exceedingly simple. All we have to do is replace the default functionality where the existing state is returned.

```
default:
  return [];
```

The Speaker Detail component

We are now ready to create our `SpeakerDetailPage`. There isn't much to this component. It will need to be another container component so that it can use the get speaker action. Because it is a container component, we will not be placing any markup directly into this component. The good news for us is that it means our tests will be short and simple.

To get the tests started, create an existence test.

```
describe('Speaker Detail Page', () => {
   it('exists', () => {
      expect(SpeakerDetailPage).to.exist;
   });
 });
```

Create a `SpeakerDetailPage` file and add a component to it.

```
export class SpeakerDetailPage extends Component {
   render() {
      return (<div></div>);
   }
 }
```

The next thing we want to test, the only other thing we can test without directly specifying the design, is that the model is received and somehow makes it to the screen. We only need to test one property of the model for now. We will write a test that shows that the first name of the speaker is displayed.

```
describe('Render', () => {
   it('renders', () => {
     // arrange
     const props = {
       match: { params: { id: 'test-speaker' } },
       actions: { getSpeaker: (id) => { return Promise.resolve(); } },
       speaker: { firstName: 'Test' }
     };

     // act
     const component = mount(<SpeakerDetailPage { ...props } />);

     // assert
     expect(component.find('first-name').text()).to.contain('Test');
   });
 });
```

If you are paying attention, you might have wondered why the get speaker action is just returning an empty resolved promise. We are not attached to Redux, so kicking off the action doesn't trigger a reducer, which doesn't update the store and doesn't trigger a refresh of the component state. We still want to complete the contract of the component in the test setup though and this component will call that function. We could leave this line out, but we will be adding it back as soon as we wire up Redux.

To make the test pass, we will need to make a couple of simple changes in the `SpeakerDetailPage` component, and create a whole new component. Following are the changes to this component,

but it will be an exercise for you to create the next component. It is only for display and we are testing that it gets populated here so all you have to do is write the presentational component.

```
export class SpeakerDetailPage extends Component {
  constructor(state, context) {
    super(state, context);

    this.state = {
      speaker: Object.assign({}, this.props.speaker)
    };
  }

  render() {
    return (
      <SpeakerDetail speaker={this.state.speaker} />
    );
  }
}
```

The previous code will make the test pass, but now we have to connect the component to Redux. We will be adding a call to the getSpeaker action, binding to the componentWillReceiveProps life cycle event, and mapping props and dispatch using the connect function. Here is the final SpeakerDetailPage component.

```
export class SpeakerDetailPage extends Component {
  constructor(state, context) {
    super(state, context);

    this.state = {
      speaker: Object.assign({}, this.props.speaker)
    };

    this.props.actions.getSpeaker(this.props.match.params.id)
  }

  componentWillReceiveProps(nextProps) {
    if(this.props.speaker.id !== nextProps.speaker.id) {
      this.setState({ speaker: Object.assign({}, nextProps.speaker) });
    }
  }

  render() {
    return (
      <SpeakerDetail speaker={this.state.speaker} />
    );
  }
}

function mapStateToProps(state, ownProps) {
  let speaker = { id: '', firstName: '', lastName: '' }

  return {
    speaker: state.speaker || speaker
  };
}

function mapDispatchToProps(dispatch) {
  return {
    actions: bindActionCreators(speakerActions, dispatch)
  }
}

export default  connect(
  mapStateToProps,
  mapDispatchToProps
)(SpeakerDetailPage);
```

Now that everything passes the tests, we have one last thing we need to do before we can

properly develop further. Earlier we replaced the mock API with a call to a factory. We did this so that the tests could affect the state of the mock API in the actions. That same modification has made it possible to configure a starting point for our application. In the `index.js` file, add the following code after the store has been configured; now, when you run the app, you will have speakers available to test the UI with.

```
const speakers = [{
    id: 'clayton-hunt',
    firstName: 'Clayton',
    lastName: 'Hunt'
}, {
    id: 'john-callaway',
    firstName: 'John',
    lastName: 'Callaway'
}];

let service = factory.createSpeakerService();
speakers.forEach((speaker) => {
    service.create(speaker);
});
```

Summary

That does it for unit-testing a React application, for now. We still don't have an example of testing some kind of input. Try to test and implement a `CreateSpeakerPage`. What would you need to do from a React standpoint? What would Redux cause you to do? In this chapter, we have attacked the React components as if they were components. For display-only components, which is what these have been, this approach is probably better. However, for a component with some real functionality you might want to try testing the functionality as a plain old JavaScript class before even attaching it to React. We also left quite a bit of work for you to do in this chapter. Don't be shy about looking at the source related to this chapter if you get lost or need a hint while you are filling in the blanks to complete the code.

Exploring Integrations

In this chapter, we'll explore integration-testing the Speaker Meet application. The React front-end application will be tested and configured to hit the real back-end API, and the .NET application will be tested to ensure that it functions properly from controller to database.

In this chapter, we cover:

- Implementing a real API service
- Removing mocked API calls
- End-to-end integration
- Integration tests

Implementing a real API service

The time has come to actually receive data from the server. Our current data model is still not 100% correct, but the groundwork is there. When we receive the correct data structure from the server, we will need to update our views accordingly. We will leave that part as an exercise for you.

In this section, we will look at pulling our mocked API out of the factory that we created and replacing it with a real API. In our existing tests, we will use Sinon to override the default functionality of our Ajax component with the functionality from our mock API.

Lastly, we will need to create an application configuration object to manage the base path for the API to determine the correct path in both dev and prod.

Replacing the mock API with the real API service

To keep things as simple as possible, we will be using the fetch API to get data from the server. We will begin by breaking all the tests that are currently using the mock API. That is because we are going to create a stub class that implements the same interface as the mock API, but it will not be doing anything:

src/services/fetchSpeakerService.js

```
import * as errorTypes from '../reducers/errorTypes';

export default class FetchSpeakerService {
  constructor() { }

  create(speaker) {
    return;
  }

  getAll() {
    return;
  }

  getById(id) {
    return;
  }
}
```

Now, replace the mock service that is created by the factory with the creation of the fetch based service:

```
import FetchSpeakerService from './fetchSpeakerService';

export default class ServiceFactory {
  constructor() {
    this._speakerService = null;
  }

  static createSpeakerService() {
    return this._speakerService =
      this._speakerService || new FetchSpeakerService();
  }
}
```

Thankfully, only four tests are failing because of that change. Looking at the failed tests, three of them are failing because we did not return a promise. One test, however, is failing because we are no longer returning the mock API. We are going to ignore the failing tests caused by missing promises by excluding them temporarily. Then, we will focus on test checking for a specific instance.

The test that is failing is in the service factory tests. We don't actually want the service factory to return a `MockSpeakerService`. We want it to return a `FetchSpeakerService`. Even more accurately, we want any implementation of a `SpeakerService`. Let's create a base class that will behave like an

interface or abstract class from C#:

/src/services/speakerService.js

```
export default class SpeakerService {
   create(speaker) {
     throw new Error("Not Implemented!")
   }

   getAll() {
     throw new Error("Not Implemented!")
   }

   getById(id) {
     throw new Error("Not Implemented!")
   }
 }
```

Now we have an abstract base class, we need to inherit from that base class in both our existing service classes:

```
import SpeakerService from './speakerService';

 export default class MockSpeakerService extends SpeakerService {
   constructor() {
     super();

     this._speakers = [];
  }
 …
```

```
import SpeakerService from './speakerService';

 export default class FetchSpeakerService extends SpeakerService {
   constructor() {
     super();
   }
 …
```

And then we need to modify the factory tests to expect an instance of the base class instead of the derived class:

```
it('returns a speaker service', () => {
   // act
   let result = factory.createSpeakerService();

   // assert
   expect(result).to.be.an.instanceOf(SpeakerService);
 });
```

Using Sinon to mock Ajax responses

Now, it is time to tackle the three tests that we have ignored. They are expecting actual responses from our service. Right now, our service is completely empty. Keep in mind, those tests were written to be unit tests and we need to protect them from the changes in the response that the real endpoint will experience over time. For that reason, we are going to, finally, introduce Sinon.

We will use Sinon to return the results from our mock API instead of the real API. This will allow us to continue to use the work we have already put into the mock API.

After we have our existing tests covered, we are going to introduce integration tests by using Sinon to mock the back-end server. Using Sinon in that way will allow us to test-drive our fetch based speaker service.

Fixing existing tests

First things first; we must make our existing tests pass. In the `speakerActions.spec.js` file, find the first test that we skipped and remove the skip. This will cause that test to fail with:

```
Cannot read property 'then' of undefined
```

Back in the `beforeEach` method, where we are creating the speaker service, we need to create a new Sinon stub for a service method. Looking at the test, we can see that the first service call we make is to get all speakers. So, let's start there:

```
beforeEach(() => {
    let service = factory.createSpeakerService();
    let mockService = new MockSpeakerService();

    getAll = sinon.stub(service, "getAll");
    getAll.callsFake(mockService.getAll.bind(mockService));

    mockStore = configureMockStore(middleware);
});
```

Looking at this code, what we have done is to create a new Sinon stub and redirect calls to the service `getAll` method to the `mockService getAll` method. Lastly, we bind the `mockService` call to the `mockService` to preserve access to private variables in the `mockService`.

Running the tests again, we get a new error:

```
Attempted to wrap getAll which is already wrapped
```

What this error is telling us is that we have already created a stub for the method we are trying to stub. At first, this error may not make any sense. But, if you look we are doing this in a `beforeEach`. Sinon is a singleton and we are running our mocking commands inside a `beforeEach`, so it already has a `getAll` stub registered by the time the second test is preparing to run. What we must do is remove that registration before we try to register it again. Another way to say this is that we must remove the registration after each test run. Let's add an `afterEach` method and remove the registration there:

```
afterEach(() => {
    getAll.restore();
});
```

That fixes the first failing test that we had, now to fix the other two. The process will be largely the same, so let's get started.

Remove the skip from the next test. The test fails. We are calling the `getSpeaker` action in this test and if we look at the speaker actions, we can see that it uses the `getById` service method. As before we will need to stub this method in the `beforeEach`.

```
getById = sinon.stub(service, "getById");
getById.callsFake(mockService.getById.bind(mockService));
```

As before, we are now getting the already wrapped message:

```
Attempted to wrap getById which is already wrapped
```

We can fix this one the same way we fixed the last one, by removing the stub in the `afterEach` function.

```
getById.restore();
```

We are back to all passing tests with one skipped. The last test is the exact same process. Here are the full `beforeEach` and `afterEach` functions when we are done:

```
beforeEach(() => {
    let service = factory.createSpeakerService();
    let mockService = new MockSpeakerService();

    getAll = sinon.stub(service, "getAll");
    getAll.callsFake(mockService.getAll.bind(mockService));

    getById = sinon.stub(service, "getById");
    getById.callsFake(mockService.getById.bind(mockService));

    create = sinon.stub(service, "create");
    create.callsFake(mockService.create.bind(mockService));

    mockStore = configureMockStore(middleware);
});

afterEach(() => {
  create.restore();
  getAll.restore();
  getById.restore();
});
```

Don't forget to remove the skip from the last test. When all is said and done you should have 42 passing tests and 0 skipped tests.

Mocking the server

Now that we have fixed our existing tests, we are ready to start writing tests for our real service, the `fetchSpeakerService`. Let's get started by looking at the test we used for our mock service. The tests will largely be the same as we are trying to achieve the same pattern of functionality.

First, we will want to create the test file `fetchSpeakerService.spec.js.` Once the file is created, we can add the standard existence test:

```
describe('Fetch Speaker Service', () => {
   it('exits', () => {
     expect(FetchSpeakerService).to.exist;
   });
 });
```

Because we stubbed out the fetch speaker service earlier, this test should just pass after we add the appropriate import.

Following the mock speaker service tests, the next test is a construction and type verification test:

```
it('can be constructed', () => {
   // arrange
   let service = new FetchSpeakerService();

   // assert
   expect(service).to.be.an.instanceof(FetchSpeakerService);
 });
```

This test, too, should pass right away, because when we stubbed the fetch service we created it as a class. Continuing to follow the progression of the mock service tests, we have an `After Initialization` section with a `Create` section inside it. The only test in the *Create* section is an exists test for the `Create` method. Writing this test, it should pass:

```
describe('After Initialization', () => {
   let service = null;

   beforeEach(() => {
     service = new FetchSpeakerService();
   });

   describe('Create', () => {
     it('exists', () => {
       expect(service.create).to.exist;
     });
   });
 });
```

Because we are copying the flow from the mock service tests, we have already extracted the service to a `beforeEach` instantiation.

In the next section, our tests will start to get interesting and won't just pass right away. Before we move on, to verify that the tests are doing what they should be doing, it is a good idea to comment out parts of the fetch service and see the appropriate tests pass.

Moving on to the `Get All` section, still inside the `After Initialization` section, we have an existence test checking the `getAll` method:

```
describe('Get All', () => {
   it('exists', () => {
      // assert
      expect(service.getAll).to.exist;
   });
});
```

As with the other tests so far, to fail this test you will have to comment out the `getAll` method in the fetch service to see it fail. Immediately following this test are two more sections: `No Speakers Exist` and `Speaker Listing`. We will add them one at a time starting with `No Speakers Exist`:

```
describe.skip('No Speakers Exist', () => {
   it('returns an empty array', () => {
      // act
      let promise = service.getAll();

      // assert
      return promise.then((result) => {
        expect(result).to.have.lengthOf(0);
      });
   });
});
```

Finally, we have a failing test. The failure is complaining because it doesn't look like we returned a promise. Let's begin the proper implementation of the fetch service and we will use Sinon in the tests to mock the back-end. In the fetch service, add the following:

```
constructor(baseUrl) {
   super();

   this.baseUrl = baseUrl;
}

getAll() {
   return fetch(`${this.baseUrl}/speakers`).then(r => {
      return r.json();
   });
}
```

This is a very basic fetch call. We are use the HTTP verb, `GET`, so there is no reason to call a method on fetch; by default it will use `GET`.

In our tests, we are now getting a meaningful result. `fetch is not defined`. This result is because fetch does not exist as part of our testing setup yet. We will need to import a new NPM package to handle fetch calls in testing. The package we want to import is `fetch-ponyfill`.

```
>npm install fetch-ponyfill
```

After installing the `ponyfill` library, we must modify our test setup file `scripts/test.js`:

```
import { JSDOM } from'jsdom';
import Enzyme from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
import fetchPonyfill from 'fetch-ponyfill';
const { fetch } = fetchPonyfill();

const jsdom = new JSDOM('<!doctype html><html><body></body></html>');
const { window } = jsdom;
window.fetch = window.fetch || fetch;
```

```
…
global.window = window;
global.document = window.document;
global.fetch = window.fetch;
```

After those modifications, we must restart our tests for the changes to take effect. We are now getting a test failure telling us that only absolute URLs are supported. We are getting this message because when we instantiate our fetch service we aren't passing a baseURL. For the tests it doesn't matter what the URL is so let's just use localhost:

```
beforeEach(() => {
    service = new FetchSpeakerService('http://localhost');
});
```

After making this change we have moved the error forward and now we are getting a fetch error to the effect that localhost refused a connection. We are now ready to replace the back-end with Sinon. We will start in the `beforeEach` and `afterEach`:

```
let fetch = null;

beforeEach(() => {
  fetch = sinon.stub(global, 'fetch');
  service = new FetchSpeakerService('http://localhost');
});

afterEach(() => {
  fetch.restore();
});
```

In the test, we will need some items from the `fetch-ponyfill` package so let's add the import statements while we are close to the top of the file.

```
import fetchPonyfill from 'fetch-ponyfill';
const {
  Response,
  Headers
} = fetchPonyfill();
```

And now in the test, we need to configure the response from the server:

```
it('returns an empty array', () => {
    // arrange
    fetch.returns(new Promise((resolve, reject) => {
      let response = new Response();
      response.headers = new Headers({
        'Content-Type': 'application/json'
      });
      response.ok = true;
      response.status = 200;
      response.statusText = 'OK';
      response.body = JSON.stringify([]);

      resolve(response);
    }));

    // act
    let promise = service.getAll();

    // assert
    return promise.then((result) => {
      expect(result).to.have.lengthOf(0);
    });
});
```

That finishes the `No Speakers Exist` scenario. We will refactor the server response once we have a better idea about what data will be changing.

We are now ready for the speaker listing scenario. As before, we start by copying the test from the mock service tests. Remove the arrange from the mock service test and copy the arrange from our previous test.

After adding the arrange from the no speakers test, we get a message expecting a length of 1 instead of 0. This is an easy fix and for the purposes of this test we can simply add an empty object to the body array of the response. Here is what the test should look like, once it is passing:

```
describe('Speaker Listing', () => {
    it('returns speakers', () => {
      // arrange
      fetch.returns(new Promise((resolve, reject) => {
        let response = new Response();
        response.headers = new Headers({
          'Content-Type': 'application/json'
        });
        response.ok = true;
        response.status = 200;
        response.statusText = 'OK';
        response.body = JSON.stringify([{}]);

        resolve(response);
      }));

      // act
      let promise = service.getAll();

      // assert
      return promise.then((result) => {
        expect(result).to.have.lengthOf(1);
      });
    });
  });
});
```

Now that we are using basically the same arrange twice, it's time to refactor our tests. The only thing that has really changed is the body. Let's extract an `okResponse` function to use:

```
function okResponse(body) {
    return new Promise((resolve, reject) => {
      let response = new Response();
      response.headers = new Headers({
        'Content-Type': 'application/json'
      });
      response.ok = true;
      response.status = 200;
      response.statusText = 'OK';
      response.body = JSON.stringify(body);

      resolve(response);
    });
  }
```

We have placed this helper function at the top of the `After Initialization describe`. Now in each test, replace the arrange with a call to the function, passing in the body that is specific to that test.

