Build Your Own Test Framework

A Practical Guide to Writing Better Automated Tests

Daniel Irvine

Foreword by Maaret Pyhäjärvi

apress®

Build Your Own Test Framework

A Practical Guide to Writing Better Automated Tests

Daniel Irvine

Foreword by Maaret Pyhäjärvi

Apress®

Build Your Own Test Framework: A Practical Guide to Writing Better Automated Tests

Daniel Irvine
London, UK

Table of Contents

About the Author

Daniel Irvine is a freelance software developer and technical coach. He specializes in simplifying software codebases and improving the technical confidence of dev teams. His first exposure to automated testing was in 2005, when he was tasked with writing Ruby test suites for a million-line C++ application. Since then, he's been an advocate for developer-led testing practices. He is the author of *Mastering React Test-Driven Development*, now in its second edition.

He can be contacted via his website at `www.danielirvine.com`.

About the Technical Reviewer

Sourabh Mishra is an entrepreneur, developer, speaker, author, corporate trainer, and animator. He is a Microsoft guy; he is very passionate about Microsoft technologies and a true .Net Warrior. Sourabh started his career when he was just 15 years old. He's loved computers from childhood. His programming experience includes C/C++, ASP.Net, C#, VB.net, WCF, SQL Server, Entity Framework, MVC, Web API, Azure, jQuery, Highcharts, and Angular. Sourabh has been awarded a Most Valuable Professional (MVP) status. He has the zeal to learn new technologies and shares his knowledge on several online community forums.

He is the author of *Practical Highcharts with Angular* (Apress, 2020), which talks about how you can develop stunning and interactive dashboards using Highcharts with Angular.

He is the founder of "IECE Digital" and "Sourabh Mishra Notes," an online knowledge-sharing platform where one can learn new technologies very easily and comfortably.

He can be reached via the following platforms:

- YouTube: sourabhmishranotes

- Twitter: sourabh_mishra1

- Facebook: facebook.com/sourabhmishranotes

- Instagram: sourabhmishranotes

- Email: sourabh_mishra1@hotmail.com

You can find his books on Amazon via www.amazon.com/stores/author/B084DMG1WG.

Acknowledgments

I would like to thank all the readers of the original Leanpub version of the book who have made this new edition possible.

The whole team at Apress have been wonderful and made the process an absolute breeze. Particular thanks go to Divya Modi, the lead editor, and Shobana Srinivasan, the project coordinator, both of whom were extremely patient with me as I juggled tasks.

Finally, I am indebted to all of my past and present clients, who have made my own journey to expertise possible.

Foreword

My first thought on *Build Your Own Test Framework* was curious: Why would I want to? The experience I come from is that there's plenty of test frameworks to go around, and endless conversations on which of those is worth choosing, even more so among those working in the software testing space. I found my answer to this question reading this book.

Moving from choosing a test framework to building a test framework showed me a great way of leveling learning in the tests space. It guides focus on what tests are for, what features are useful for tests, what features are generally available, and how we might build them. Daniel Irvine does a brilliant job at leveling this learning for the reader. Reinventing a wheel, you learn a lot about wheels. This is exactly what we need.

Breaking a slightly abstract tool like a test framework into features and walking us through design and implementation helps us not only build a test framework but a foundation from which we can contribute on other test frameworks and understand our own tests better.

Work through the chapters and code along and I'm sure you will enjoy features such as `it.behavesLike` to an extent you start missing the concept in frameworks you may have run into and are inspired to extend your own framework—and build your own for practice if not production use.

<div align="right">

Maaret Pyhäjärvi
Principal Test Engineer at Vaisala Oyj

</div>

Introduction

This book is a follow-along practical exercise in building an automated unit test framework, written in the JavaScript language and running on the Node platform. You can think of it as a less functional replacement for well-known packages like Jest,[1] Mocha,[2] Jasmine,[3] and Vitest.[4]

The framework is called `concise-test,` and it takes the form of an NPM package that you'll start building in Chapter 1. The book also makes use of a package named `todo-example,` which is a sample application that will make use of `concise-test`, as a package dependency.

By the end of Chapter 1, you'll be able to run the command `npm test` in your `todo-example` project directory and have your test runner execute the application test scripts.

Do I Need to Know JavaScript?

You'll need to know modern JavaScript syntax to use this book, including arrow functions,[5] destructuring assignments,[6] and rest parameters.[7] It also uses ECMAScript Modules[8] for importing and exporting values.