The get all speakers functionality is now covered by the tests. Let's move on to getting a specific speaker by ID. Copy the tests for `getById` from the mock service tests and apply a skip to the

describes. Now, remove the skip from the outer-most describe. This should enable the existence test, which should pass.

The next test is for when a speaker is not found; removing skip from that test results in a message indicating that we are not returning a promise. Let's go into the body of the `getById` function and use fetch to get a speaker:

```
getById(id) {
   return fetch(`${this.baseUrl}/speakers/${id}`);
 }
```

Adding fetch to our function should have fixed the error but hasn't. Remember we are mocking the response from fetch so if we don't set a response then fetch won't return anything at all. Let's configure the mock response. In this case we are expecting a 404 from the server so let's configure that response:

```
// arrange
 fetch.returns(new Promise((resolve, reject) => {
   let response = new Response();
   response.headers = new Headers({
     'Content-Type': 'application/json'
   });
   response.ok = false;
   response.status = 404;
   response.statusText = 'NOT FOUND';

   resolve(response);
 }));
```

That makes our test pass, but it's not for the right reason. Let's add a `then` clause to the assertion to prove the false positive:

```
// assert
 return promise}).then(() => {
   throw { type: 'Error not returned' };
 }).catch((error) => {
   expect(error.type).to.equal(errorTypes.SPEAKER_NOT_FOUND);
 });
```

Now our test will fail with expected `'Error not returned'` to equal `'SPEAKER_NOT_FOUND'`. Why is this? Shouldn't a 404 cause a rejection of the promise? With fetch, the only thing that will cause a rejected promise is a network connection error. For that reason, we didn't reject when we mocked the server response. What we need to do is check for that condition in the service and cause a promise rejection on that side. The easiest way to accomplish this is to wrap the fetch call with a promise of our own. Once wrapped, we can check for the appropriate condition and reject our promise:

```
getById(id) {
   return new Promise((resolve, reject) => {
     fetch(`${this.baseUrl}/speakers/${id}`).then((response) => {
       if (!response.ok) {
         reject({
           type: errorTypes.SPEAKER_NOT_FOUND
         });
       }
     });
   });
 }
```

That should do it for this test. We are now ready for our last test. Before we move on, let's do a

quick refactoring of the arrange in this test to shorten the test and have it make a bit more sense to future readers. While we are doing that, we will refactor the existing response function to reduce duplication and enforce some default values:

```
function baseResponse() {
   let response = new Response();
   response.headers = new Headers({
     'Content-Type': 'application/json'
   });
   response.ok = true;
   response.status = 200;
   response.statusText = 'OK';

   return response;
}

function okResponse(body) {
   return new Promise((resolve, reject) => {
     let response = baseResponse();
     response.body = JSON.stringify(body);

     resolve(response);
   });
}

function notFoundResponse() {
   return new Promise((resolve, reject) => {
     let response = baseResponse();
     response.ok = false;
     response.status = 404;
     response.statusText = 'NOT FOUND';

     resolve(response);
   })
}
```

Use the `notFoundResponse` function in the test just like we used the `okResponse` function. Moving on to our last test for the current functionality of the fetch service, remove the skip from the next describe and we will begin looking at the errors generated and make the necessary changes to make the test pass.

This last test is fairly simple after the work we have already done to make mock responses easier. We need the fetch call to return an `ok` response with the speaker as the body:

```
describe('Speaker Exists', () => {
   it('returns the speaker', () => {
     // arrange
     const speaker = {
       id: 'test-speaker'
     };
     fetch.returns(okResponse(speaker));

     // act
     let promise = service.getById('test-speaker');

     // assert
     return promise.then((speaker) => {
       expect(speaker).to.not.be.null;
       expect(speaker.id).to.equal('test-speaker');
     });
   });
});
```

Now, we are getting a timeout error. That is because our service isn't actually handling the case where the speaker exists. Let's add that now:

```
getById(id) {
    return new Promise((resolve, reject) => {
      fetch(`${this.baseUrl}/speakers/${id}`).then((response) => {
        if (response.ok) {
          resolve(response.json());
        } else {
          reject({
            type: errorTypes.SPEAKER_NOT_FOUND
          });
        }
      });
    });
  }
```

Now all our tests are passing and we have verified all the expected behavior of the system. There are a few more things we could do and some developers will choose to do them. We will discuss some of them but will not be providing examples.

Application configuration

Now that all the tests are passing there is still some application configuration that must be taken care of before the application can be used.

In the service factory, we must set a base URL for the fetch service to use when the application is running. This can be done many different ways and which way exactly is up to you. The simplest but least flexible way is to just hard-code a string value as the base URL used to construct the service. However, you could get as fancy as having a dynamic class that sets the value based on the applications, running environment. Again this decision is left to you.

End-to-end integration tests

The last subject we will discuss in this chapter is end-to-end integration tests. These tests involve actually calling the server and checking the real responses.

Benefits

So, what are the benefits from testing the actual client server connection? The most valuable benefit is that you know your application will work in the deployed environment. Sometimes an application will get deployed and not work because a network or database connection was incorrectly configured and that will wreak havoc on a deployment.

Additionally, this will help to verify the system is working properly. A series of smoke tests could be employed after a deployment to ensure the deployment was successful.

Detriments

E2E tests are usually skipped for one of two reasons. The first reason is that they are difficult to write. You have a lot of extra setup to get these tests to run, including a completely different test runner than what you normally use for unit testing. If not a different runner, they at least need to be a separate test run and not included in your normal unit tests.

The second issue is that E2E tests are fragile. Any change to the system and these tests break. They are not commonly run all the time like a unit test is and so the broken code will not be noticed until they are run in the production environment.

For these reasons we generally do not write that many E2E tests, if any at all.

How much end-to-end testing should you do?

If you choose to do end-to-end testing, you will want to do as little as possible. These are the top tier of tests and should be the least numerous type of test in your system. A recommendation is to only write as many tests as you have third-party connections to your application, that is, one test for each back-end server that you must communicate with. Additionally, use the simplest and most basic case which is not anticipated to change.

That completes integration testing from the front-end. There are still some things that can be done. We will leave them as an exercise for you. You might have noticed that the front-end and back-end are not fully in agreement for the model that is being passed back and forth. As an exercise, add or remove and refine the model that is being used by both systems so that they agree on the format.

Another task would be to set the base URL for the fetch service and run both applications locally to verify interconnectability.

Configuring the API project

With the React project now configured to hit the real API, it's time to turn our attention to the .NET solution. In order to verify that everything is wired up correctly, you'll want to write a series of integration tests to ensure that the whole system is working properly.

Integration test project

Create a new xUnit Project called `SpeakerMeet.Api.IntegrationTest` within the existing solution. This will be where the .NET integration tests will be created. You may want to explore separating these out according to your preferences and/or team coding standards, but that can wait. For now, a single integration test project will do.

For our purposes, we'll be testing whether the system functions from API entry all the way to the database, and back. However, it's best to start small test individual integration points, and grow from there.

Where to begin?

You could certainly start by creating a test that will call an API endpoint. In order to achieve this, an HTTP Request will need to be made to a controller. The controller will then call into a service within the business layer, which in turn will make a call to the repository, and finally a command is sent to the database. That feels like a lot of moving parts. Perhaps there's a better place to start.

In order to break down the problem into smaller, more manageable pieces, perhaps it's best to start testing closer to the persistence layer of the application.

Verifying the repository calls into the DB context

A good place to start is verifying that the system is fully integrated; let's first test that the repository can access the database. Create a folder within the integration test project called `RepositoryTests` and create a new test file called `GetAll`. This will be where the integration tests for the `GetAll` method of the repository will be created.

You could create a test that verifies that the repository can be created, like so:

```
[Fact]
public void ItExists()
{
  var options = new DbContextOptions<SpeakerMeetContext>();
  var context = new SpeakerMeetContext(options);

  var repository = new Repository.Repository<Speaker>(context);
}
```

However, that's not going to pass. If you run the test you will receive the following error:

```
System.InvalidOperationException: No database provider has been configured for this DbContext.
```

This is easily fixed by configuring an appropriate provider.

InMemory database

Running tests against a SQL Server is time-consuming, error-prone, and potentially costly. Establishing a connection to a database takes time, and remember, you want your test suite to be lightning-fast. It might also be a problem to rely on data if the database is used by others, whether in a development environment, by quality assurance engineers, and so on. You certainly wouldn't want to run your integration tests against a production database. Additionally, running tests against a database hosted in the cloud (for example, AWS, Azure, and so on) could potentially incur a dollar cost in terms of bandwidth and processing.

Luckily, it's quite trivial to configure a solution that uses Entity Framework to use an `InMemory` database.

First, install a `NuGet` package for the `InMemory` database.

```
Microsoft.EntityFrameworkCore.InMemory
```

Now, modify the test you created before so that the database context is created `InMemory`:

```
[Fact]
public void ItExists()
{
  var options = new DbContextOptionsBuilder<SpeakerMeetContext>()
      .UseInMemoryDatabase("SpeakerMeetInMemory")
      .Options;

  var context = new SpeakerMeetContext(options);

  var repository = new Repository.Repository<Speaker>(context);
}
```

The test should now pass because the context is now being created `InMemory`.

Next, create a test to verify that a collection of Speaker entities is returned when the `GetAll` method is called:

```
[Fact]
public void GivenSpeakersThenQueryableSpeakersReturned()
{
  using (var context = new SpeakerMeetContext(_options))
  {
    // Arrange
    var repository = new Repository.Repository<Speaker>(context);

    // Act
    var speakers = repository.GetAll();

    // Assert
    Assert.NotNull(speakers);
    Assert.IsAssignableFrom<IQueryable<Speaker>>(speakers);
  }
}
```

Now, turn your attention to the `Get` method in the repository. Create a new test method to verify that a null Speaker entity is returned when a speaker with the given ID is not found:

```
[Fact]
public void GivenSpeakerNotFoundThenSpeakerNull()
{
  using (var context = new SpeakerMeetContext(_options))
  {
    // Arrange
    var repository = new Repository.Repository<Speaker>(context);

    // Act
    var speaker = repository.Get(-1);

    // Assert
    Assert.Null(speaker);
  }
}
```

This should pass right away. Now, create a test to verify that a Speaker entity is returned when a speaker with the supplied ID exists:

```
[Fact]
public void GivenSpeakerFoundThenSpeakerReturned()
{
  using (var context = new SpeakerMeetContext(_options))
  {
    // Arrange
    var repository = new Repository.Repository<Speaker>(context);

    // Act
    var speaker = repository.Get(1);

    // Assert
    Assert.NotNull(speaker);
    Assert.IsAssignableFrom<Speaker>(speaker);
  }
}
```

This test will not pass quite yet. Regardless of whether or not a Speaker with the ID of 1 exists in your development database, the speakers table in the InMemory database is currently empty. Adding data to the InMemory database is quite simple.

Adding speakers to the InMemory database

In order to test that the repository will return specific Speaker entities when querying the database, you first must add Speakers to the database. In order to do this, add a few lines of code to your test file:

```
using (var context = new SpeakerMeetContext(_options))
{
  context.Speakers.Add(new Speaker { Id = 1, Name = "Test"... });
  context.SaveChanges();
}
```

Feel free to add as many speakers as you want, with as much detail as you feel necessary. Your test should now pass. More tests can be created, and should continue to be added as the system grows in functionality and complexity. The bulk of the logic should be tested already in the unit tests, but verifying that the system functions as a whole is equally important.

Verify that the service calls the DB through the repository

Moving on to the business layer, you should verify that each service can retrieve data from the `InMemory` database through the repository.

First, create a new folder in the integration test project called `ServiceTests`. Within that folder, create a folder named `SpeakerServiceTests`. This folder is where the tests specific to the `SpeakerService` will be created.

Create a new test file named `GetAll`. Add a test method to verify that the service can be created:

```
[Fact]
public void ItExists()
{
  var options = new DbContextOptionsBuilder<SpeakerMeetContext>()
      .UseInMemoryDatabase("SpeakerMeetInMemory")
      .Options;

  var context = new SpeakerMeetContext(options);

  var repository = new Repository<Speaker>(context);
  var gravatarService = new GravatarService();

  var speakerService = new SpeakerService(repository, gravatarService);
}
```

ContextFixture

There's a lot of setup code here and quite a bit of duplication from our previous tests. Luckily, you can use what's known as a *Test Fixture*.

A Test Fixture is simply some code that is run to configure the system under test. For our purposes, create a *ContextFixture* to set up an `InMemory` database.

Create a new class named `ContextFixture`, which is where all the `InMemory` database creation will happen:

```
public class ContextFixture : IDisposable
{
  public SpeakerMeetContext Context { get; }

  public ContextFixture()
  {
    var options = new DbContextOptionsBuilder<SpeakerMeetContext>()
        .UseInMemoryDatabase("SpeakerMeetContext")
        .Options;

    Context = new SpeakerMeetContext(options);

    if (!Context.Speakers.Any())
    {
      Context.Speakers.Add(new Speaker {Id = 1, Name = "Test"...});
      Context.SaveChanges();
    }
  }

  public void Dispose()
  {
    Context.Dispose();
  }
}
```

Now, modify the test class to use the new `ContextFixture` class:

```
[Collection("Service")]
[Trait("Category", "Integration")]
public class GetAll : IClassFixture<ContextFixture>
{
  private readonly IRepository<Speaker> _repository;
  private readonly IGravatarService _gravatarService;

  public GetAll(ContextFixture fixture)
  {
    _repository = new Repository<Speaker>(fixture.Context);
    _gravatarService = new GravatarService();
  }

  [Fact]
  public void ItExists()
  {
    var speakerService = new SpeakerService(_repository, _gravatarService);
  }
}
```

That's quite a bit cleaner. Now, create a new test to ensure a collection of `SpeakerSummary` objects is returned when the `GetAll` method of the `SpeakerService` is called:

```
[Fact]
public void ItReturnsCollectionOfSpeakerSummary()
{
  // Arrange
  var speakerService = new SpeakerService(_repository, _gravatarService);

  // Act
  var speakers = speakerService.GetAll();

  // Assert
  Assert.NotNull(speakers);
  Assert.IsAssignableFrom<IEnumerable<SpeakerSummary>>(speakers);
}
```

Next, create a new test class for the `Get` method of the `SpeakerService`. The first test should validate that an exception is thrown when a speaker does not exist with the supplied ID:

```
[Fact]
public void GivenSpeakerNotFoundThenSpeakerNotFoundException()
{
  // Arrange
  var speakerService = new SpeakerService(_repository, _gravatarService);

  // Act
  var exception = Record.Exception(() => speakerService.Get(-1));

  // Assert
  Assert.IsAssignableFrom<SpeakerNotFoundException>(exception);
}
```

You can reuse the `ContextFixture` that you created earlier:

```
[Fact]
public void GivenSpeakerFoundThenSpeakerDetailReturned()
{
  // Arrange
  var speakerService = new SpeakerService(_repository, _gravatarService);

  // Act
  var speaker = speakerService.Get(1);

  // Assert
  Assert.NotNull(speaker);
  Assert.IsAssignableFrom<SpeakerDetail>(speaker);
}
```

Verify the API calls into the service

Now, turn your attention to the web API controllers. As covered in a previous chapter, you could simply create a new instance of the controller and call the method under test. However, that would not exercise the entire system.

It would be far better to call the method with an HTTP request. Deploying to a web server would be prohibitively time-consuming.

TestServer

ASP.NET Core has the ability to configure a host for testing purposes. Install the `TestServer` from `NuGet`:

```
Microsoft.AspNetCore.TestHost
```

There's a little setup involved. First, you'll add an instance of the `TestServer`. Create a new `WebHostBuilder` and use the existing `Startup` class of the web API project. This will wire up the Dependency Injection container that was set up previously. Now, configure the services to set up a new `InMemory` database.

Take a look at the test here to see the setup required:

```csharp
[Fact]
public async void ItShouldCallGetSpeakers()
{
  // Arrange
  var server = new TestServer(new WebHostBuilder()
      .UseStartup<Startup>()
      .ConfigureServices(services =>
      {
        services.AddDbContext<SpeakerMeetContext>(o =>
          o.UseInMemoryDatabase("SpeakerMeetInMemory"));
      }));

  var client = server.CreateClient();

  // Act
  var response = await client.GetAsync("/api/speaker");

  // Assert
  Assert.NotNull(response);
}
```

ServerFixture

In order to move the setup out of the controller tests, again use a test fixture. This time, create a new class named ServerFixture. This will be where the setup will live for the controller tests:

```
public class ServerFixture : IDisposable
{
  public TestServer Server { get; }
  public HttpClient Client { get; }

  public ServerFixture()
  {
    Server = new TestServer(new WebHostBuilder()
              .UseStartup<Startup>()
              .ConfigureServices(services =>
              {
                services.AddDbContext<SpeakerMeetContext>(o =>
                  o.UseInMemoryDatabase("SpeakerMeetContext"));
              }));

    if (Server.Host.Services.GetService(typeof(SpeakerMeetContext)) is SpeakerMeetContext cont
    {
      context.Speakers.Add(new Speaker {Id = 1, Name = "Test"...});
      context.SaveChanges();
    }

    Client = Server.CreateClient();
  }

  public void Dispose()
  {

    Server.Dispose();
    Client.Dispose();
  }
}
```

Now, return to the previous test. Modify the test class to use the ServerFixture:

```
[Collection("Controllers")]
[Trait("Category", "Integration")]
public class GetAll : IClassFixture<ServerFixture>
{
  private readonly HttpClient _client;

  public GetAll(ServerFixture fixture)
  {
    _client = fixture.Client;
  }

  [Fact]
  public async void ItShouldCallGetSpeakers()
  {
    // Act
    var response = await _client.GetAsync("/api/speaker");

    Assert.NotNull(response);
  }
}
```

Now, verify that the response returns an OK status code by creating a new test:

```
[Fact]
public async void ItShouldReturnSuccess()
{
  // Act
  var response = await _client.GetAsync("/api/speaker/");
  response.EnsureSuccessStatusCode();

  // Assert
  Assert.Equal(HttpStatusCode.OK, response.StatusCode);
}
```

And finally, ensure that the proper speaker is returned:

```
[Fact]
public async void ItShouldReturnSpeakers()
{
  // Act
  var response = await _client.GetAsync("/api/speaker");
  response.EnsureSuccessStatusCode();

  var responseString = await response.Content.ReadAsStringAsync();
  var speakers = JsonConvert.DeserializeObject<List<SpeakerSummary>>(responseString);

  // Assert
  Assert.Equal(1, speakers[0].Id);
}
```

Remember, you want to make sure your test suite is clean and well maintained. To clean this test up a bit, you might want to consider creating a ReadAsJsonAsync extension. Here's what that might look like:

```
public static class Extensions
{
  public static async Task<T> ReadAsJsonAsync<T>(this HttpContent content)
  {
    var json = await content.ReadAsStringAsync();

    return JsonConvert.DeserializeObject<T>(json);
  }
}
```

And now, modify the test to use the new extension method:

```
[Fact]
public async void ItShouldReturnSpeakers()
{
  // Act
  var response = await _client.GetAsync("/api/speaker");
  response.EnsureSuccessStatusCode();

  var speakers = await response.Content.ReadAsJsonAsync<List<SpeakerSummary>>();

  // Assert
  Assert.Equal(1, speakers[0].Id);
}
```

That's much better. Now this extension can be used and reused over and over, and its first use has now been documented in the ItShouldReturnSpeakers test.

Now, move on to testing that the single speaker endpoint can be called. Create a test named ItShouldCallGetSpeaker and ensure that a response is returned:

```
[Fact]
public async void ItShouldCallGetSpeaker()
{
  // Act
```

```
   var response = await _client.GetAsync("/api/speaker/-1");

   Assert.NotNull(response);
}
```

Now, test that the proper response code is returned if a Speaker with the given ID does not exist:

```
[Fact]
public async void ItShouldReturnError()
{
   // Act
   var response = await _client.GetAsync("/api/speaker/-1");

   // Assert
   Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}
```

Now validate that an OK status code is returned when a speaker with the supplied ID exists:

```
[Fact]
public async void ItShouldReturnSuccess()
{
   // Act
   var response = await _client.GetAsync("/api/speaker/1");
   response.EnsureSuccessStatusCode();

   // Assert
   Assert.Equal(HttpStatusCode.OK, response.StatusCode);
}
```

And finally, confirm that the speaker returned is the one that is expected. Note that the ReadAsJsonAsync can be used here:

```
[Fact]
public async void ItShouldReturnSpeaker()
{
   // Act
   var response = await _client.GetAsync("/api/speaker/1");
   response.EnsureSuccessStatusCode();

   var speakerSummary = await response.Content.ReadAsJsonAsync<SpeakerDetail>();

   // Assert
   Assert.Equal(1, speakerSummary.Id);
}
```

Only the Get and GetAll methods for speakers have been tested in the preceding pages. Feel free to add tests for the Search methods to grow your integration test suite.

Summary

You should now have a firm grasp of integration testing, its benefits, and detriments. The mock API calls have been removed and the real API service has been implemented. Integration tests have been created and now ensure separate parts of the application are working well together.

Change is inevitable, especially in software development. In the next chapter, we'll be discussing how to handle a change in requirements. Whether these changes include new features, resolve defects, or change existing logic, through TDD these can be easily managed.

Changes in Requirements

As progress is made on any application, new and different requirements will likely be added. Sometimes these requirements enhance the existing functionality of the application. At other times, these new requirements may conflict with the existing functionalities. When requirements conflict, it's important that issues are resolved so that the proper functionality can be built.

So, what are the changes in requirements you might expect to see? Changes often consist of alterations to a business rule, new features or enhancements, or modifications needed to resolve a bug or defect discovered in the system.

As time goes on, there will often be a need to modify an existing business rule. This may be in response to user feedback, clarification from the business, or a need discovered through use of the system. When the need for change is discovered, then the existing application will need to change. A comprehensive test suite will ensure that the rest of the system still operates as expected once the new changes are implemented. Start by modifying and/or creating new tests to cover the new desired functionality of the system.

There's a common saying in software development that *software is never finished; it is merely abandoned*. That is to say that an application will continue to grow and evolve through new development if it is to continue to be useful. If new features aren't being added, then it is likely that the project has simply been abandoned. If an application is to continue to be of use, then you can expect that new features will need to be implemented. Again, start with the tests and add new tests which will help guide your implementation of any and all new features.

When a bug is discovered and the root cause identified, then a change will need to be made to resolve the issue. In order to prevent this bug from appearing again in the future, a new test, or series of tests, should be written to cover any potential scenarios that would result in the erroneous behavior.

In this chapter, we will gain an understanding of:

- Changing requirements
- A new feature
- Dealing with defects
- Changes to Speaker Meet
- Premature optimization

Hello World

Stepping back to one of our first examples, take a look at the sample *Hello World* application. Remember that, depending on the time of day, a different message is displayed to the user. Before noon, the user is greeted with Good morning, and after noon, Good afternoon is returned to the user.

A change in requirements

Depending on the time of day, the user is greeted with Good morning or Good afternoon. To extend the functionality and introduce a new feature, let's address the user with Good evening if the time of day is between 6 p.m. and midnight.

Good evening

In order to introduce this new feature, begin with the tests. Modification of an existing test will be needed, as well as adding one or more new tests to cover the change in requirements.

Modify the `Theory` data provided to `GivenAfternoon_ThenAfternoonMessage` so that only noon through 6 p.m. are included for this test. Now, create a new test method, `GivenEvening_ThenEveningMessage`:

```
[Theory]
[InlineData(19)]
[InlineData(20)]
[InlineData(21)]
[InlineData(22)]
[InlineData(23)]
public void GivenEvening_ThenEveningMessage(int hour)
{
  // Arrange
  var eveningTime = new TestTimeManager();
  eveningTime.SetDateTime(new DateTime(2017, 7, 13, hour, 0, 0));
  var messageUtility = new MessageUtility(eveningTime);

  // Act
  var message = messageUtility.GetMessage();

  // Assert
  Assert.Equal("Good evening", message);
}
```

Now make the `Theory` pass by modifying the existing code:

```
public string GetMessage()
{
  if (_timeManager.Now.Hour < 12)
    return "Good morning";
  if (_timeManager.Now.Hour <= 18)
    return "Good afternoon";
  return "Good evening";
}
```

This is a fairly simple example, for sure. The implementation is starting to grow a design with which you may or may not be satisfied. Feel free to experiment with alternative implementations. You should now have sufficient tests that you feel safe to refactor to a pattern with which you're happier. If you break the implementation or discover a bug you may have introduced, add a test for this scenario.

FizzBuzz

Moving on to the FizzBuzz example from Chapter 2, *Setting Up the .NET Test Environment*, extend the classic behavior of this code kata and introduce some new behavior.

A new feature

A new requirement has been added to the classic FizzBuzz kata. The new requirement states that when a number is not divisible by 3 or 5, and is greater than 1, then the message Number not found should be returned. This should be easy enough. Start, once again, with the tests, and make the necessary modifications.

Number not found

To get started, a new test method is needed to verify that the Number not found message is returned:

```
[Fact]
public void GivenNonDivisibleGreaterThan1ThenNumberNotFound()
{
  // Arrange
  // Act
  var result = FizzBuzz(2);
  // Assert
  Assert.Equal("Number not found", result);
}
```

Now, make the test pass by modifying the existing code:

```
private object FizzBuzz(int value)
{
  if (value % 15 == 0)
    return "FizzBuzz";
  if (value % 5 == 0)
    return "Buzz";
  if (value % 3 == 0)
    return "Fizz";
  if (value == 2)
    return "Number not found";
}
```

This covers the first instance. However, does this satisfy the new requirement? Create a `Theory` set to force the proper solution:

```
[Theory]
[InlineData(2)]
[InlineData(4)]
[InlineData(7)]
[InlineData(8)]
public void GivenNonDivisibleGreaterThan1ThenNumberNotFound(int number)
{
  // Arrange
  // Act
  var result = FizzBuzz(number);
  // Assert
  Assert.Equal("Number not found", result);
}
```

Make the test pass, the right way. Modify the existing code so that the desired results are returned:

```
private object FizzBuzz(int value)
{
  if (value % 15 == 0)
    return "FizzBuzz";
  if (value % 5 == 0)
    return "Buzz";
  if (value % 3 == 0)
    return "Fizz";
  return value == 1 ? (object)value : "Number not found";
}
```

Note that all the existing tests should continue to pass throughout this exercise. If you find a bug, write a new test to verify the scenario, and correct the code accordingly.

TODO app

The *TODO* app was another one of our early TDD examples. This app is far from complete, and we have received new requirements from the business, asking to add a feature to the application.

The business now wants the ability to complete a task in the TODO list. This feature is *schedule current sprint* and is the next story for us to work on.

Mark complete

For the *Mark complete* story, we have been asked to allow the user to complete any of the tasks in the TODO list. Adding this feature should be much like any other TDD exercise in this book. Before reading our solution to this problem, try to complete this one on your own. After you have passing tests, come back and look at the solution in this book.

Adding tests

In the `ToDoApplicationTests` file, we have added a `yak shaving` test to force us to create the complete method. This test also helps to define the API for the method:

```
[Fact(Skip = "Yak shaving - no longer needed")]
public void CompleteTodoExists()
{
  // Arrange
  var todo = new TodoList();
  var item = new Todo();

  todo.AddTodo(item);

  // Act
  todo.Complete(item);
}
```

This causes us to create a method stub in the `TodoList` class. To get this test to pass, we had to remove the not implemented exception from the generated method. After creating the method, we added a skip to this test, similar to the previous `yak shaving` test in the same file.

Next, we needed to create a `TodoListCompleteTests` file to house the functionality tests for the complete method:

```
public class TodoListCompleteTests
{
  [Fact]
  public void ItRemovesAnItemFromTheList()
  {
    // Arrange
    var todo = new TodoList();
    var item = new Todo();

    todo.AddTodo(item);
    // Act
    todo.Complete(item);
    // Assert
    Assert.Equal(0, todo.Items.Count());
  }
}
```

After writing this first test and implementing the code to make it pass, we were hard pressed to write another test that would fail. So, we assume that we are done for now.

Production code

The code to make the tests for completing a task is quite simple and only requires a single line method:

```
public void Complete(Todo item)
{
  _items.Remove(item);
}
```

That is all we need. We are now ready for the sprint demo.

But don't remove from the list!

During the sprint demo, our product owner asked what happened to the task when it was completed. We explained that it was removed from this list. This was not good. The product owner was hoping that we could provide metrics on tasks further down the road. She would like for us to track the completion of the task instead of deleting it.

After some discussion with the other developers, we have decided the task will gain a completed attribute and be hidden from the list. To accomplish this, we will have to do a bit of refactoring and add new tests. Again, try to complete this exercise on your own and then look at our solution for comparison.

Adding tests

This change required quite a few new tests. Before we could make new tests, though, we had to first rename our existing completion test to represent the correct functionality. Adding two more tests to the `TodoListCompleteTests` file, we verify both that the item is marked complete and that it is not removed from the TODO list:

```
public class TodoListCompleteTests
{
  [Fact]
  public void ItHidesAnItemFromTheList()
  {
    // Arrange
    var todo = new TodoList();
    var item = new Todo { Description = "Test Todo" };

    todo.AddTodo(item);

    // Act
    todo.Complete(item);

    // Assert
    Assert.Equal(0, todo.Items.Count());
  }

  [Fact]
  public void ItMarksAnItemComplete()
  {
    // Arrange
    var todo = new TodoList();
    var item = new Todo { Description = "Test Todo" };

    todo.AddTodo(item);

    // Act
    todo.Complete(item);

    // Assert
    Assert.True(item.IsComplete);
  }

  [Fact]
  public void ItShowsCompletedItems()
  {
    // Arrange
    var todo = new TodoList();
    var item = new Todo { Description = "Test Todo" };

    todo.ShowCompleted = true;
    todo.AddTodo(item);

    // Act
    todo.Complete(item);

    // Assert
    Assert.Equal(1, todo.Items.Count());
  }
}
```

In order to add `ShowComplete`, we created a `yak shaving` test in the `ToDoApplicationTests` file for completeness:

```
[Fact(Skip = "Yak shaving - no longer needed")]
public void ShowCompletedExists()
{
  // Arrange
  var todo = new TodoList();

  // Act
  todo.ShowCompleted = true;
}
```

We also had to add a similar test to the `TodoModelTests` file:

```
[Fact]
public void ItHasIsComplete()
{
  // Arrange
  var todo = new Todo();

  // Act
  todo.IsComplete = true;
}
```

Production code

For such a small code base, the changes required by the new tests caused a fairly significant change. First, we added an `IsComplete` property to the `Todo` model:

```
internal class Todo
{
  public bool IsComplete { get; set; }
  public string Description { get; set; }

  internal void Validate()
  {
    Description = Description ?? throw new DescriptionRequiredException();
  }
}
```

The rest of the changes affect the `TodoList` class. A boolean property was added to toggle the visibility of completed items, the `Complete` method was modified to only mark the item as complete, and a `where` clause was added to the items retrieved from the list:

```
internal class TodoList
{
  private readonly List<Todo> _items = new List<Todo>();

  public IEnumerable<Todo> Items => _items.Where(t => !t.IsComplete || ShowCompleted);

  public bool ShowCompleted { get; set; }

  public void AddTodo(Todo item)
  {
    item = item ?? throw new ArgumentNullException();
    item.Validate();
    _items.Add(item);
  }

  public void Complete(Todo item)
  {
    item.IsComplete = true;
  }
}
```

Changes to Speaker Meet

Change is inevitable with any application. Requirements change as a result of a new business rule, feature enhancement, discovery and remediation of a defect, and so on. Change is especially certain when test driving an application. Luckily, through the process of TDD, your application should be easily and safely modifiable.

If a system is loosely coupled, then changes to one part of a system should, in theory, have little to no impact on the rest of the system. A comprehensive suite of unit tests should alleviate the fear of making changes.

Unfortunately, the tests are only valid for the scenarios which they define. If sufficient tests aren't written to cover certain scenarios or edge cases, then it is certainly possible that a bug could find its way into production. If the TDD approach is not taken, or worse, tests aren't written at all, then you may discover that it is quite easy for a bug to make it through all of the checks of your code review process and CI/CD build pipeline.

Take a look at the new requirements for the Speaker Meet application.

Changes to the back-end

As the Speaker Meet application progressed, a new requirement was introduced. Speakers had to be *approved* before they were visible in parts of the system. This included the full listing of speakers, returning of speaker detail information, and through search results.

In this scenario, a developer came in to help out with the implementation. This developer was not familiar with TDD and did not write tests to validate his work. The new requirement was implemented and a code review was submitted:

```
public Models.SpeakerDetail Get(int id)
{
  var speaker = _repository.Get(id);

  if (speaker == null || speaker.IsDeleted || speaker.IsActive)
  {
    throw new SpeakerNotFoundException(id);
  }

  var gravatar = _gravatarService.GetGravatar(speaker.EmailAddress);

  return new Models.SpeakerDetail
  {
    Id = speaker.Id,
    Name = speaker.Name,
    Location = speaker.Location,
    Gravatar = gravatar
  };
}
```

And a change to the class was added:

```
public class Speaker
{
  public int Id { get; set; }

  [Required]
  [StringLength(50)]
  public string Name { get; set; }

  [Required]
  [StringLength(50)]
  public string Location { get; set; }

  [Required]
  [StringLength(255)]
  public string EmailAddress { get; set; }

  public bool IsDeleted { get; set; }

  public bool IsActive { get; set; }
}
```

Can you spot the issue?

The code was reviewed and comments left. However, the comments were misunderstood (or just flatly ignored) and the code was committed, merged, and pushed through the deployment process. A breakdown for sure, but one that happens from time to time.

The CI server ran the test suite. The existing tests passed. The bug was not discovered, as there was no existing scenario that would have caught the error. Since new tests were not created, there was no test failure. The CD process ran and the code made it into production.

So what test can be added to ensure the proper code is implemented? When dealing with bugs, it is often best to simply write the test that verifies the incorrect behavior. In this case, we want an error to be thrown. So, the below test should assert that the correct error is thrown:

```
[Fact]
public void GivenSpeakerIsNotActiveThenSpeakerNotFoundException()
{
  // Arrange
  var expectedSpeaker = SpeakerFactory.Create(_fakeRepository);
  expectedSpeaker.IsActive = false;
  var service = new SpeakerService(_fakeRepository, _fakeGravatarService);
  // Act
  var exception = Record.Exception(() => service.Get(expectedSpeaker.Id));
  // Assert
  Assert.IsAssignableFrom<SpeakerNotFoundException>(exception);
}
```

Make this new test pass by modifying the service:

```
public Models.SpeakerDetail Get(int id)
{
  var speaker = _repository.Get(id);

  if (speaker == null || speaker.IsDeleted || !speaker.IsActive)
  {
    throw new SpeakerNotFoundException(id);
  }

  var gravatar = _gravatarService.GetGravatar(speaker.EmailAddress);

  return new Models.SpeakerDetail
  {
    Id = speaker.Id,
    Name = speaker.Name,
    Location = speaker.Location,
    Gravatar = gravatar
  };
}
```

However, with this change, a number of existing tests will now break. This is because the default value for the IsActive property is false.

To quickly get these tests to pass, you could do something like:

```
public bool IsActive { get; set; } = true;
```

This could potentially introduce unexpected results, so be sure to create some guard tests to verify correctness.

This explains why this bug wasn't initially caught. The IsActive property was added to the database with a default value of true. The bug wasn't discovered until new speakers were added to the database with a value of false in the IsActive column. Once the incorrect behavior was discovered, the defect was easily identified and remedied.