---

[1] https://jestjs.io

[2] https://mochajs.org

[3] https://jasmine.github.io

[4] https://vitest.dev

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

[6] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

[7] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters

[8] https://nodejs.org/api/esm.html#modules-ecmascript-modules

That being said, don't stop reading if you don't know any JavaScript; as long as you have experience in at least one programming language, you should be able to pick it up as we go along. If you're unsure what some of the syntax means, take a moment to stop and look it up online.

JavaScript is a wonderfully flexible language. It is dynamic and functional at its core, but also has support for objects in a rather unique way, which is empowering.

The beauty of unit testing is that it is universally applicable to all mainstream programming languages, so if you were really keen, you could rebuild all of the same code in your own favorite language. The only chapters that are directly related to JavaScript are Chapters 9 and 17.

Is This Book for You?

If you've ever written unit tests, then, yes.

If you're a novice, this book is a wealth of practical advice that will give you highly valuable skills, helping you to be a more productive developer.

If you've got plenty of automated testing experience already, the breadth of this book means that there are likely to be new ideas for you to uncover. In particular, the discussion questions will help you question your own biases and opinions.

I encourage you to get together with a group of colleagues to complete the practical exercises and work through the discussion questions. Although the book is designed to be followed in order, the chapters are self-contained enough that you can pick and choose where to stop and start.

How the Book Is Structured

There are 17 chapters in this book. Each chapter of this book follows a similar layout: beginning with some detailed theory of unit testing, continuing with a bit of follow-along coding, and ending with a set of practical exercises and discussion questions.

What we're building here is a *spike*. That means that, ironically enough, there are no tests for what we're about to build. Spikes help us quickly explore a problem space without getting too bogged down in plumbing code.

The following diagram shows the full extent of the system we're going to build.

In Part 1, "Building the Core of a Test Framework," you will build a barebones implementation of a test runner that features all the major components of an xUnit-style test runner.

In Part 2, "Constructing a Usable Framework," you extend the test framework to include ergonomic features that make the `concise-test` runner into a viable competitor to mainstream test frameworks.

In Part 3, "Extending for Power Users," we add some unique features that are borrowed from other programming environments.

In Part 4, "Test Doubles and Module Mocks," we end the book with a look at unit testing's most misunderstood feature, the humble test double.
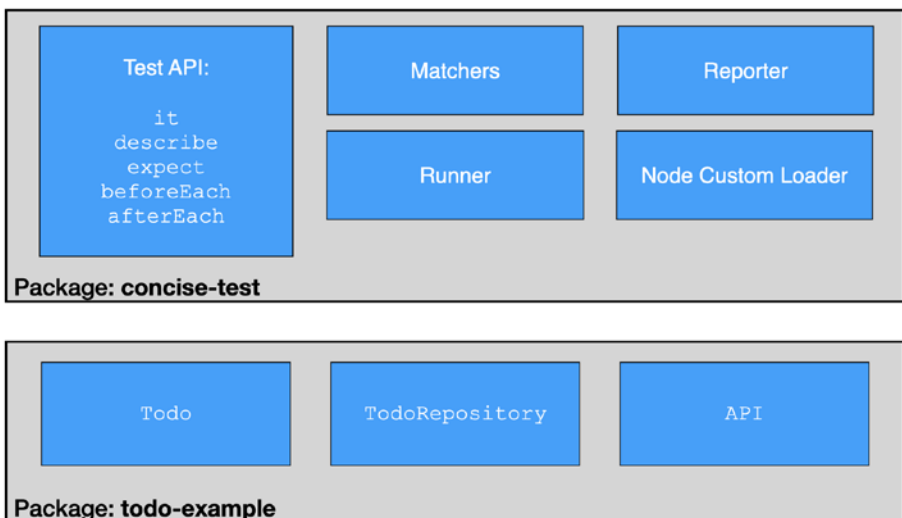
What Is an *Automated Unit Test Framework*?

Let's start with **automated**: we are talking about the opposite of manual testing. Manual testing is when you spin up your application and use it: clicking around, entering data, observing results. Manual testing is time consuming and error prone. Automated testing, on the other hand, should be very quick and give the same results every time.

Unit test is harder to define. Very often people think of them as tests for specific classes or objects within your application code, which are the "unit." But this isn't a rule. It's just a tendency that unit tests have. And each unit test *tends* to be small and exercise just a tiny sliver of functionality. And we *tend* to end up with a whole lot of unit tests that together exercise the entire system.

For me, the key defining point about unit tests is that they do not instrument any behavior *external to the application process*, like performing or relying on network requests. For many developers that also means file system access or database access. The benefit to avoiding external behavior is that it keeps the tests extremely fast and keeps tests behaving consistently across test runs.