Changes to the front-end

There is no difference, from a concept or approach perspective, for changes to the front-end. You will need to write the appropriate test to ensure the desired behavior from the application and then write the production code to make the test pass.

As a quick example though, let's add a new feature to the front-end code we have been working on.

Sorted by rating on client side

The feature we are going to add is sorting the speakers by rating. In previous chapters, rating was not discussed or even enforced, so modifications will need to happen to include rating in the model that has been built so far. That is, of course, if you have not already completed the full model as defined by the C# code.

As with earlier examples in this chapter, try to add this behavior yourself and then look at our following solution .

In the `speakerReducer.spec.js` file, we have added a single test for default sorting of speakers by rank. The test can be added to the describe block for the speaker reducer:

```
it('sorts speakers by rank', () => {
   // Arrange
   const initialState = [];
   const speaker1 = { id: 'test-speaker-1', firstName: 'Test 1', lastName: 'Speaker', rank: 1]
   const speaker2 = { id: 'test-speaker-2', firstName: 'Test 2', lastName: 'Speaker', rank: 2]
   const action = actions.getSpeakersSuccess([speaker1, speaker2]);

   // Act
   const newState = speakersReducer(initialState, action);

   // Assert
   expect(newState).to.have.lengthOf(2);
   expect(newState[0]).to.deep.equal(speaker2);
 });
```

And the code to make this test pass is in the `speakerReducer.js` file:

```
export function speakersReducer(state = [], action) {
   switch(action.type) {
     case types.GET_SPEAKERS_SUCCESS:
       return action.speakers.sort((a, b) => {
         return b.rank > a.rank;
       });
     default:
       return state;
   }
 }
```

What now?

Moving forward, it should be easy to implement any change necessary. This might include a new feature, a change in requirements, or a discovered defect. That isn't to say that the application is complete or error-free, but you should have some level of confidence that the application behaves in the ways accounted for with the existing test suite.

Premature optimization

For the purpose of clarification, we are defining optimization as anything that obfuscates the code, making it less clear or more difficult to understand, or anything that limits the possibilities further than the test requires. A premature optimization is an optimization that is done for any reason other than specified by a requirement.

Typically, optimizations are done using performance as an excuse. Before these types of modifications of the code are done, a requirement specifying the need for the change should exist.

Even through the practice of Test-Driven Development, it is possible to paint yourself into a corner. Often during refactoring or during the process of designing your next test, it is possible to solve too much of the problem at once or refactor too much.

Always keep in mind that, in TDD, we want to break a problem down into the smallest steps possible. Also, don't go for the solution in the first test if the solution is more than a line or two. At the same time, even for small solutions, if the solution is calculation or algorithm heavy, it should still be broken down, even if the eventual solution is a single line of production code.

Beware of premature optimizations.

Refactoring, according to Kent Beck, is the process of removing duplication. Remember that while refactoring your tests. By only removing duplication, we can avoid premature optimization via refactoring. It is completely possible, and even attractive at times, to refactor a solution and reduce the code significantly, or even to use a fancy new language feature or Linq expression to make your test pass. These solutions are fine in the long run, but while the tests are still being built, these hidden optimizations can cause you and your tests to become derailed extremely quickly.

Summary

You can now see how a change in requirements, a new feature request, or a defect might require an application to change. Through TDD and a comprehensive suite of unit tests, these changes can be made safely and easily.

In Chapter 12, *The Legacy Problem*, we'll discuss how to deal with a legacy application that may not have been written with testing in mind.

The Legacy Problem

This chapter is all about legacy code. If you have never had to deal with legacy code, count yourself lucky and know that it is coming. Some of you may be permanently stuck in maintenance development. Your life is legacy code. Whatever the situation, this chapter is all about dealing with legacy code. We want to either prevent legacy code from happening, or fight it back to the depths from which it came.

In this chapter, we discuss:

- What makes code legacy
- The issues that legacy code can create
- How legacy code can inhibit testing
- What we can do to deal with and fight back against the legacy problem

What is legacy code?

Most of you have probably had to work on a dreaded legacy project. Working on that project is no fun; the code is a mess, and you want to find whoever wrote it and find out what they were thinking when they wrote it.

At some point in your career, you have been or will be that person to someone else. We all write code that we will not be proud of later. But why does the code get so bad? When does a project become legacy? Lastly, what can be done to prevent this?

Why does code go bad?

In short, code goes bad because we are afraid to change it. Why would the code not changing cause it to be bad? You would hope that, when the code was written, it was the best code that developer was capable of producing at the time. So, that code should have been good, right? This is a complicated answer, but assume, for the moment, that the code was something to be proud of when it was originally written. That still begs the question, how did it go bad?

The answer is staring you in the face. The only reason you are seeing this code is because it needs to change. Chances are, you are not the first person that has needed to make a change in this code. So, this is not the code that was written by a developer doing his or her best to write good code. This code was written by many developers. Still, each of those developers should have been doing their best to write good code. So, again, how did this code go bad?

This is where the fear comes in, because we are afraid to change the code. When we have to change it, we generally try to change the code as little as possible to get the requested update working. After all, we don't want to force ourselves or QA to do a full regression test because we refactored the whole thing, do we? So, we modify the code; we change the way it is expected to work. But we can't change the structure, and we can't modify the decisions of the developer who originally wrote the code.

Over time, making these small changes and being afraid to modify the structure and architectural decisions of the original developer causes severe code rot. Soon, the code will have massive conditional statements and methods that no longer fit on the page. The class containing the code will grow to tens of methods and the file will be thousands of lines long.

When does a project become legacy?

This is a question that is answered by many people in many different ways. Generally, an application has become legacy when no one wants to work on it anymore.

In the beginning, applications are built with a small and defined purpose. Over time, the scope and breadth of a system may grow beyond its original intent. When any change to the application causes the developer to work against what the application was designed for, it will cause friction.

As mentioned previously, the application design is not simply changed because of the fear a developer might have that the application will break. So, more and more cruft is added to the system. Okay, so how long does it take for this to happen? When do we stop hacking new modifications into the existing application and just rewrite it?

Honestly, the cruft starts getting added by the original developer as he or she is writing the application for the first time. When you start to work on a new application, or even just a new feature in an existing application, you start with a preliminary design in mind. Everyone does this. Some developers whiteboard the design or make complete **UML** (**Unified Modeling Language**) diagrams. Other developers just have an idea in their head to guide decisions. Either way, you have a design you want when you sit down to develop an application.

How long is it before you discover an issue with your design and have to start modifying it? You might get one line of code in before you have to change your design, or you might get 75% of the way through before you discover an issue. This is largely determined by the complexity of the problem you are solving and how detailed your planning was. Regardless of the thoroughness of your planning, you will find an issue and have to start changing your design before the first QA review.

The second you make that change, you are adding cruft, so almost all of the time, you are working in a system that was not designed for the code that is being forced into it. In other words, you will probably be writing legacy code the next time you are at work, even if you are working on a new application.

> *Cruft, in software, is any code that is unnecessary or needlessly complex.*

What can be done to prevent legacy decay?

There must be something that can be done to prevent this decay, right? The answer is probably predictable, given the topic of this book. But let me answer with a quote from Michael Feathers on the definition of legacy code:

To me, legacy code is simply code without tests.

- Michael Feathers, Working Effectively with Legacy Code

As we discussed in earlier chapters, tests allow you to refactor. The ability to change the structure of the code is precisely what can prevent the rot and decay that is legacy code.

While tests can allow you to prevent legacy code from forming, be aware that they themselves do not prevent the legacy problem. It takes the dedication of every person on the team understanding that building cruft into a system is a negative behavior and must be avoided. If you feel yourself working against the design of the system, then it is your responsibility to refactor the application into a design that works for today's needs and is flexible enough for tomorrow's needs.

Making a system flexible is not as hard as you might think. Following the SOLID principles (discussed in Chapter 3, *Setting up a JavaScript Environment*) will help to produce a maintainable and flexible system. Even with a flexible system, it takes discipline and determination to maintain a standard of recognizing and fixing friction in the application.

The process of finding that friction could be considered **PDD** (**Pain Driven Development**). This concept means to do the simplest thing to solve your existing problem and actively recognize any friction that arises during future modifications to the application.

PDD can be applied to any system, including the application, your team, and your personal life. Following this strategy, you will become obsessed with removing friction in all things, and can get a little carried away. So, it is important to keep in mind that you might be the only one looking for this friction, and the rest of the world might be ignorant to the pain they are causing themselves. Also, keep in mind that people do not, generally, enjoy having their ignorance pointed out.

Typical issues resulting from legacy code

There is a reason we fear working on legacy code. But, what is it that we fear when working on legacy code? It's not the code itself; the code cannot harm us. Instead, what we fear is hearing that we have introduced a bug. The most dreaded word that a developer can hear. A bug means that we have failed and that we will have to work on the legacy code again.

Exploring the types of issues we might run into while working on legacy code, we find several. Firstly, because we don't know the code, a change to one part might cause unintended side effects in a different part of the application. Another issue is the code could be over-optimized or written by someone who was trying to be clever when they wrote it. Lastly, tight coupling can make updating the code difficult.

Unintended side effects

With all the changes that push an application towards the legacy realm, often the methods or functions in the application will be used in unexpected places, far away from the code that you are changing.

There are two primary violations of the SOLID principles that have led to this issue, and the same two can help you to avoid it going forward. The first is the **OCP** (**Open Closed Principle**), and the second is the **LSP** (**Liskov Substitution Principle**).

Open Closed Principle and legacy code

As discussed previously, the Open Closed Principle states that code should be open for extension yet closed for modification. This principle is designed to prevent the issues with legacy code.

If the modification that has been requested of you is one that will change the behavior of a specific piece of the application, then try to instead clone the method in question and modify the clone. Then the part of the application that needs the change can call the clone instead. This will prevent the change from affecting any parts of the application except the parts you are intending to affect.

Until we know for sure that the code we just avoided is not being used elsewhere in the application, we can't delete it. Eventually, once we are sure that the old code is truly orphaned and not used, we want to clean up and delete the unused method to maintain a code base with just a little less cruft.

On the other hand, if the change is for a bug, then the fix is a little more complicated. You must first determine whether the bug should be fixed everywhere that this code might be used, or whether the bug is relative only to a specific portion of the application. When in doubt, fall back to cloning the method and only affecting intentional parts of the application.

Liskov Substitution Principle and legacy code

How do you determine whether change should affect the entire application or just a slice? One way is to employ the LSP. Simply put, LSP says that a class should do what it sounds like it does. Any behavior change that would change that should be a different dependency.

That is, any change that changes behavior should probably be a new method or a new class with the appropriate method in it. This will prevent accidental side effects in the rest of the application and keep your code cleaner.

Over-optimization

It has been said that premature optimization is a bad thing. What is optimization, though? Generally, to optimize is to reduce the number of steps from point A to point B. In a computer program, that means to reduce the number of cycles required to compute a result.

An unfortunate side effect of optimizing code is that the code usually becomes much more difficult to read and comprehend. Optimizations tend to obfuscate the code in such a way that the only person who can understand it is the person who wrote it, and after some time, they may not be able to understand it either.

It is a fact that hard to understand code is code that is hard to change. This is the reason why optimizations that happen before they are needed are a bad thing.

So, when is an optimization needed? An optimization is needed when it is clear that the current implementation will not be able to meet the demands on the system within a reasonable timeframe in the future.

What makes a timeframe reasonable depends on the complexity of the needed optimization and the speed of the business. When a business is growing quickly, demand will follow along the same curve.

A slower moving company may require several months of planning and preparation before assigning work to a developer. In this situation, it is reasonable to plan for optimizations several months before they are needed. It is important to monitor the performance of an application so that these needs can be predicted.

Overly clever code

Most developers start writing code because they enjoy it. It is not common to find a developer that came into the field simply because they heard they could make lots of money. Working for a company writing the same boring code all the time can cause developers to want to have some fun once in a while.

When developers get bored, they come up with interesting and often overly complicated solutions that are simply not required. Sometimes, developers will come up with the cleverest solution they can figure out to solve a problem.

The problem with clever solutions is that to fix a problem, you have to be more skilled than the person who fixed it. So, if you write the cleverest code you can write, then you are no longer qualified to debug the code, and you bring your own, and everyone else's, progress to a halt.

Tight coupling to third-party software

Everyone uses some third-party plugin or library. In the software community, it is inevitable that you will have to depend on someone else's code. What you don't know when you use that code is the quality, stability, and ability to meet your future needs.

With that in mind, it is a bad idea to rely directly on the classes and interfaces presented to you by that third-party. Instead, use a hexagonal architecture, also known as ports and adapters. For anyone doing C#, this includes abstracting the .NET framework.

Any code that you and your team did not write should be abstracted to protect your code from the potential external changes. This includes code written at your company but by a different team. If it is outside your control, put it behind an abstraction. The preferred abstraction is one or more interfaces that provide the desired functionality.

Issues that prevent adding tests

Deadlines are tight. The scope is ever changing. We just don't have time to write tests. It's more important to get functionality out the door. We've all been there. Whatever the case may be, sometimes you will find yourself working on a project that was not written with testing in mind.

> *There never seems to be enough time to do it right, but there's always time to do it over.*

So, what are the issues you might face that would prevent you from adding tests to a legacy application?

When a system wasn't written with testing in mind, it can be quite difficult to go back and add tests at a later date. Classes with concrete dependencies and tight coupling make software applications difficult to test. Things such as large classes and functions, Law of Demeter violations, global state, and static methods can also make for a system that can be very difficult to test.

Much like building a house by starting with a shaky foundation, untestable code begets untestable code. Unless pieces of the system can be decoupled from the rest of the application, the untestable trend will likely continue, and often does.

Direct dependence on framework and third-party code

As was covered in a previous chapter, dependency on framework and third-party code makes for a tightly coupled system. Any time that the *new* operator in C#, for example, is invoked, then a direct dependency is made to that particular class. We want to minimize those dependencies as much as possible.

Remember that even framework dependencies should be avoided, or, at the very least, abstracted as much as possible. Think back to the `DateTime` example, where we were able to supply our own `DateTime` value for testing purposes in the sample application.

Any `using` or `import` statement at the top of a class or file should be carefully considered and avoided if at all possible. Instead, ensure that your code is dependent on an *interface* whose definition is directly under your control. That way, you can minimize the coupling and isolate the functionality within your own classes and methods. This will help you write cleaner, more testable code.

Law of Demeter

The Law of Demeter, in its simplest form, states that, *Each unit should have only limited knowledge about other units: only units "closely" related to the current unit*. Further, *Each unit should only talk to its friends; don't talk to strangers*. Simply put, *Only talk to your immediate friends*.

When a class or function has knowledge of the inner workings of something outside its immediate control, then there is some tight coupling happening there. In order to test a method that has one or more Law of Demeter violations, the amount of setup involved is often fairly substantial. In order to test a method of one class that violates the Law of Demeter, you must set up the other class or method, or provide a reasonable fake implementation in order to test effectively.

Remember, keep your test methods small and nimble so that they run quickly and are easy to understand. If you follow this rule, your production code will likely also be similarly simple and easy to follow. This will pay off in the long run, as it will be easier to maintain in the future.

Work in the constructor

When a new instance of a class is created that has logic in the constructor, it is often very difficult to test that class. If, for some reason, you need to set up a test scenario that requires different values or behavior than that which is set up in the constructor, it will be quite difficult to proceed. It would be best to minimize the work done in the constructor and extract helper methods or some other scenario more easily tested and better implemented elsewhere.

Keep in mind that particular patterns may prove to be better alternatives to setting up a specific class or function. You should familiarize yourself with common software patterns and how to best implement them. This will help you grow an application by working to solve similar problems that have been resolved by others before you. By utilizing known software patterns, you can more easily communicate your intent with the code within the system.

The builder pattern, for example, might be employed to construct an object with the proper values set that would otherwise be added to the constructor.

Take the following example of the `car` class:

```
public Car(string make, string model, int doors)
{
  Make = make;
  Model = model;
  Doors = doors;
}
```

You could easily write a builder class to construct a specific type of car, such as a `ToyotaCamryBuilder` or `FordMustangBuilder`. Creating a new instance of either a Toyota Camry or a Ford Mustang would be quite easy, simple, and clean. Not to mention, it would be quite easy to test.

Global state

Global state is prone to the side effects from parts of the application far away. These side effects will change the results of the code you run. Functional programming has caught on in recent years, as one of the tenants is to reduce side effects, as they can cause unpredictable and undesirable behavior in a system. Instead, you should strive to break down your code into what are known as pure functions. Pure functions take an input and produce an output. For any given input, the output will always be the same.

Static methods

Static methods are not in and of themselves bad, but they do hint at a misplaced responsibility code smell. Static anything tells you that you have put the code in the wrong place. It doesn't share anything in common with the rest of the code in the scope and should probably be removed and put somewhere with its friends.

Large classes and functions

Does class size really matter? What's the problem with having a large method or function? Large classes and functions often scream complexity. Remember the SOLID principles and what each letter in the acronym represents. A large class or method is likely violating one or more of the principles.

We want our classes and functions to be small and have only one reason to change (*Single Responsibility Principle*). A large class is likely hiding logic that can and should be broken into two or more separate and distinct classes. A large method or function likewise often hides two or more methods. Look for ways to keep your methods simple and keep an eye out for possible logical boundaries with which to break out smaller helpers, classes, and utilities.

Classes and functions should be divided and grouped logically. The purpose of a system should be easily understood by the names and groupings of the files associated with the application. The structure of the system should be simple and make sense to those in charge of enhancing and maintaining the application.

Dealing with legacy problems

We have been discussing all the issues with legacy code. Now it is time to tackle solving those issues. The first thing we must do is bring sanity to the targeted legacy code, and then we can begin testing and eventually fix the code and bring it back from death.

Safe refactoring

The term refactoring is often used incorrectly. When refactoring, you are merely changing the structure of the code. If the logic and/or signature of the code in question changes, then this does not qualify as refactoring. This is a change; most likely a breaking change.