Finally, a **framework** means that it's more than a library. In this case, a test framework consists of two things: the test runner and the test API. To explain what those are, let's look specifically at what we'll build in this book.

What You'll Need to Get Started

You'll need to install a recent version of Node, instructions for which can be found at `https://nodejs.org`. You'll need at least version 18.12.

You'll need to have Git installed to be able to access the source code repository. If you're new to Git, a good place to start is the website

https://github.com, which is also where the source code for this book is stored.

You will also need basic command-line knowledge, like how to navigate between directories with the cd command and how to use the git and npm commands.

Working with the Source Code Repository

You can find the code repository at the following URL:

github.com/Apress/Build-your-Own-Test-Framework-by-Daniel-Irvine

You will find a directory for each chapter, from Chapter01 to Chapter17. Within each of these directories, there are two subdirectories:

- A Start directory, which is the starting point for each chapter, which you can use if you want to follow along with the text in the book

- An Exercises directory, which is the end of each chapter text, but before any of the exercises have been completed

In each of these, you'll find two further subdirectories:

- A concise-test directory, which stores the concise-test package

- A todo-example directory, which stores the todo-example package

Initializing Projects for Each Chapter

Every time you switch into a new chapter location, you will need to *link* the two packages together. This is described in detail in Chapter 1,

but the essence of it is the following commands that you'd enter on the command line:

```
cd Chapter01/Start      # choose your starting point here
cd concise-test
npm link
cd ../todo-example
npm link concise-test
```

That will correctly set up the `todo-example` package dependency so that when you type the `npm test` command, you are executing code from its sibling `concise-test` directory.

Simultaneously Working in Two Packages

Because this book involves two dependent packages, you should take a moment to think about your development environment.

Whatever editor or IDE you choose, make sure you can easily open files across both project directories and have easy access to a command shell so that you can enter the `npm test` command (you'll be running that one a whole lot).

PART I

Building the Core of a Test Framework

In this part you will build a barebones implementation of a test runner that features all the major components of an xUnit-style test runner.

In Chapter 1, "Creating a Barebones Test Runner," you will create a new NPM package for `concise-test`, including an entrypoint for the test runner.

In Chapter 2, "Building a Function to Represent a Test Case," we'll implement the `it` function for defining tests and start printing out some useful test run information on-screen.

In Chapter 3, "Grouping Tests," we'll add support for grouping tests with the `describe` function, and we'll continue to build out test reporting on-screen.

In Chapter 4, "Promoting Conciseness with Shared Setup and Teardown," we continue our quest for concise testing facilities with the addition of `beforeEach` and `afterEach` functions.

In Chapter 5, "Improving Legibility with Expectations and Matchers," we finish off the core of our API by building an abstraction over `throw new Error`.

CHAPTER 1

Creating a Barebones Test Runner

In this chapter, you'll learn all about test runners: what they are, why they exist, and how to build one for the Node runtime environment.

At the highest level, the test runner is the piece of software that does exactly what it says: it *runs* (or executes) your *tests* (also known as *test cases*). The input to your test runner will be your test files, and its output will be a test report showing how your test cases performed.

By the end of this chapter, you'll have learned the following:

- How the practice of unit testing came about

- How to create a basic Node library

- How to build a very basic test runner that can be invoked by using the `npm test` command

Before we get on to test runners, we first need to understand the concept of an *application entrypoint*.

What Is an Entrypoint?

Think about how your web browser loads a typical web-based JavaScript application. You point the browser at a specific URL (like `www.example.com`), and that pulls down an HTML page that the browser parses and loads.

This HTML page might contain a `script` HTML tag that references another URL: this URL points to a JavaScript source file. And this source file is special because it marks your application *entrypoint*: it is the first bit of code that your browser will fetch, parse, load, and execute.

What sets apart the entrypoint from any other JavaScript file? Well, your entrypoint source file (or *module*) may import and load other source files. This network of files represents your entire application, but only one of these files is the entrypoint.

The browser in this example represents the *runtime environment*. The JavaScript source file is your *application*, and the *entrypoint* is very simply that same file.

Every computer program has an entrypoint, and every computer program operates within a runtime environment, be it the browser or your terminal.