If I'm changing the structure of the code (refactoring), then I don't ever change its behavior at the same time. If I'm changing the interface by which some logic is invoked, I never change the logic itself at the same time.

– Kent Beck

A safe refactoring is one that is guaranteed to not accidentally break the code. Other changes that aren't intended to actually change the behavior of the code but could do it accidentally are considered unsafe refactoring. These usually involve changes to the private areas of the code that aren't directly exposed to consumers of your application.

Converting values to variables

One of the first and easiest things that can be done is to extract any hard-coded values to be represented by variables. Having a variable allows for a quicker and more consistent update. It also helps to convey intent.

When creating the variables, make sure the name is descriptive enough for the scope of the variable. Variables with a short scope can have a short name. On the other hand, variables with a long scope must have a longer, more descriptive name. The further a variable is from its usages, the more descriptive it needs to be so that the context it represents isn't lost.

Check the scope of your variables and make sure they do not have a larger scope than is necessary. Also, check for variables that should have a larger scope but are instead passed between private methods instead of being class members.

It is not recommended at this time to update the private and protected methods that depend on variables that could be moved to a class scope. Instead, make note of them and move them around after tests have been added.

Extracting a method

Working with legacy code often involves working with very large methods. A large method is any method that is longer than twenty lines. Preferably, methods are kept as small as possible, even down to just a few lines.

A large method can mean that the code is violating the Single Responsibility Principle. What needs to be done is to find the seams in the method. Seams can be found by commenting the different sections of the method. Once you have commented the sections, you have identified the seams.

> *Seam*
> *In code, this is the location where two pieces of business logic meet. Normally, you might refer to the location where the private method is called by a public method as a seam in the public method. The code has been stitched together at that location. In this case, there are no private methods, so we are identifying where we want the seams to be.*

Each one of those seams is probably a lower order method that can be extracted. In most editors and IDEs, highlight the code you want and use the extract method refactoring provided through either a right-click, context, menu, or via the menu bar.

Extracting a class

Just like methods, sometimes a large method should really be a class. While extracting methods, if you extract three or more methods, then you have probably found a class that needs to be extracted.

Extracting a class is similar to extracting a method and is likely supported by your editor or IDE. Group and highlight the code you want to extract, then use the extract class menu option.

If your editor does not support extracting a class, all is not lost. Instead, highlight and cut all the methods you extracted that should be in the class. Create a new class file and paste those methods into the new class. Lastly, replace the calls to those methods in the original method with the instantiation and calling of the new class and methods.

Abstracting third-party libraries and framework code

Now that we have variables, methods, and classes abstracted, it is time to abstract third-party libraries, framework code, and those classes we just created.

Firstly, let's start with framework details. Things like `DateTime`, `Random`, and `Console` are best hidden behind classes that you design to fit the needs of your application. There are several reasons for this; most importantly, putting these in their own classes will allow for testing. Without abstracting these to a separate class, it is almost impossible to test with things like `DateTime` that change values on their own.

Next up are the third-party libraries. Anywhere the code is calling to create a new class from a third-party, you need to abstract that to a new class specifically for the purpose of utilizing that third-party library. For the moment, replace the call that instantiates the third-party library with a call that instantiates your class.

Lastly, we can now deal with the calls to `new` that are left in the code. Everywhere that the code is calling `new` needs to be replaced with dependency injection. This will allow for testing and make the code cleaner and more flexible in the future.

To create the dependency injection without modifying the signature of the class, we will be using a pattern called poor man's dependency injection, also known as property injection. Below is an example of property injection in C#. The same process can be done in JavaScript with almost no modifications:

```
public class Example
{
  private Injected _value;
  public Injected Value
  {
    get => _value = _value ?? new Injected();
    set => _value = value;
  }
}
```

Using this pattern, it is possible to allow the class to create its dependency lazily, when asked for it. It is also possible to set the dependencies value for tests or other purposes. Although not shown in this quick example of the pattern, it is better to have the property and backing variable be of an interface type. This will make the injection of some other value easier.

Early tests

If an application of any significant size and complexity isn't properly segmented, it can be quite a daunting task to know how and where to begin writing tests. With a little practice in testing legacy systems it will become easier.

The *when* to write tests within a legacy system can easily be answered with, "When it makes sense to." It would be difficult to sell the idea to any business owner that time (and money) should be spent going back to write tests to cover the existing functionality of a legacy system. It makes much more sense to add tests as enhancements are added to the application or when defects are being addressed. As you're working in the code and have immediate context surrounding the functionality that you wish to test, that is the optimal time to begin to test parts of a legacy application.

So, how do you begin to write tests against a legacy system? Isolate small functions that can be easily tested. Extract methods and smaller classes as needed. Ensure that functionality is not being modified, but that code is simply being reorganized in order to facilitate testability.

It may be necessary to change a private method to be protected so that it may be tested. Changing the scope of the method does make it more available and can reduce the effective abstraction, but if the change is required to aid in testing, the trade-off is almost always worth it. You might also consider that private methods made public might better belong to a different utility or helper class, and so can remain public. It depends on the method in question, but there are certainly options available to help you make a legacy system more testable.

Gold standard tests

Gold standard tests, or characterization tests, are those tests that simply define the expected functionality of a method. If you were to add tests to a legacy system, you would likely begin by writing gold standard tests to define the "happy path" through the system. You might run the application to determine what values a given method returns based on a given input, and then write a test to duplicate the results.

> *Gold standard tests are used because they provide a shortcut. Normally, to test legacy code, you would have to abstract third-party libraries and set up dependency injections of some sort. You may also have to refactor the code significantly just to get to the point where you can test anything. By using a gold standard test, most of this work can, temporarily, be bypassed. The only abstractions needed are screen output, date/time, and random. Just about everything else can be used as is.*

This would provide a baseline for a suite of tests and help ensure that expected functionality does not change with future refactoring or modifications. Gold standard tests do not validate correctness; they merely confirm that the system does what the system did.

As a basis, gold standard tests provide a certain level of comfort to guard against any unwanted behavior changes. These likely will not be enough to provide adequate code coverage and should have additional tests added to cover edge cases and alternate paths through the system.

As the test suite grows and the coverage becomes more meaningful and complete, it may prove wise to remove the original gold standard tests. Again, you want your test suite to be able to execute quickly so that it is run always and often. Removing tests that may be superfluous will help minimize the feedback cycles when running your tests. In other words, you will know if you have broken something faster and will be more likely to run the tests if the tests complete faster.

Testing all potential outcomes

It's not necessarily important to test for all possible values for an individual method. As in the example of gold standard tests, you certainly don't want to run the application with all possible values in order to write tests for each of the possibilities. It is far more important to test for every path of execution.

If a method is small enough and its potential outcomes limited in scope, it should be quite trivial to write a handful of tests to cover all potential scenarios. Take the following method as an example:

```
public int GetPercent(int current, int maximum)
{
  if (maximum == 0)
  {
    throw new DivideByZeroException();
  }

  return (int) ((double) current / maximum * 100);
}
```

What are the potential paths through this method? What tests might you write to ensure adequate coverage?

First, you might consider writing a test in the case that the *maximum* input parameter is equal to `0`. This should cover the `DivideByZeroException` in this scenario.

Next, you might write a test where the *current* parameter is `0`, ensuring that the result of this method is always zero, assuming *maximum* is non-zero.

Finally, you would want to write one or more tests to validate that the algorithm above is indeed calculating the percentage correctly, based on inputs.

At this point, it may be tempting to add tests for things like negative values or to check the rounding that C# is doing, but remember that we are working with legacy code and, as far as the business is concerned, this code is working as is. You don't have a record of the business requirements that spawned this code, so it would be unnecessary, and possibly irresponsible, to test more than what this code is telling you. So, if you believe the code is flawed in that it doesn't cover certain business criteria, or that it could produce incorrect values, discuss these things with your business and make a determination together. Any change to the code would have to be through either a bug or new work.

Moving forward

Once the legacy system has been sufficiently refactored and a comprehensive suite of tests has been added, you may begin to think of the application as non-legacy, current, or a present-day system. It should now be trivial to add new features and squash any newly discovered defects. From this point forward, any new feature requested should be easily added with the confidence that other parts of the system will not be negatively affected.

The legacy application is no longer legacy. With a comprehensive suite of tests, you are now safe to proceed in Test-Driven Development fashion and write tests as every new feature is added. Remember to keep your tests as clean and well-refactored as any part of the production system.

Taking the `GetPercent` example above, how might you modify this in order to return two decimal places? Why, by writing new tests, of course! Start by creating a test to return two decimal places based on the input value.

Your test might look something like this:

```
[Fact]
public void ItReturnsTwoDecimalPlaces()
{
  // Arrange
  // Act
  var result = GetPercent(1, 3);
  // Assert
  Assert.Equal(33.33, result);
}
```

Now, modify the existing method to return only two decimal places. We'll leave this as an exercise for the reader.

Fixing bugs

Fixing bugs in a legacy system is a dangerous endeavor. Remember that any existing behavior may be accounted for in other parts of your system, or by external consumers of your application. By fixing a bug, you may be breaking functionality, albeit wrong, that someone else is depending on. So, a change to the execution results of code should be considered carefully before being done.

Free to do unsafe refactoring

Refactoring is, by definition, modifying the structure of the code without modifying its behavior. Safe refactoring includes variable injection, method extraction, and so on. Unsafe refactoring would affect the architecture of the code, the way the code interacts with the rest of the system, and more. By having a fully tested section of code, you can now modify the architecture and be assured that this section still does what it is supposed to do.

Summary

In this chapter, we discussed how we define legacy code and the issues that legacy code can create. Legacy code can inhibit testing, but now you should know how to fight back against the legacy problem.

In Chapter 13, *Unraveling a Mess*, we'll explore a rather extreme example of the types of things you might encounter in a legacy system. We'll explore safe refactoring and how to best unravel a mess into well structured, testable code.

Unraveling a Mess

Not all applications were written with testing in mind. Few were originally developed using TDD. Often, the original developers are long gone, and documentation is incorrect, incomplete, or missing entirely.

In this chapter, we will gain an understanding of:

- Dealing with inherited code
- Characterization tests
- Refactoring with tests

Inheriting code

This chapter is a case study of legacy code that needs (what should be) a minor change. We will quickly find out that the change is not so minor. To begin, let's look at what the legacy application does.

Here is some sample output from a run of this code:

```
Take a guess: AAAA
---+
Take a guess: BBBA
-+-+
Take a guess: CBCA
++
Take a guess: DBDA
++-+

Take a guess: DBEA
++++
Congratulations you guessed the password in 5 tries.
Press any key to quit.
```

Looking at the interactions, this program doesn't look that bad. In speaking with the business analyst, the application was explained as a game.

The game

This particular game is called *Mastermind* and is a code breaking puzzle. According to the business analyst, the code consists of the letters A through F and contains four of the letters chosen at random. It is the goal of the player to determine the passcode.

The player is given hints along the way. For a correctly placed letter, the player receives a plus symbol. For a correct letter in the wrong position, the player receives a minus symbol. If the letter is incorrect, the player receives no symbol.

A change is requested

During play testing, it was determined that players were discovering the passcodes too quickly. As a result, the game wasn't as much fun as it could be. The suggested solution was to make the passcode more complex by allowing more than six letters to be used. It is our job to extend the character range to A through Z.

We can start by looking at the existing code to determine where we might have to make the change. That is where we discover this!

In the file `Program.cs`:

```
class Program
{
  static void Main(string[] args)
  {
    char[] g;
    char[] p = new[] { 'A', 'A', 'A', 'A' };
    int i = 0;
    int j = 0;
    int x = 0;
    int c = 0;
    Random rand = new Random(DateTime.Now.Millisecond);
    if (args.Length > 0 && args[0] != null) p = args[0].ToCharArray();
    else goto randomize_password;
    guess: Console.Write("Take a guess: ");
    g = Console.ReadLine().ToArray();
    i = i + 1;
    if (g.Length != 4) goto wrong_size;
    if (g == p) goto success;
    x = 0;
    c = 0;
    check_loop:
    if (g[x] > 65 + 26) g[x] = (char)(g[x] - 32);
    if (g[x] == p[x]) Console.Write("+", c = c + 1);
    else if (p.Contains(g[x])) Console.Write("-");
    x = x + 1;
    if (x < 4) goto check_loop;
    Console.WriteLine();
    if (c == 4) goto success;
    goto guess;
    success: Console.WriteLine("Congratulations you guessed the
    password in " + i + " tries.");
    goto end;
    wrong_size: Console.WriteLine("Password length is 4.");
    goto guess;
    randomize_password: j = 0;
    password_loop: p[j] = (char)(rand.Next(6) + 65);
    j = j + 1;
    if (j < 4) goto password_loop;
    goto guess;
    end: Console.WriteLine("Press any key to quit.");
    Console.ReadKey();
  }
}
```

We now have several problems. Firstly, it's not exactly clear where the letters are coming from. Secondly, there is no way this code is tested. Lastly, even if making the change were straight forward, making sure we didn't break something would not be. We have to do a full manual

regression test to verify that any of this is working, and trying to verify that all letters, A through Z, are possible may take a very long time.

Life sometimes hands you lemons

While I hope you never receive code this bad, we are going to walk through what is needed to turn even this into readable, maintainable, and fully tested code. The best part, the part you aren't going to believe, is that transforming this code is actually safe and fairly easy.

Getting started

In any code situation like this, the first thing we must do is remove the code in question from the environment where we have no control. In this case, we can't test the code if it is sitting in `Program.main`. So, let's grab the whole thing and put it into a class named `Mastermind`. We will have a single function named `Play` that will run the game. This is considered a safe refactoring, because we are not changing any of the existing code, simply moving it somewhere else.

In the file `Program.cs`:

```
class Program
{
  static void Main(string[] args)
  {
    var game = new Mastermind();
    game.Play(args);
  }
}
```

In the file `Mastermind.cs`:

```
class Mastermind
{
  public void Play(string[] args)
  {
    char[] g;
    char[] p = new[] { 'A', 'A', 'A', 'A' };
    int i = 0;
    int j = 0;
    int x = 0;
    int c = 0;
    Random rand = new Random(DateTime.Now.Millisecond);
    if (args.Length > 0 && args[0] != null) p = args[0].ToCharArray();
    else goto randomize_password;
    guess: Console.Write("Take a guess: ");
    g = Console.ReadLine().ToArray();
    i = i + 1;
    if (g.Length != 4) goto wrong_size;
    if (g == p) goto success;
    x = 0;
    c = 0;
    check_loop:
    if (g[x] > 65 + 26) g[x] = (char)(g[x] - 32);
    if (g[x] == p[x]) Console.Write("+", c = c + 1);
    else if (p.Contains(g[x])) Console.Write("-");
    x = x + 1;
    if (x < 4) goto check_loop;
    Console.WriteLine();
    if (c == 4) goto success;
    goto guess;
    success: Console.WriteLine("Congratulations you guessed the
    password in " + i + " tries.");
    goto end;
    wrong_size: Console.WriteLine("Password length is 4.");
    goto guess;
    randomize_password: j = 0;
    password_loop: p[j] = (char)(rand.Next(6) + 65);
    j = j + 1;
    if (j < 4) goto password_loop;
    goto guess;
    end: Console.WriteLine("Press any key to quit.");
```

```
      Console.ReadKey();
    }
}
```

Running the code again at this point shows that everything still works. The next step is a cosmetic one; let's spread the `Play` method out into sections. This should help us determine what private methods exist inside the large public method.

In the file `Mastermind.cs`:

```
class Mastermind
{
  public void Play(string[] args)
  {
    // Variable Declarations - Global??
    char[] g;
    char[] p = new[] { 'A', 'A', 'A', 'A' };
    int i = 0;
    int j = 0;
    int x = 0;
    int c = 0;
    // Initialize randomness
    Random rand = new Random(DateTime.Now.Millisecond);
    // Determine if a password was passed in?
    if (args.Length > 0 && args[0] != null) p = args[0].ToCharArray();
    else goto randomize_password; // Create a password if one was not
    provided
    // Player move - guess the password
    guess: Console.Write("Take a guess: ");
    g = Console.ReadLine().ToArray();
    i = i + 1;
    if (g.Length != 4) goto wrong_size;
    if (g == p) goto success;
    x = 0;
    c = 0;
    // Check if the password provided by the player is correct
    check_loop:
    if (g[x] > 65 + 26) g[x] = (char)(g[x] - 32);
    if (g[x] == p[x]) Console.Write("+", c = c + 1);
    else if (p.Contains(g[x])) Console.Write("-");
    x = x + 1;
    if (x < 4) goto check_loop; // Still checking??
    Console.WriteLine();
    if (c == 4) goto success; // Password must have been correct
    goto guess; // No correct, try again
    // Game over you win
    success: Console.WriteLine("Congratulations you guessed the
    password in " + i + " tries.");
    goto end;
    // Password guess was wrong size - Error Message
    wrong_size: Console.WriteLine("Password length is 4.");
    goto guess;
    // Create a random password
    randomize_password: j = 0;
    password_loop: p[j] = (char)(rand.Next(6) + 65);
    j = j + 1;
    if (j < 4) goto password_loop;
    goto guess; // Start the game
    // Game is complete - exit
    end: Console.WriteLine("Press any key to quit.");
    Console.ReadKey();
  }
}
```

We have now used whitespace to split the program into several pieces and added comments explaining what we think each piece is doing. At this point, we are almost ready to begin testing. We have just a couple things in the way, the worst of which is the `Console` class.

Abstracting a third-party class

If we tried to test right now, the application would hit the first `ReadLine` call and the test would time out. Console has the ability to redirect the input and output, but we are not going to use this feature, because it is specific to Console and we want to demonstrate a more generic solution that you can apply anywhere.

What we need is a class that gives us a similar interface to Console. Then we can dependency inject our class for the tests and a thin wrapper for the production code. Let's test drive that interface now.

In the file `InputOutputTests.cs`:

```
public class InputOutputTests
{
  [Fact]
  public void ItExists()
  {
    var inout = new MockInputOutput();
  }
}
```

In the file `ReadLineTests.cs`:

```
public class ReadLineTests
{
  [Fact]
  public void ItCanBeReadFrom()
  {
    var inout = new MockInputOutput();
    inout.InFeed.Enqueue("Test");

    // Act
    var input = inout.ReadLine();
  }

  [Fact]
  public void ProvidedInputCanBeRetrieved()
  {
    // Arrange
    var inout = new MockInputOutput();
    inout.InFeed.Enqueue("Test");

    // Act
    var input = inout.ReadLine();

    // Assert
    Assert.Equal("Test", input);
  }

  [Fact]
  public void ProvidedInputCanBeRetrievedInSuccession()
  {
    // Arrange
    var inout = new MockInputOutput();
    inout.InFeed.Enqueue("Test 1");
    inout.InFeed.Enqueue("Test 2");

    // Act
    var input1 = inout.ReadLine();
```

```
    var input2 = inout.ReadLine();

    // Assert
    Assert.Equal("Test 1", input1);
    Assert.Equal("Test 2", input2);
  }
}
```

In the file `ReadTests.cs`:

```
public class ReadTests
{
  [Fact]
  public void ItCanBeReadFrom()
  {
    var inout = new MockInputOutput();
    inout.InFeed.Enqueue("T");

    // Act
    var input = inout.Read();
  }

  [Fact]
  public void ProvidedInputCanBeRetrieved()
  {
    // Arrange
    var inout = new MockInputOutput();
    inout.InFeed.Enqueue("T");

    // Act
    var input = inout.Read();

    // Assert
    Assert.Equal('T', input);
  }

  [Fact]
  public void ProvidedInputCanBeRetrievedInSuccession()
  {
    // Arrange
    var inout = new MockInputOutput();
    inout.InFeed.Enqueue("T");
    inout.InFeed.Enqueue("E");

    // Act
    var input1 = inout.Read();
    var input2 = inout.Read();

    // Assert
    Assert.Equal('T', input1);
    Assert.Equal('E', input2);
  }
}
```

In the file `WriteTests.cs`:

```
public class WriteTests
{
  [Fact]
  public void ItCanBeWrittenTo()
  {
    var inout = new MockInputOutput();

    // Act
    inout.Write("Text");
  }

  [Fact]
  public void WrittenTextCanBeRetrieved()
  {
    // Arrange
```

```
      var inout = new MockInputOutput();
      inout.Write("Text");

      // Act
      var writtenText = inout.OutFeed;

      // Assert
      Assert.Single(writtenText);
      Assert.Equal("Text", writtenText.First());
    }
}
```

In the file `WriteLineTests.cs`:

```
public class WriteLineTests
{
  [Fact]
  public void ItCanBeWrittenTo()
  {
    var inout = new MockInputOutput();

    // Act
    inout.WriteLine("Text");
  }

  [Fact]
  public void WrittenTextCanBeRetrieved()
  {
    // Arrange
    var inout = new MockInputOutput();
    inout.WriteLine("Text");

    // Act
    var writtenText = inout.OutFeed;

    // Assert
    Assert.Single(writtenText);
    Assert.Equal("Text" + Environment.NewLine, writtenText.First());
  }
}
```

In the file `IInputOutput.cs`:

```
public interface IInputOutput
{
  void Write(string text);
  void WriteLine(string text);
  char Read();
  string ReadLine();
}
```

In the file `MockInputOutput.cs`:

```
public class MockInputOutput : IInputOutput
{
  public List<string> OutFeed { get; set; }
  public Queue<string> InFeed { get; set; }

  public MockInputOutput()
  {
    OutFeed = new List<string>();
    InFeed = new Queue<string>();
  }

  public void Write(string text)
  {
    OutFeed.Add(text);
  }
```

```csharp
    public void WriteLine(string text)
    {
      OutFeed.Add(text + Environment.NewLine);
    }

    public char Read()
    {
      return InFeed.Dequeue().ToCharArray().First();
    }

    public string ReadLine()
    {
      return InFeed.Dequeue();
    }
  }
}
```

That handles our mock input and output, but we need to create the production wrapper class for Console, and we need to use `Program.cs` to inject that class into the `Mastermind` class.

Unexpected Input

While replacing calls to Console with calls to our injected class, we found two use cases that we did not plan for. The first use case is fairly involved and has a couple parameters we need to handle:

```
Console.Write("+", c = c + 1);
```

The second use case is more simple and doesn't take any parameters:

```
Console.WriteLine();
```

The second use case is the easiest to deal with, so let us write a quick test for that now.

In the file `WriteLineTests.cs`:

```
[Fact]
public void ItCanWriteABlankLine()
{
  // Arrange
  var inout = new MockInputOutput();

  // Act
  inout.WriteLine();

  // Assert
  Assert.Single(inout.OutFeed);
  Assert.Equal(Environment.NewLine, inout.OutFeed.First());
}
```

In the file `IInputOutput.cs`:

```
void WriteLine(string text = null);
```

In the file `MockInputOutput.cs`:

```
public void WriteLine(string text = null)
{
  OutFeed.Add((text ?? "") + Environment.NewLine);
}
```

The next issue is slightly more complicated. If we want to handle it accurately, we will need to do quite a bit of regular expression and string manipulation. However, we don't need it to be "correct"; we only need it to work as expected by the application. In the singular case where this is being used, the value that should be placed into the string being written, isn't. The original developer abused the functionality of `Console.Write` to reduce the number of lines in the if statement so they could avoid brackets. So, all we need to do for the code to continue to work is allow for the input to take place. A simple interface extension should provide that for us.

In the file `IInputOutput.cs`:

```
void Write(string text, params object[] args);
```

In the file `MockInputOutput.cs`:

```
public void Write(string text, params object[] args)
{
  OutFeed.Add(text);
}
```

Back in the application code, we can finish making our changes. Here is the updated application.

In the file `ConsoleInputOutput.cs`:

```
public class ConsoleInputOutput : IInputOutput
{
  public void Write(string text, params object[] args)
  {
    Console.Write(text, args);
  }

  public void WriteLine(string text)
  {
    Console.WriteLine(text);
  }

  public char Read()
  {
    return Console.ReadKey().KeyChar;
  }

  public string ReadLine()
  {
    return Console.ReadLine();
  }
}
```

In the file `Program.cs`:

```
class Program
{
  static void Main(string[] args)
  {
    var inout = new ConsoleInputOutput();
    var game = new Mastermind(inout);
    game.Play(args);
  }
}
```

In the file `Mastermind.cs`:

```
public class Mastermind
{
  private readonly IInputOutput _inout;

  public Mastermind(IInputOutput inout)
  {
    _inout = inout;
  }

  public void Play(string[] args)
  {
    // Variable Declarations - Global??
    char[] g;
    char[] p = new[] { 'A', 'A', 'A', 'A' };
    int i = 0;
    int j = 0;
    int x = 0;
    int c = 0;

    // Initialize randomness
```

```
        Random rand = new Random(DateTime.Now.Millisecond);

        // Determine if a password was passed in?
        if (args.Length > 0 && args[0] != null) p = args[0].ToCharArray();
        else goto randomize_password; // Create a password if one was not
        provided
        // Player move - guess the password
        guess: _inout.Write("Take a guess: ");
        g = _inout.ReadLine().ToArray();
        i = i + 1;
        if (g.Length != 4) goto wrong_size;
        if (g == p) goto success;
        x = 0;
        c = 0;

        // Check if the password provided by the player is correct
        check_loop:
        if (g[x] > 65 + 26) g[x] = (char)(g[x] - 32);
        if (g[x] == p[x]) _inout.Write("+", c = c + 1);
        else if (p.Contains(g[x])) _inout.Write("-");
        x = x + 1;
        if (x < 4) goto check_loop; // Still checking??
        _inout.WriteLine();
        if (c == 4) goto success; // Password must have been correct
        goto guess; // No correct, try again

        // Game over you win
        success: _inout.WriteLine("Congratulations you guessed the password
        in " + i + " tries.");
        goto end;
        // Password guess was wrong size - Error Message
        wrong_size: _inout.WriteLine("Password length is 4.");
        goto guess;

        // Create a random password
        randomize_password: j = 0;
        password_loop: p[j] = (char)(rand.Next(6) + 65);
        j = j + 1;
        if (j < 4) goto password_loop;
        goto guess; // Start the game

        // Game is complete - exit
        end: _inout.WriteLine("Press any key to quit.");
        _inout.Read();
    }
}
```

A quick test run confirms that the application is working correctly:

```
Take a guess: AAAA
Take a guess: BBBB
Take a guess: CCCC
Take a guess: DDDD
+-++
Take a guess: DEDD
+++
Take a guess: DFDD
++++
Congratulations you guessed the password in 6 tries.

Press any key to quit.
```

Now, we can write a gold standard or characterization test that will verify all the parts of the code are working correctly. The only piece of the code this test will not cover is the random password generation:

```
public class GoldStandardTests
{
  [Fact]
  public void StandardTestRun()
```

```
  {
    // Arrange
    var inout = new MockInputOutput();
    var game = new Mastermind(inout);

    // Arrange - Inputs
    inout.InFeed.Enqueue("AAA");
    inout.InFeed.Enqueue("AAAA");
    inout.InFeed.Enqueue("ABBB");
    inout.InFeed.Enqueue("ABCC");
    inout.InFeed.Enqueue("ABCD");
    inout.InFeed.Enqueue("ABCF");
    inout.InFeed.Enqueue(" ");

    // Arrange - Outputs
    var expectedOutputs = new Queue<string>();
    expectedOutputs.Enqueue("Take a guess: ");
    expectedOutputs.Enqueue("Password length is 4." +
     Environment.NewLine);
    expectedOutputs.Enqueue("Take a guess: ");
    expectedOutputs.Enqueue("+");
    expectedOutputs.Enqueue("-");
    expectedOutputs.Enqueue("-");
    expectedOutputs.Enqueue("-");
    expectedOutputs.Enqueue(Environment.NewLine);
    expectedOutputs.Enqueue("Take a guess: ");
    expectedOutputs.Enqueue("+");
    expectedOutputs.Enqueue("+");
    expectedOutputs.Enqueue("-");
    expectedOutputs.Enqueue("-");
    expectedOutputs.Enqueue(Environment.NewLine);
    expectedOutputs.Enqueue("Take a guess: ");
    expectedOutputs.Enqueue("+");
    expectedOutputs.Enqueue("+");
    expectedOutputs.Enqueue("+");
    expectedOutputs.Enqueue("-");
    expectedOutputs.Enqueue(Environment.NewLine);
    expectedOutputs.Enqueue("Take a guess: ");
    expectedOutputs.Enqueue("+");
    expectedOutputs.Enqueue("+");
    expectedOutputs.Enqueue("+");
    expectedOutputs.Enqueue(Environment.NewLine);
    expectedOutputs.Enqueue("Take a guess: ");
    expectedOutputs.Enqueue("+");
    expectedOutputs.Enqueue("+");
    expectedOutputs.Enqueue("+");
    expectedOutputs.Enqueue("+");
    expectedOutputs.Enqueue(Environment.NewLine);
    expectedOutputs.Enqueue("Congratulations you guessed the password
     in 6 tries." + Environment.NewLine);
    expectedOutputs.Enqueue("Press any key to quit." +
     Environment.NewLine);
    // Act
    game.Play(new[] { "ABCF" });

    // Assert
    inout.OutFeed.ForEach((text) =>
    {
      Assert.Equal(expectedOutputs.Dequeue(), text);
    });
  }
}
```

This is an extremely long test, and it has an out of the ordinary structure, but this single test runs through almost all the logic in the application. You may not always be able to do this with a single test, but before beginning any heavy refactoring, these tests must exist.

Making sense of the madness

Now that we have the gold standard test written, we can begin to safely refactor the code. Any changes that we try to make that break the gold standard test will have to be undone and a new approach will have to be taken.

Looking at the `Mastermind` class, all those variables at the top of the `Play` method can be moved out to be class level fields. This will make them available to all the code within the class and help to both figure out what they are for and how often they are used in the app:

```
private char[] g;
private char[] p = new[] { 'A', 'A', 'A', 'A' };
private int i = 0;
private int j = 0;
private int x = 0;
private int c = 0;
```

Next, we will just work our way down the `Play` method, extracting all that we can into tiny private methods. We are only able to do some tiny refactoring before we need to switch gears and start fixing some of the antiquated logic in this application:

```
public class Mastermind
{
  private readonly IInputOutput _inout;
  private char[] g;
  private char[] p = new[] { 'A', 'A', 'A', 'A' };
  private int i = 0;
  private int j = 0;
  private int x = 0;
  private int c = 0;

  public Mastermind(IInputOutput inout)
  {
    _inout = inout;
  }

  public void Play(string[] args)
  {
    // Determine if a password was passed in?
    if (args.Length > 0 && args[0] != null) p = args[0].ToCharArray();
    else CreateRandomPassword(); // Create a password if one was not
    provided
    // Player move - guess the password
    guess:
    _inout.Write("Take a guess: ");
    g = _inout.ReadLine().ToArray();
    i = i + 1;
    if (g.Length != 4) goto wrong_size;
    if (g == p) goto success;
    x = 0;
    c = 0;

    // Check if the password provided by the player is correct
    check_loop:
    if (g[x] > 65 + 26) g[x] = (char)(g[x] - 32);
    if (g[x] == p[x]) _inout.Write("+", c = c + 1);
    else if (p.Contains(g[x])) _inout.Write("-");
    x = x + 1;
    if (x < 4) goto check_loop; // Still checking??
    _inout.WriteLine();
```

```
    if (c == 4) goto success; // Password must have been correct
    goto guess; // No correct, try again

    // Password guess was wrong size - Error Message
    wrong_size: _inout.WriteLine("Password length is 4.");
    goto guess;

    // Game over you win
    success: _inout.WriteLine("Congratulations you guessed the password
     in " + i + " tries.");
    _inout.WriteLine("Press any key to quit.");
    _inout.Read();
  }
  private void CreateRandomPassword()
  {
    // Initialize randomness
    Random rand = new Random(DateTime.Now.Millisecond);

    j = 0;

    password_loop:
    p[j] = (char)(rand.Next(6) + 65);
    j = j + 1;
    if (j < 4) goto password_loop;
  }
}
```

We were able to break out a password generation method. We were also able to simplify the
structure of the success code. We cannot, however, proceed without addressing the complexity
of the chosen looping structures. The developer that wrote this did not use general looping
structures, such as while and for loops. We need to fix that in order to better understand and
work with this code:

```
public void Play(string[] args)
{
  // Determine if a password was passed in?
  if (args.Length > 0 && args[0] != null) p = args[0].ToCharArray();
  else CreateRandomPassword(); // Create a password if one was not
   provided
  // Player move - guess the password
  while (c != 4)
  {
    _inout.Write("Take a guess: ");
    g = _inout.ReadLine().ToArray();

    i = i + 1;

    if (g.Length != 4)
    {
      // Password guess was wrong size - Error Message
      _inout.WriteLine("Password length is 4.");
    }
    else
    {
      // Check if the password provided by the player is correct
      for (x = 0, c = 0; g.Length == 4 && x < 4; x++)
      {
        if (g[x] > 65 + 26) g[x] = (char)(g[x] - 32);
        if (g[x] == p[x]) _inout.Write("+", c = c + 1);
        else if (p.Contains(g[x])) _inout.Write("-");
      }

      _inout.WriteLine();
    }
  }

  // Game over you win
  _inout.WriteLine("Congratulations you guessed the password in " + i +
  " tries.");
  _inout.WriteLine("Press any key to quit.");
```

```
    _inout.Read();
  }
```

We now have a structure that we can begin to work with. Let's start by making some sense of these variable names:

```
C ~= Correct Letter Guesses
G ~= Current Guess
P ~= Password
I ~= Tries
X ~= Loop Index / Pointer to Guess Character being checked
J ~= Loop Index / Pointer to Password Character being generated
```

We will want to make updates to the `Play` method that reflect our determinations for what the variables mean. Following we have replaced the single letter variable names with names that more appropriately represent what the variables are used for:

```
public void Play(string[] args)
{
  // Determine if a password was passed in?
  if (args.Length > 0 && args[0] != null) password =
   args[0].ToCharArray();
  else CreateRandomPassword(); // Create a password if one was not
   provided
  // Player move - guess the password
  while (correctPositions != 4)
  {
    _inout.Write("Take a guess: ");
    guess = _inout.ReadLine().ToArray();

    tries = tries + 1;

    if (guess.Length != 4)
    {
      // Password guess was wrong size - Error Message
      _inout.WriteLine("Password length is 4.");
    }
    else
    {
      // Check if the password provided by the player is correct
      for (x = 0, correctPositions = 0; x < 4; x++)
      {
        if (guess[x] > 65 + 26) guess[x] = (char)(guess[x] - 32);
        if (guess[x] == password[x]) _inout.Write("+", correctPositions
          = correctPositions + 1);
        else if (password.Contains(guess[x])) _inout.Write("-");
      }
      _inout.WriteLine();
    }
  }
  // Game over you win
  _inout.WriteLine("Congratulations you guessed the password in " +
   tries + " tries.");
  _inout.WriteLine("Press any key to quit.");
  _inout.Read();
}
```

Next, it would be nice if we could update the interface now that we understand the application a little better. Two things that we would like to change are the input and the very end of the game. It would be nice if the input was a simple string instead of a character array. The `Play` method could take a string and the program could figure out how to get the password string from the arguments.

Along those same lines, we could reduce the overall number of writes and turn the consecutive plus and minus `Write` commands into a single `WriteLine` command. This would break our gold

standard test, but wouldn't actually change the functionality of the code. It would still print the pluses and minuses on a single line.