It's the job of the runtime environment to load and execute your code, as visualized in Figure 1-1.
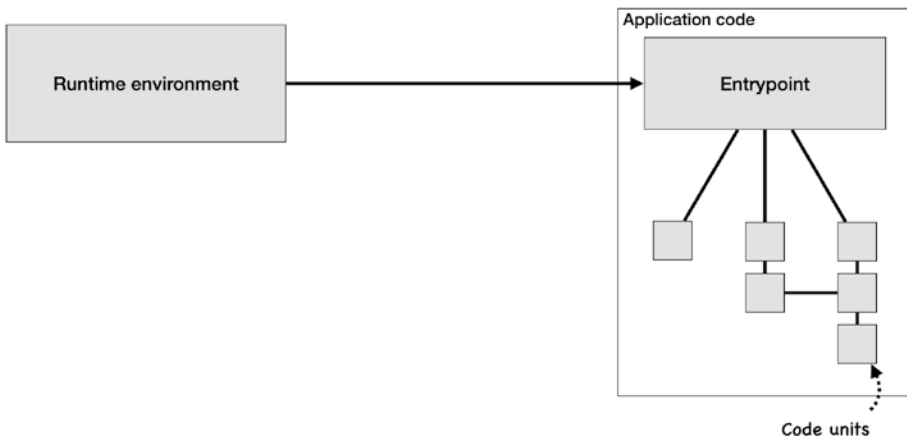
Figure 1-1. *The relationship between a runtime environment, an application, and its component parts*

With that context, think about how you might write some *automated tests* for any given computer program. By automated test I mean one that invokes your code, checks the result and expected value match, and reports that information to you. The only action you need to take is the one to start the test.

The First Stage of Automated Test Enlightenment

If you've been programming for even the shortest time, you'll understand that *debugging* your software is a time-consuming and tiring process. You change your application code, try it out, find a bug, attempt to fix it, try it out again, and get ever more frustrated as you go round in circles trying to get it all working.

The first stage of automated test enlightenment is to realize that you should find a way to get the computer to do the repeated manual debugging work for you. It will save you time and energy.

For the simplest of programs, such as a command-line tool that takes a bunch of inputs and prints out some computed output, you can simply run the program.

Take, for example, the `touch` utility, which updates the modification timestamp of a file on disk, given its file path. If you're using a Bash-like runtime environment, you could write a Bash script that runs the utility and then verifies that the file's modification has been updated, outputting either `PASS` or `FAIL` on-screen.

You can do the same with web applications: plenty of tools exist that will drive a browser, automatically pointing and clicking according to a script.

Then you can build up a whole suite of these test cases, which can run in sequence and provide you with a test report at the end.

However, there's a problem.

Beyond the simplest of applications, it's extremely difficult to run an application from start to finish and make it behave in a consistent way. That's because it will depend on *hidden variables* that you can't specify as inputs: things like the current time, the state of the current file system, the type of web browser that the user is running, and so on.

Even the simple `touch` utility isn't immune. It behaves differently depending on whether or not the file already exists.

The Second Stage of Automated Test Enlightenment

It turns out that controlling all your test inputs is very hard. But we really want our tests to be consistent, because without consistent tests, any test result will effectively be meaningless: test failures will need to be checked in case they were caused by invalid test inputs, and test successes will *also* need to be checked just in case they are false positives.

Since the early days of structured programming, our programs have been split into functions and modules.

And the second stage of automated test enlightenment is that we can test these *units* in isolation, controlling the inputs to one unit only.

It turns out that many programming environments allow us to sidestep the application entrypoint and instead point directly at another piece of executable code of our choosing. As long as that code has an *exported name*, then we can reference it in our test cases.

This is the essence of *unit testing*: when our test runner ignores the normal application entrypoint and instead allows us to instrument the functions directly within our code.

The Inner Workings of a Unit Test Runner

The test runner, then, can be thought of doing the following:

- Loading your test suites

- Loading individual units of your codebase, avoiding the application entrypoint

- Running each test and recording the result

- Printing out a test report

Because the application isn't running in the conventional sense, the application entrypoint is effectively ignored. Its code will not be executed by the test runner, as illustrated in Figure 1-2. That's why you're often advised to keep an entrypoint as short as possible, because it won't have unit test coverage.
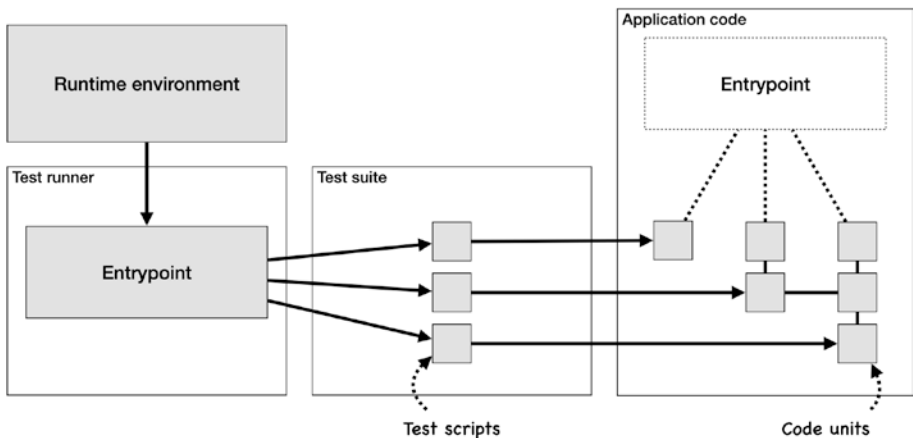
Figure 1-2. *How a test runner executes your application code*