To convert the guess from a character array to a string, we must first understand what is happening on this line:

```
if (guess[x] > 65 + 26) guess[x] = (char)(guess[x] - 32);
```

Analyzing the line, we see the numbers 65, 26, and 32. If you are familiar with ASCII codes, then these lines might make sense to you. The number 65 is the starting point of the alphabet characters on the ASCII tables. There are 26 letters in the English alphabet. And, there are 32 values between "a" and "A". So, it is to be assumed that this code is either uppercasing or lowercasing the character at the specified index. We can approximate this in C# using the `String.ToUpper()` method.

While we are doing a small bit of gold standard changes, we should also remove the last two lines of the `Play` method and move them to `Program.cs`, as they are more related to a Console application than anything else.

In the file `Program.cs`:

```csharp
class Program
{
  static void Main(string[] args)
  {
    var inout = new ConsoleInputOutput();
    var game = new Mastermind(inout);

    var password = args.Length > 0 ? args[0] : null;
    game.Play(password);

    inout.WriteLine("Press any key to quit.");
    inout.Read();
  }
}
```

In the file `Mastermind.cs`:

```csharp
public class Mastermind
{
  private readonly IInputOutput _inout;
  private string guess;
  private int tries;
  private int correctPositions;

  public Mastermind(IInputOutput inout)
  {
    _inout = inout;
  }

  public void Play(string password = null)
  {
    // Determine if a password was passed in?
    password = password ?? CreateRandomPassword();

    // Player move - guess the password
    while (correctPositions != 4)
    {
      _inout.Write("Take a guess: ");
      guess = _inout.ReadLine();
      tries = tries + 1;

      if (guess.Length != 4)
```

```
      {
        // Password guess was wrong size - Error Message
        _inout.WriteLine("Password length is 4.");
      }
      else
      {
        // Check if the password provided by the player is correct
        guess = guess.ToUpper();
        var guessResult = "";

        for (var x = 0; x < 4; x++)
        {
          if (guess[x] == password[x])
          {
            guessResult += "+";
          }
          else if (password.Contains(guess[x]))
          {
            guessResult += "-";
          }
        }

        correctPositions = guessResult.Count(c => c == '+');
        _inout.WriteLine(guessResult);
      }
    }

    // Game over you win
    _inout.WriteLine("Congratulations you guessed the password in " +
     tries + " tries.");
  }

  private string CreateRandomPassword()
  {
    // Initialize randomness
    Random rand = new Random(DateTime.Now.Millisecond);

    var password = new [] {'A', 'A', 'A', 'A'};

    var j = 0;

    password_loop:
    password[j] = (char)(rand.Next(6) + 65);
    j = j + 1;

    if (j < 4) goto password_loop;
    return password.ToString();
  }
}
```

In the file `GoldStandardTests.cs`:

```
public class GoldStandardTests
{
  [Fact]
  public void StandardTestRun()
  {
    // Arrange
    var inout = new MockInputOutput();
    var game = new Mastermind(inout);

    // Arrange - Inputs
    inout.InFeed.Enqueue("AAA");
    inout.InFeed.Enqueue("AAAA");
    inout.InFeed.Enqueue("ABBB");
    inout.InFeed.Enqueue("ABCC");
    inout.InFeed.Enqueue("ABCD");
    inout.InFeed.Enqueue("ABCF");
    inout.InFeed.Enqueue(" ");

    // Arrange - Outputs
    var expectedOutputs = new Queue<string>();
```

```csharp
      expectedOutputs.Enqueue("Take a guess: ");
      expectedOutputs.Enqueue("Password length is 4." +
      Environment.NewLine);
      expectedOutputs.Enqueue("Take a guess: ");
      expectedOutputs.Enqueue("+---" + Environment.NewLine);
      expectedOutputs.Enqueue("Take a guess: ");
      expectedOutputs.Enqueue("++--" + Environment.NewLine);
      expectedOutputs.Enqueue("Take a guess: ");
      expectedOutputs.Enqueue("+++-" + Environment.NewLine);
      expectedOutputs.Enqueue("Take a guess: ");
      expectedOutputs.Enqueue("+++" + Environment.NewLine);
      expectedOutputs.Enqueue("Take a guess: ");
      expectedOutputs.Enqueue("++++" + Environment.NewLine);
      expectedOutputs.Enqueue("Congratulations you guessed the password
      in 6 tries." + Environment.NewLine);

      // Act
      game.Play("ABCF");

      // Assert
      inout.OutFeed.ForEach(text =>
      {
        Assert.Equal(expectedOutputs.Dequeue(), text);
      });
  }
}
```

Final beautification

Now that everything else is done and the code is working correctly, it is time for the last refactoring before we start our enhancement. We want our methods to be as small as possible. In this case, that means that the `Play` function should have practically no logic outside the main game loop.

In general, if a method has any kind of block in it (for example, if, while, for, and so on), we want that block to be the only thing in the method. Often, there are also guard statements checking input, but that should be it. Let's refactor to follow that convention and see what the code looks like afterwards.

In the file `Mastermind.cs`:

```
public class Mastermind
{
  private readonly IInputOutput _inout;
  private int _tries;

  public Mastermind(IInputOutput inout)
  {
    _inout = inout;
  }

  public void Play(string password = null)
  {
    password = password ?? CreateRandomPassword();
    var correctPositions = 0;

    while (correctPositions != 4)
    {
      correctPositions = GuessPasswordAndCheck(password);
    }

    _inout.WriteLine("Congratulations you guessed the password in " +
    _tries + " tries.");
  }

  private int GuessPasswordAndCheck(string password)
  {
    var guess = Guess();
    return Check(guess, password);
  }

  private int Check(string guess, string password)
  {
    var checkResult = "";

    for (var x = 0; x < 4; x++)
    {
      if (guess[x] == password[x])
      {
        checkResult += "+";
      }
      else if (password.Contains(guess[x]))
      {
        checkResult += "-";
      }
    }
```

```
      _inout.WriteLine(checkResult);
      return checkResult.Count(c => c == '+');
    }

  private string Guess()
  {
    _tries = _tries + 1;

    _inout.Write("Take a guess: ");
    var guess = _inout.ReadLine();

    if (guess.Length == 4)
    {
      return guess.ToUpper();
    }

    // Password guess was wrong size - Error Message
    _inout.WriteLine("Password length is 4.");
    return Guess();
  }

  private string CreateRandomPassword()
  {
    // Initialize randomness
    Random rand = new Random(DateTime.Now.Millisecond);

    var password = new[] { 'A', 'A', 'A', 'A' };

    var j = 0;

    password_loop:
    password[j] = (char)(rand.Next(6) + 65);
    j = j + 1;

    if (j < 4) goto password_loop;

    return password.ToString();
  }
}
```

There are many ways that this code could have been refactored; this is just one. Now that the code is refactored, we are ready to move on and begin working on enhancements.

Ready for enhancements

We are now to a point where the code makes enough sense that we can begin to work on our change requests. We have broken the random password generation portion of the code into its own method, so now we can work on it independently.

One of the first things we need to do is to stop using `Random`. `Random` is, by nature, unpredictable and outside of our control. We need a way to feed the number generation to verify that we can get the expected outputs when `Random` provides specific inputs.

We will extract an interface and mock class similar to what we did for Console. Here is the first round of tests, the mock class, and the interface that were created.

In the file `RandomNumberTests.cs`:

```
public class RandomNumberTests
{
  private readonly MockRandomGenerator _rand;

  public RandomNumberTests()
  {
    _rand = new MockRandomGenerator();
  }

  [Fact]
  public void ItExists()
  {
    _rand.Number();
  }

  [Fact]
  public void ItReturnsDefaultValue()
  {
    // Act
    var result = _rand.Number();

    // Assert
    Assert.Equal(0, result);
  }

  [Fact]
  public void ItCanReturnPredeterminedNumbers()
  {
    // Arrange
    _rand.SetNumbers(1, 2, 3, 4, 5);

    // Act
    var a = _rand.Number();
    var b = _rand.Number();
    var c = _rand.Number();
    var d = _rand.Number();
    var e = _rand.Number();

    // Arrange
    Assert.Equal(1, a);
    Assert.Equal(2, b);
    Assert.Equal(3, c);
    Assert.Equal(4, d);
    Assert.Equal(5, e);
  }
```

```csharp
    [Fact]
    public void ItCanHaveAMaxRange()
    {
      // Arrange
      const int maxRange = 3;
      _rand.SetNumbers(1, 2, 3, 4, 5);

      // Act
      var a = _rand.Number(maxRange);
      var b = _rand.Number(maxRange);
      var c = _rand.Number(maxRange);
      var d = _rand.Number(maxRange);
      var e = _rand.Number(maxRange);

      // Arrange
      Assert.Equal(1, a);
      Assert.Equal(2, b);
      Assert.Equal(3, c);
      Assert.Equal(3, d);
      Assert.Equal(3, e);
    }

    [Fact]
    public void ItCanHaveAMinMaxRange()
    {
      // Arrange
      const int minRange = 2;
      const int maxRange = 3;
      _rand.SetNumbers(1, 2, 3, 4, 5);

      // Act
      var a = _rand.Number(minRange, maxRange);
      var b = _rand.Number(minRange, maxRange);
      var c = _rand.Number(minRange, maxRange);
      var d = _rand.Number(minRange, maxRange);
      var e = _rand.Number(minRange, maxRange);

      // Arrange
      Assert.Equal(2, a);
      Assert.Equal(2, b);
      Assert.Equal(3, c);
      Assert.Equal(3, d);
      Assert.Equal(3, e);
    }
}
```

In the file `IRandomGenerator.cs`:

```csharp
public interface IRandomGenerator
{
  int Number(int max = 100);
  int Number(int min, int max);
}
```

In the file `MockRandomGenerator.cs`:

```csharp
public class MockRandomGenerator : IRandomGenerator
{
  private readonly List<int> _numbers;
  private List<int>.Enumerator _numbersEnumerator;

  public MockRandomGenerator(List<int> numbers = null)
  {
    _numbers = numbers ?? new List<int>();
    _numbersEnumerator = _numbers.GetEnumerator();
  }

  public int Number(int min, int max)
  {
    var result = Number(max);
```

```
      return result < min ? min : result;
    }

    public int Number(int max = 100)
    {
      _numbersEnumerator.MoveNext();
      var result = _numbersEnumerator.Current;

      return result > max ? max : result;
    }

    public void SetNumbers(params int[] args)
    {
      _numbers.AddRange(args);
      _numbersEnumerator = _numbers.GetEnumerator();
    }
}
```

Now, to create the production `RandomGenerator` class and inject it into our application.

In the file `RandomGenerator.cs`:

```
public class RandomGenerator : IRandomGenerator
{
  private readonly Random _rand;

  public RandomGenerator()
  {
    _rand = new Random();
  }

  public int Number(int max = 100)
  {
    return _rand.Next(0, max);
  }

  public int Number(int min, int max)
  {
    return _rand.Next(min, max);
  }
}
```

In the file `Program.cs`:

```
class Program
{
  static void Main(string[] args)
  {
    var rand = new RandomGenerator();
    var inout = new ConsoleInputOutput();
    var game = new Mastermind(inout, rand);

    var password = args.Length > 0 ? args[0] : null;
    game.Play(password);

    inout.WriteLine("Press any key to quit.");
    inout.Read();
  }
}
```

In the file `Mastermind.cs`:

```
public class Mastermind
{
  private readonly IInputOutput _inout;
  private readonly IRandomGenerator _random;

  private int _tries;
```

```csharp
public Mastermind(IInputOutput inout, IRandomGenerator random)
{
  _inout = inout;
  _random = random;
}

public void Play(string password = null)
{
  password = password ?? CreateRandomPassword();
  var correctPositions = 0;

  while (correctPositions != 4)
  {
    correctPositions = GuessPasswordAndCheck(password);
  }

  _inout.WriteLine("Congratulations you guessed the password in " +
  _tries + " tries.");
}

private int GuessPasswordAndCheck(string password)
{
  var guess = Guess();
  return Check(guess, password);
}

private int Check(string guess, string password)
{
  var checkResult = "";

  for (var x = 0; x < 4; x++)
  {
    if (guess[x] == password[x])
    {
      checkResult += "+";
    }
    else if (password.Contains(guess[x]))
    {
      checkResult += "-";
    }
  }

  _inout.WriteLine(checkResult);
  return checkResult.Count(c => c == '+');
}

private string Guess()
{
  _tries = _tries + 1;

  _inout.Write("Take a guess: ");
  var guess = _inout.ReadLine();

  if (guess.Length == 4)
  {
    return guess.ToUpper();
  }

  // Password guess was wrong size - Error Message
  _inout.WriteLine("Password length is 4.");
  return Guess();
}

private string CreateRandomPassword()
{
  var password = new[] { 'A', 'A', 'A', 'A' };

  var j = 0;

  password_loop:
  password[j] = (char)(_random.Number(6) + 65);
  j = j + 1;
```

```
      if (j < 4) goto password_loop;

      return new string(password);
   }
}
```

And lastly, let's modify the gold standard test to use random password generation.

In the file GoldStandardTests.cs:

```
public class GoldStandardTests
{
  [Fact]
  public void StandardTestRun()
  {
    // Arrange
    var inout = new MockInputOutput();
    var rand = new MockRandomGenerator();
    var game = new Mastermind(inout, rand);

    // Arrange - Inputs
    rand.SetNumbers(0, 1, 2, 5);
    inout.InFeed.Enqueue("AAA");
    inout.InFeed.Enqueue("AAAA");
    inout.InFeed.Enqueue("ABBB");
    inout.InFeed.Enqueue("ABCC");
    inout.InFeed.Enqueue("ABCD");
    inout.InFeed.Enqueue("ABCF");
    inout.InFeed.Enqueue(" ");

    // Arrange - Outputs
    var expectedOutputs = new Queue<string>();
    expectedOutputs.Enqueue("Take a guess: ");
    expectedOutputs.Enqueue("Password length is 4." +
    Environment.NewLine);
    expectedOutputs.Enqueue("Take a guess: ");
    expectedOutputs.Enqueue("+---" + Environment.NewLine);
    expectedOutputs.Enqueue("Take a guess: ");
    expectedOutputs.Enqueue("++--" + Environment.NewLine);
    expectedOutputs.Enqueue("Take a guess: ");
    expectedOutputs.Enqueue("+++-" + Environment.NewLine);
    expectedOutputs.Enqueue("Take a guess: ");
    expectedOutputs.Enqueue("+++" + Environment.NewLine);
    expectedOutputs.Enqueue("Take a guess: ");
    expectedOutputs.Enqueue("++++" + Environment.NewLine);
    expectedOutputs.Enqueue("Congratulations you guessed the password
    in 6 tries." + Environment.NewLine);

    // Act
    game.Play();

    // Assert
    inout.OutFeed.ForEach(text =>
    {
      Assert.Equal(expectedOutputs.Dequeue(), text);
    });
  }
}
```

Now we are ready to refactor the password generation method and extend it to provide us with the requested change. First, there is a looping structure that is not core to the language. Let's focus in on the CreateRandomPassword method and fix the looping structure:

```
private string CreateRandomPassword()
{
  var password = new[] { 'A', 'A', 'A', 'A' };

  for(var j = 0; j < 4; j++)
```

```
  {
    password[j] = (char)(_random.Number(6) + 65);
  }

  return new string(password);
}
```

Next, for fun, let's see if we can generalize and compress this loop, since we have a very similar loop in the `check` method. While not necessary, this is fun example of reducing duplication of code. Here is what that refactoring looks like:

```
private int Check(string guess, string password)
{
  var checkResult = "";

  Times(4, x => {
    if (guess[x] == password[x])
    {
      checkResult += "+";
    }
    else if (password.Contains(guess[x]))
    {
      checkResult += "-";
    }
  });

  _inout.WriteLine(checkResult);
  return checkResult.Count(c => c == '+');
}

private string CreateRandomPassword()
{
  var password = new[] { 'A', 'A', 'A', 'A' };

  Times(4, x => password[x] = (char)(_random.Number(6) + 65));

  return new string(password);
}

private static void Times(int count, Action<int> act)
{
  for (var index = 0; index < count; index++)
  {
    act(index);
  }
}
```

Now let's do one more refactoring before we extend the application. Looking at how the characters are generated, it is not very obvious what is going on. Instead, we would like the code to be as straightforward as possible. There is no reason that the random generator class can't just directly return letters, so let's add that functionality.