We already mentioned one benefit to structuring our tests like this: the ability to instrument smaller units means we have better control over external inputs, which makes tests behave consistently.

But there are other benefits, like faster test runs, because you're not having to execute the entire program for every test run, and test independence, which means that a bug in one unit won't affect the tests in other units, which helps pinpoint where errors lie (instead of every single test falling over when an error is introduced).

The Third Stage of Automated Test Enlightenment

One of the first widely used unit test runners was designed and described by Kent Beck in 1994.[1] This tool introduced the concept of *Fixtures*, *Cases*, and *Checks*, all of which will feature in the `concise-test` system we're building in this book.

Since that time, many other people have brought forward even more ideas that build on top of these concepts. Many of these ideas are discussed within the pages of this book.

So the third stage of automated test enlightenment is to notice that many, many smart developers have done the hard work so you don't have to. All you need to do is read and learn about their work. We truly are standing on the shoulders of giants.

Now let's look at a sequence diagram[2] of a test runner for the Node runtime environment, shown in Figure 1-3. This depicts the `concise-test` program that we'll be building in this book.

---

[1] The Smalltalk Report, October 1994

[2] This is stylized as a Unified Modeling Language (UML) sequence diagram. UML has gone out of fashion, but I find the core concepts highly valuable.
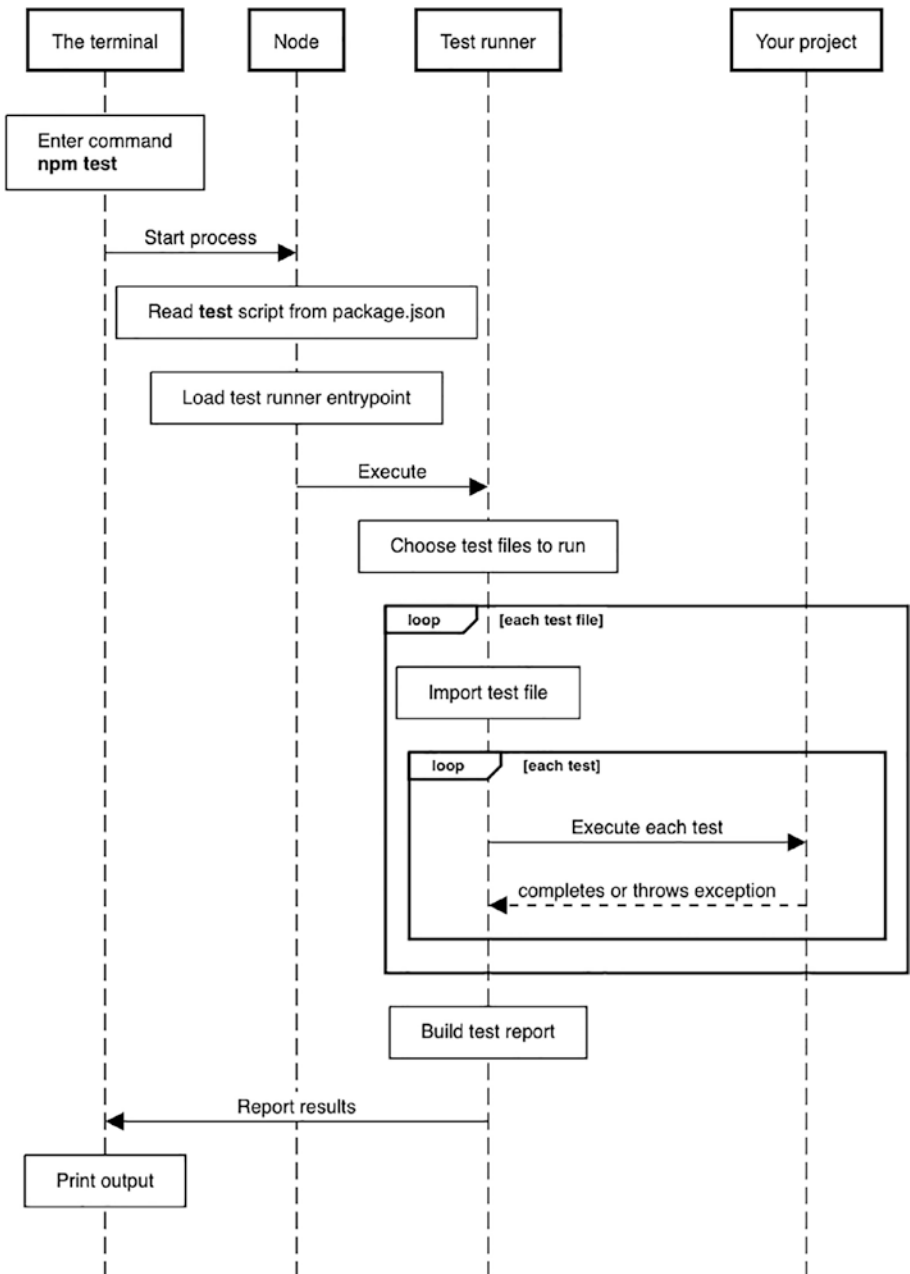
Figure 1-3. *A sequence diagram showing the operation of our test runner*

The vertical *lifelines* represent the main actors within an automated test run: the terminal (or command line) where you initiate the test run, the Node executable, the test runner code, and your own project files.

A test run starts with the user entering the `npm test` command. This could also be run by a Continuous Integration (CI) environment. Either way, it causes Node to figure out what you mean by the action `test`. In our case, we'll define that to mean the `concise-test` executable script.

Complexity lies within the two loops: discovering test files and then executing all the test cases held within them. In this first chapter, we'll ignore all of that, creating a *barebones* implementation that gets all the rest of the plumbing in order.

The rest of this chapter will cover the following:

- Creating the initial Node project for the `concise-test` test runner

- Creating the executable script that acts as the test runner entrypoint

- Verifying the test runner works by breaking a test

Time to write some code!

Cloning the Project Repository

As mentioned in the "Introduction," this book already has a pre-written sample application project ready to go. It is a very simple todo application. You can also find it online at this GitHub location:

https://github.com/Apress/Build-your-Own-Test-Framework-by-Daniel-Irvine

You should fork this repository now, via the GitHub.com website, which will allow you to save your own changes to it.

Now let's clone the repository. Open a terminal window and change to your default workspace directory (for me that's `~/work`), and then use `git clone` to pull down the repository to your local machine.

Then, rename the directory to something much shorter, like `byotf`, which will make it easier to see long file paths.

Navigate into the `Chapter01/Start` directory, which is where we'll be starting the work in this chapter. You'll see it has two subdirectories:

```
- ~/work/byotf/Chapter01/Start
  - /concise-test
  - /todo-example
```

The `concise-test` directory is empty other than a `README.md` file. In the next section, you'll create a new NPM project in this directory.

Creating an NPM Project for the Test Runner

Type the following commands:

```
cd concise-test
mkdir src
npm init -y
```

The `-y` switch to `npm init` causes it to initialize your project with default values for all its configuration settings. It will print out a message that it's written a new file named `package.json` and show you the contents of the file. Next we'll customize that.

Setting Package Configuration

We need to set a couple of extra entries in `package.json`. Open that file now and add the following properties at the bottom of the file:

```
  "type": "module",
```

```
"bin": "./src/cli.mjs",
"exports": {
  ".": "./src/runner.mjs"
}
```

Then delete the following line:

```
"main": "index.mjs"
```

The changes we've made tell Node a few things:

- A `type` of `module` tells Node that the project uses ES6 modules by default. It should honor `import` and `export` statements instead of `require` calls.

- The `bin` property tells Node the location of the executable file to run.[3] We'll fill in this file in the next step.

- The `exports` property is a list of named exports that can be imported via `import <x> from "concise-test/<name>"` statements. The `"."` export is the *default* export and defines the main entrypoint.[4] Exports from the given file can be imported using the simpler form `import <x> from "concise-test"`.

Save the file and let's get on with defining the executable file that will act as our application entrypoint.