In the file `RandomLetterTests.cs`:

```
public class RandomLetterTests
{
  private readonly MockRandomGenerator _rand;

  public RandomLetterTests()
  {
    _rand = new MockRandomGenerator();
  }

  [Fact]
  public void ItExists()
  {
    _rand.Letter();
```

```csharp
    }

    [Fact]
    public void ItReturnsDefaultValue()
    {
      // Act
      var result = _rand.Letter();

      // Assert
      Assert.Equal('A', result);
    }

    [Fact]
    public void ItCanReturnPredeterminedLetters()
    {
      // Arrange
      _rand.SetLetters('A', 'B', 'C', 'D', 'E');

      // Act
      var a = _rand.Letter();
      var b = _rand.Letter();
      var c = _rand.Letter();
      var d = _rand.Letter();
      var e = _rand.Letter();

      // Assert
      Assert.Equal('A', a);
      Assert.Equal('B', b);
      Assert.Equal('C', c);
      Assert.Equal('D', d);
      Assert.Equal('E', e);
    }

    [Fact]
    public void ItCanHaveAMaxRange()
    {
      // Arrange
      const char maxRange = 'C';
      _rand.SetLetters('A', 'B', 'C', 'D', 'E');

      // Act
      var a = _rand.Letter(maxRange);
      var b = _rand.Letter(maxRange);
      var c = _rand.Letter(maxRange);
      var d = _rand.Letter(maxRange);
      var e = _rand.Letter(maxRange);

      // Arrange
      Assert.Equal('A', a);
      Assert.Equal('B', b);
      Assert.Equal('C', c);
      Assert.Equal('C', d);
      Assert.Equal('C', e);
    }

    [Fact]
    public void ItCanHaveAMinMaxRange()
    {
      // Arrange
      const char minRange = 'B';
      const char maxRange = 'C';
      _rand.SetLetters('A', 'B', 'C', 'D', 'E');

      // Act
      var a = _rand.Letter(minRange, maxRange);
      var b = _rand.Letter(minRange, maxRange);
      var c = _rand.Letter(minRange, maxRange);
      var d = _rand.Letter(minRange, maxRange);
      var e = _rand.Letter(minRange, maxRange);

      // Arrange
      Assert.Equal('B', a);
      Assert.Equal('B', b);
```

```
      Assert.Equal('C', c);
      Assert.Equal('C', d);
      Assert.Equal('C', e);
    }
  }
}
```

In the file `MockRandomGenerator.cs`:

```
public class MockRandomGenerator : IRandomGenerator
{
  private readonly List<int> _numbers;
  private List<int>.Enumerator _numbersEnumerator;

  private readonly List<char> _letters;
  private List<char>.Enumerator _lettersEnumerator;

  private const char NullChar = '\0';

  public MockRandomGenerator(List<int> numbers = null, List<char>
  letters = null)
  {
    _numbers = numbers ?? new List<int>();
    _numbersEnumerator = _numbers.GetEnumerator();

    _letters = letters ?? new List<char>();
    _lettersEnumerator = _letters.GetEnumerator();
  }

  public int Number(int min, int max)
  {
    var result = Number(max);

    return result < min ? min : result;
  }

  public int Number(int max = 100)
  {
    _numbersEnumerator.MoveNext();
    var result = _numbersEnumerator.Current;

    return result > max ? max : result;
  }

  public void SetNumbers(params int[] args)
  {
    _numbers.AddRange(args);
    _numbersEnumerator = _numbers.GetEnumerator();
  }

  public int Letter(char min, char max)
  {
    var result = Letter(max);

    return result < min ? min : result;
  }

  public char Letter(char max = 'Z')
  {
    _lettersEnumerator.MoveNext();
    var result = _lettersEnumerator.Current;
    result = result == NullChar ? 'A' : result;

    return result > max ? max : result;
  }

  public void SetLetters(params char[] args)
  {
    _letters.AddRange(args);
    _lettersEnumerator = _letters.GetEnumerator();
  }
}
```

In the file `IRandomGenerator.cs`:

```
public interface IRandomGenerator
{
  int Number(int max = 100);
  int Number(int min, int max);
  char Letter(char max = 'Z');
  char Letter(char min, char max);
}
```

In the file `RandomGenerator.cs`:

```
public class RandomGenerator : IRandomGenerator
{
  private readonly Random _rand;

  public RandomGenerator()
  {
    _rand = new Random();
  }

  public int Number(int max = 100)
  {
    return Number(0, max);
  }

  public int Number(int min, int max)
  {
    return _rand.Next(min, max);
  }

  public char Letter(char max = 'Z')
  {
    return Letter('A', max);
  }

  public char Letter(char min, char max)
  {
    return (char) _rand.Next(min, max);
  }
}
```

In the file `Mastermind.cs`:

```
private string CreateRandomPassword()
{
  var password = new[] { 'A', 'A', 'A', 'A' };

  Times(4, x => password[x] = _random.Letter('F'));

  return new string(password);
}
```

In the file `GoldStandardTests.cs`:

```
// Arrange - Inputs
rand.SetLetters('A', 'B', 'C', 'F');
```

That is the final refactoring for this exercise. We only have one thing to do, and that is to extend the application to generate passwords using the full range of the English alphabet. Because of the effort we put into testing and refactoring, this is now a trivial matter, and, in fact, only requires the removal of three characters in the `Mastermind` class.

In the file `Mastermind.cs`:

```
private string CreateRandomPassword()
{
  var password = new[] { 'A', 'A', 'A', 'A' };

  Times(4, x => password[x] = _random.Letter());

  return new string(password);
}
```

Now a more complicated password, consisting of the full range of the alphabet, is created. This causes a much more difficult password, and a game with output similar to the following:

```
Take a guess: AAAA
Take a guess: BBBB
Take a guess: CCCC
---+
Take a guess: DDDC
+
Take a guess: EEEC
+
Take a guess: FFFC
+
Take a guess: GGGC
+
Take a guess: HHHC
+
Take a guess: IIIC
+
Take a guess: JJJC
+
Take a guess: KKKC
+
Take a guess: LLLC
+
Take a guess: mmmc
+
Take a guess: nnnc
+
Take a guess: oooc
+--+
Take a guess: oppc
++-+
Take a guess: opqc
+++
Take a guess: oprc
+++
Take a guess: opsc
+++
Take a guess: optc
+++
Take a guess: opuc
+++
Take a guess: opvc
+++
Take a guess: opwc
++++
Congratulations you guessed the password in 23 tries.

Press any key to quit.
```

Summary

You now have a well-written example, covered by tests. The effort involved can be daunting, but for anything more than a trivial application, it can be well worth the effort.

In Chapter 14, *A Better Foot Forward*, we'll summarize what we've learned as well as give you some pointers on how to rejoin the world as a TDD expert.

A Better Foot Forward

You've made it to the final chapter of *Practical Test-Driven Development using C# 7*. We thank you. But, your journey as a **Test-Driven Development** (**TDD**) practitioner is just beginning. Soon it will be time for you to rejoin the world as a TDD expert.

In this chapter, we'll summarize the main topics from previous chapters and give you some pointers to help you continue on this voyage. In this chapter, we will gain an understanding of:

- Why TDD is important
- Growing applications through tests
- Introducing TDD to your team
- Rejoining the world as a TDD expert

What we've covered

You're likely not yet an expert. That's okay. There will be times when you may become stuck or doubt the benefits of TDD. Fret not. Reading this book has been but a step on the journey to becoming a TDD master. The road is long but well worth the time and effort devoted to this journey. You are a professional, a craftsman devoted to your trade.

By now, you should feel confident setting up your development environment. You can configure your IDE of choice to run your suite of unit tests. You should be comfortable with choosing a test runner and the specific nuances and features involved in that choice. And, of course, you know how to assemble a comprehensive set of unit tests.

You can grow an application guided by tests. Refactoring should now be a breeze as you have the confidence to move code around without introducing breaking changes. You can demonstrate correctness to application stakeholders and you have guarded against regression bugs with the confidence provided by your tests.

The world in the TDD space makes perfect sense. You'll wonder how you ever lived without your new-found knowledge. Let's take a moment and review what we've learned.

Moving forward

So, where do you go from here? Hopefully, you're as excited about TDD as you were when you first successfully compiled a software application. Each successive passing test is proof of your hard work and understanding of the problem at hand. Celebrate each small victory for the accomplishment that it is. Validate your understanding along the way by introducing more and more functionality to the system through your tests.

As you continue in your career, it is up to you to choose to operate in a TDD fashion. Adherence to the philosophy and how well you maintain the TDD mindset is completely up to you. Don't be discouraged if your boss isn't familiar with TDD. You don't need permission.

Continue to grow applications through tests. If others ask about TDD, share your knowledge and enthusiasm. Introduce the practice to your team, but do not force the practice on anyone who is not ready.

TDD is a personal practice

First and foremost, TDD is a personal practice. It should not be a line item on anyone's budget. TDD is the way in which high-quality software is developed by those who care deeply about their craft.

If you suspect you might get pushback from your team, managers, or project sponsors then there is no need to involve them in the decision to develop software in this way.

It is often better that the development team is made aware that there are tests in the system so as to avoid breaking them, but it need not be necessary to seek permission.

You don't need anyone's permission to do good work.

You don't need permission

If you introduce TDD on a project that was not originally started that way, you may be asked who gave you permission to do so. You don't need anyone's permission to do good work. TDD is a personal practice to ensure you deliver high-quality software. You shouldn't need anyone's permission to do so.

Take pride in your work and do the best that you know how. As you grow and become more comfortable with TDD, it will likely become your default method of software delivery.

Grow applications through tests

It's easy to fall into the trap of *Big Design Upfront*. These design sessions are important, but the artifacts of these meetings are the exploratory ideas they generate, not a note-for-note plan of attack. Applications should be grown organically, guided by the tests written by you, the developer.

When developing a greenfield application, it's much easier to get started with TDD. Designing with testing in mind from the beginning is much easier than trying to retrofit tests at a later date. Your applications will benefit greatly when developed from the beginning with testing in mind. Guided by tests, your software will be simpler and much easier to grow and maintain.

Enhancements, assuming the needs and expectations are clearly defined, are effortlessly added to a system through tests. If a user story has been well defined, then the requirements translate easily to a series of new tests. Subsequently, new production code can easily be added to the system by making the new tests pass. Fear of introducing new defects or changing existing behavior should be minimal, assuming an existing comprehensive suite of unit tests.

Defects can be addressed as they are discovered. Simply write one or more tests to define the expected behavior and modify the production code as you go. More defects may be uncovered, or existing behavior, and by extension existing tests may need to change as a result. Don't worry, the suite of tests is there to guard against bugs and to give you, the developer, a sense of security.

Introducing TDD to your team

At this point, you are probably quite excited at sharing the wonderful world of TDD with your organization and fellow team members. Be aware that others may not have the same enthusiasm. Writing unit tests may be a scary proposition to someone that has never tried it before. There may be some negative connotations associated with the idea for some on your team.

As developers, we're paid to be the experts. We're expected to have the answers. When something new and unknown is introduced, it can be an anxiety-inducing experience. Work towards reducing the anxiety your team might feel when learning something new. Remember how it felt when you heard the term TDD and didn't know where to begin. Think about how you felt before picking up this book.

There are good ways and not so good ways to introduce TDD to your team. If you truly want your team to adopt the practice, then think about ways in which to get them excited about the prospect.

Don't force TDD on anyone

There have been countless tales of woe associated with introducing TDD to a team. An over enthusiastic member tries to cram the merits of TDD down the throats of the rest of the team. Project leaders may balk and try to ban the process altogether.

If a team does decide to adopt the practice as a whole, be sure that each member of the team has a say. Everyone's voice is important. If a team member is not on board, team mates may recoil, and may even leave the project or the company.

Gamification of TDD

A great way to expose a new team to TDD is by making a game out of the process. Introduce the subject slowly and get people excited about the prospect of learning a new skill. Create a friendly rivalry or set of challenges for the team.

Lunch and learn sessions can be a great way to introduce the subject. There are countless video tutorials on YouTube and sites such as Pluralsight that can be a great introduction and social sharing event for your team.

Code Katas are an excellent way to expose someone to TDD. Small, 20-minute exercises are simple enough to get someone comfortable with the basic premise. Slowly introduce more complexity and different challenges as the members get more comfortable with the exercise.

Code Challenges and/or homework may prove to be a good way to get your team involved. You should have a feel for who works best under what scenarios. Some may prefer homework and working on their own, while others may be more responsive to individual challenges.

Showing your team the benefits

Your best bet is to introduce testing slowly. If you're working to resolve a defect, look to wrap the existing method or function with a test to verify the defect. Correct the code to allow the new tests to pass. Share your results with your teammates. Explain how easy the process was and how much time and effort might have been saved as a result. Do not pressure, just inform.

New features to existing applications are a great time to explore TDD with your team. If you can show your co-workers how new features can be developed using TDD in an existing application, it may prove useful. It may be easier to introduce a foreign concept, such as TDD, within an existing application that developers are comfortable with. With the comfort of the known application, the unknown is kept to a minimum and the focus can be on TDD.

If you are beginning a new project it may be beneficial to take ownership of part of the application in order to develop this part using TDD. If at least part of the application is developed with TDD, then this can be used to demonstrate the basics to others on the team. Using this piece of the application as an example, you may convince others to explore spreading TDD into the rest of the application.

Be sure to track progress along the way. You may find that individuals or parts of applications are better suited for TDD. If anyone is struggling, it may be difficult for them to admit their troubles. Keep an eye out and lend a hand where you can.

Review the results

When introducing TDD to a team, be prepared to address any potential issues often associated with developers new to TDD. Useless tests will be written that provide no value. Look for these and other issues within the test suite.

If your team does not currently utilize pull-requests or code reviews, now would be a perfect time to introduce the practice. Make a habit of reviewing tests that are written. This will go a long way to spotting any potential trouble spots. This also has the added benefit of learning more about parts of the system with which you may not be familiar.

If you're working on an existing project, or any new feature over which you have control, begin by developing this feature with TDD. If you have a code review process in place, be sure that the reviewers are made aware and demonstrate the passing tests to them.

Rejoining the world as a TDD expert

It is now time to rejoin the world as a TDD expert. Don't worry if you don't quite feel like an expert yet. You likely have more knowledge than at least one co-worker or peer. To them, you are the expert with knowledge to share. Go forth and share that knowledge. But remember, there is always more to learn.

Seek a mentor

It may be beneficial to seek out a mentor or mentors. There is likely someone in your community with TDD knowledge who is willing to speak with you. They could be employees of your company or members of the tech community in your city. Seek them out and offer to buy them lunch or meet for coffee. You may find that they're thrilled to find someone else willing and eager to discuss TDD as a practice.

User groups and meetups are a great place to network. Look for meetings in your area that you could attend. TDD has become a hot topic in recent years and you're likely to find a meeting close by. The programming language may be different from your daily routine (for example, Java vs C#), but the overall premise will likely be beneficial. Besides, exposure to new and different languages helps you explore paradigms with which you might not be familiar.

Twitter is another great resource these days. The amount of technology professionals active on Twitter and other social media these days is staggering. Never before has it been possible to strike up a conversation with the giants of our industry. Don't be afraid to ping technologists on Twitter and the like, and express interest in gaining knowledge. You may be surprised at the response you receive.

Becoming a mentor

In the same vein as seeking a mentor, you may also consider becoming a mentor. The same venues apply to becoming a mentor. You'll learn more by teaching. Share your knowledge with others and learn just as much from those you teach.

User groups and meetups are always interested in having someone speak. Consider putting together a presentation and sharing your interest in TDD. This could be anything from a 5- to 15-minute lightning talk, an hour-long presentation, or a full-day workshop.

Practice, practice, practice

There's a reason doctors and lawyers define what they do as *practice*. Like these professions, the computer programmer is engaged fully in the practice of software development. It is a practice that requires practice.

Develop a habit of spending the first 20 minutes of each day working on a different Code Kata. Solve a new puzzle using TDD. Work on a problem you've solved before using a new or different technique. Try a different approach at developing a solution. Spend time on your craft and train yourself to look for alternate solutions that are testable and verifiable.

Review your work. Ensure that you are truly driving your application development through tests. Verify that you are actually testing your application and not simply going through the motions.

Most of all, have fun!

Summary

Our journey has come to an end, but fear not. You now are well equipped to enter the world as a TDD master.

You not only understand how to develop with a TDD mindset, you also know why TDD is so important to develop testable, extensible, and maintainable software applications. Your IDE is set up to test C# and/or JavaScript applications and you have a continuous feedback loop on the quality of your software.

You understand the importance of defining and testing the boundaries of an application and the benefits of abstracting away third-party code (including the .NET Framework). Spies, mocks, and fakes, and how best to employ them are now well understood.

Approaching a green-field application with TDD is mind should now be almost trivial. Take the broader problem of the overall application and break it into meaningful chunks that can be developed independently. You have learned different approaches to developing an application such as: front to back, back to front, and inside out. Choose what is most appropriate.

Taking requirements and assembled user stories and turning them into working software using TDD should be a breeze! Utilize all the skills you've assembled to test the boundaries, testing small, individual units.

Abstract away third-party libraries, including the .NET Framework. Remove dependencies on things such as `DateTime` and Entity Framework. You've learned ways to decouple your application from specific implementations to allow your application to be testable, but also much more flexible and easy to modify in the future.

What happens when the requirements change? What happens if a bug is discovered? No problem, change a test or write a new one to cover the new requirement or to defend against the discovered bug. Then, write some new code or change some existing code to make all of our new/modified tests pass. If you do everything correctly you should feel safe to make these changes as our existing test suite will prevent you from introducing new bugs.

There are a lot of applications out there without sufficient (any?) test coverage. Even fewer were written test-first. You're now aware of some of the major problems with legacy applications that weren't written with testability in mind and know how to best to rectify this.

You know how to safely modify a legacy application that wasn't written with testing in mind, and know how to add tests to minimize the potential for introducing new bugs when modifying existing code.

Remember, TDD is a personal choice. You don't need anyone's permission to do good work. Now, rejoin the world as a TDD expert!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

Test-Driven iOS Development with Swift 3
Dr. Dominik Hauser

ISBN: 978-1-78712-907-8

- Implement TDD in Swift application development
- Find bugs before you enter the code using the TDD approach
- Use TDD to build models, view controllers, and views
- Test network code with asynchronous tests and stubs
- Write code that is a joy to read and to maintain
- Develop functional tests to ensure the app works as planned
- Employ continuous integration to make testing and deployment easier
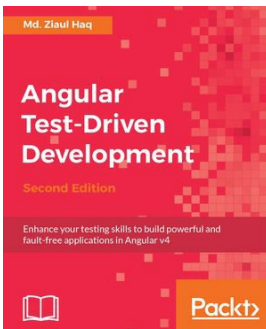
Angular Test-Driven Development - Second Edition
Md. Ziaul Haq

ISBN: 978-1-78646-547-4

- Get a clear overview of TDD in the context of JavaScript with a brief look at testing techniques, tools, and frameworks
- Get an overview of Karma and create test suites for an Angular application
- Install and configure Protractor for Angular and explore a few important Protractor APIs
- Understand automated testing and implement headless automated tests with Karma

- Implement testing techniques with mocks, broadcast events, and asynchronous behavior
- Integrate REST-based services and APIs into an application to extract data
- Automate Karma unit tests with Travis CI

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Table of Contents