Creating the Test Runner Entrypoint

Create a new file `src/cli.mjs` with the following content:

```
#!/usr/bin/env node
```

---

[3] https://docs.npmjs.com/files/package.json#bin
[4] https://nodejs.org/api/esm.html#esm_package_exports

```
import { run } from "./runner.mjs";

run();
```

On Unix (or Unix-like platforms), you'll also need to run this command from your project directory:

```
% chmod a+x src/cli.mjs
```

Next up, create a new file src/runner.mjs and add the following stub content:

```
export const run = () => console.log("hello, world!");
```

If you look back at the sequence diagram in Figure 1-3, the code we've written so far is the "Execute" line between the Node and Test runner lifelines and also (arguably) the returning "Report results" line.

Making the Package Available for Use

In order to make our shiny new concise-test test runner available to *other* test projects, we would need to register the project with the global NPM repository and then deploy it. But we're not even close to being finished with it—we've just started!

Another way to do this is *link* your local project.[5] This creates a symbolic link in your local NPM repository cache that points to the location of the current package.

Run the following command from within your concise-test project directory:

```
% npm link
```

You're now ready to use this in a sample project.

---

[5] For more information on how the npm link command works, see the documentation page: https://docs.npmjs.com/cli/v6/commands/npm-link

Using the Sample Application Project

The lifeline on the right side of the sequence diagram in Figure 1-3 is labeled "Your project." This is where your test files will reside.

Take a look in the todo-example directory (the sibling directory to the concise-test directory). This directory stores a Node project that already has a *test script* in the file test/tests.mjs. I'm calling this a test *script* because it doesn't operate with an independent test runner: it *is* the test runner itself.

As the book progresses, you'll eventually dismantle this test script and replace it with tests that work with the concise-test framework.

You can run these tests by performing the following commands:

```
% cd ../todo-example
% node test/tests.mjs
```

All being well, there won't be any output. That means the tests have been successful. Otherwise, you'd see an exception on-screen.

Let's bring in our concise-test test runner.

Run the following command:

```
% npm link concise-test
```

This command is equivalent to doing an npm install, but it is to create a symbolic link in your project's node_modules that links to the target of the symbolic link you created earlier in the local repository: in other words, the concise-test sibling directory.

npm unlink

When you're done working through this book, you may wish to remove your local package link so that you can use the published concise-test from NPM. To do that, use the npm unlink command from both project directories to delete copies of the package from your local Node repository.

Now that it's linked, we can tell Node that the `test` command should run our new executable.

Open the sample application's `package.json` and find the `scripts` entry that looks like this:

```
"test": "node test/tests.mjs"
```

Replace that entry with this:

```
"test": "concise-test"
```

Let's test that out: back on the command line, run `npm test` from within your sample application directory.

Here's what I get:

```
% npm test

> todo-example@1.0.0 test
> concise-test

hello, world!
```

Success! Our executable is now running—you can see the `hello, world!` message right at the bottom.

But we're no longer running our test script.

Update the Test Runner to Run Our Test Suite

It's time to switch back to the `concise-test` project. If you're using an IDE, you should be able to keep files open across both projects as you work on them. If you're editing files on the command line, you'll want to do that with a `cd` command, as shown in the following:

```
% cd ../concise-test
```

In the next few chapters, we'll be moving back and forth between the sample application and the test runner project. On my machine these projects are in two sibling directories: the sample application is `todo-example,` and the test runner is `concise-test.`

If you use the command line, you might want to have two windows (or tabs) open, one for each project directory. In your editor, use a vertical split so that you can see the sample application tests (`todo-example/test/ tests.mjs`) on one side and the test runner code (`concise-test/src/ runner.mjs`) on the other.

If you're using an IDE, you might also want two copies of the IDE open, one for each project, or you can try to open all files in the same window and use the `cd` command in the IDE terminal window, as necessary.

Okay, let's get back to the test runner implementation. In the `concise-test` project test directory, open `src/runner.mjs` and replace the entire contents with this implementation.

Listing 1-1. concise-test/src/runner.mjs

```
import path from "path";

export const run = async () => {
  await import(
    path.resolve(process.cwd(), "test/tests.mjs")
  );
  console.log("Test run finished");
};
```

What this code does is find the file `test/tests.mjs` in the running location and load it into memory.

Let's try that out now. Switch back to the application library and run `npm test`:

```
% npm test
```

```
> test@1.0.0 test
> concise-test

Test run finished
```

Beautiful! But... how do we know if our tests actually ran? Only one way to find out... by breaking a test!

Verify That It Works by Breaking a Test

A common mantra in unit testing is that a test isn't a valid test unless you've seen it fail at least once.

You haven't yet had a chance to explore the code in the application, but now you will. In this section we will make a very simple change to deliberately break one of the tests.

Making a deliberate error in an application is a very useful technique for proving that a test case is running correctly.

Some of the tests in `test/tests.mjs` contain checks that error messages contain specific text. So one way we can introduce a deliberate failure is to change one of these error messages.

Here's a snippet of one of those tests:

```
if (e.message !== "title cannot be blank")
  throw new Error(
    "wrong message in guard clause when " \
    "adding an empty todo"
  );
```

In the sample application, open `src/TodoRepository.mjs` and change the message that matches `title cannot be blank`. A simple way to do this is just add a prefix with an obviously wrong string. My go-to string is `123`, but feel free to be as wild as you like:

```
throw new Error("123title cannot be blank");
```

Now, what would you expect to happen when `npm test` is run?

I'd expect the test runner to return an error with a stack trace and this message:

```
wrong message in guard clause when adding an empty todo
```

Okay, so let's run the `npm test` command again:

```
% npm test

> todo-example@1.0.0
> concise-test

(node:94430) UnhandledPromiseRejectionWarning: Error: wrong
message in guard clause
when adding an empty todo
    at file:///Users/dan/todo-example/test/tests.mjs:16:11
    at ModuleJob.run (internal/modules/esm/module_job.
    mjs:109:37)
    at async Loader.import (internal/modules/esm/loader.
    mjs:133:24)
    at async run (file:///Users/dan/concise-test/src/runner.
    mjs:4:3)
(node:94430) UnhandledPromiseRejectionWarning: Unhandled
promise rejection. This
error originated either by throwing inside of an async function
without a catch
block, or by rejecting a promise which was not handled with
catch(). (rejection
id: 1)
```

Woah... what just happened? Where'd that horrendous UnhandledPromiseRejectionWarning come from?

It turns out that calls to `await import` need to be wrapped in a `try-catch` block if they're going to let exceptions bubble up to the top level.

18

Catch Exceptions from the Await Call

Back in `src/runner.mjs` in the test library—*are you getting used to the file switching yet?*—update the function to include a `try-catch` block. We'll just use `console.error` to print out the error.

Listing 1-2. concise-test/src/runner.mjs

```
import path from "path";

export const run = async () => {
  try {
    await import(
      path.resolve(process.cwd(), "test/tests.mjs")
    );
  } catch (e) {
    console.error(e);
  }
  console.log("Test run finished");
};
```

Now try it out with `npm test`:

```
> todo-example@1.0.0 test
> concise-test

Error: wrong message in guard clause when adding an empty todo
    at file:///Users/dan/todo-example/test/tests.mjs:16:11
    at ModuleJob.run (internal/modules/esm/module_job.
    mjs:109:37)
    at async Loader.import (internal/modules/esm/loader.
    mjs:133:24)
    at async run (file:///Users/dan/concise-test/src/runner.
    mjs:5:3)
Test run finished
```

While this isn't perfect—there's a lot of noise from the stack trace here—you can see a couple of important things.

First, the expected error message is shown on-screen. We will use the exception messages as the basis for our test expectations that we'll cover in Chapter 4.

Second, the very last line is the `Test run finished` output, so we know that our `try-catch` block has caught the error and suppressed it. Our test runner is in control. As you work through Chapter 2, you'll improve this last line by including test summary information.

Committing to Git

Now is a great time to commit your work. If you're using the command line, you can use the following commands:

```
% git init
% git commit -m 'Complete chapter 1'
```

If you're setting up a GitHub repository to save your work, choose the Node `.gitignore` file[6] and choose an appropriate license.

I'm not suggesting you publish this package to NPM or that you begin maintaining your own version of this package. However, it's a great idea to save your progress so you have a record of what you've achieved by going through this book.

---

[6] GitHub's default `.gitignore` file has a whole list of ignored files. I tend to start with a blank file with a single entry: `node_modules`.

Discussion Questions

1. Imagine you're working in a programming language that doesn't support exported symbols, only an entrypoint. How would you build an automated test runner that could execute test cases against units within your codebase? Draw a sequence diagram to show how execution flows from start to finish of a single test run.

2. When developing front-end web applications, it's customary to run unit tests via the Node execution environment and use a "fake" browser Document Object Model (DOM), like the `jsdom` package. Why do you think "real" web browsers are avoided for automated tests? What are the pros and cons of opting for Node?

Summary

This chapter has covered the theory behind application entrypoints and test runners. You created a barebones implementation of a test runner, which looks for a single test file, `test/tests.mjs`, and prints out the first error that's thrown before bombing out.

This is a great place to start building our first test function, `it`, which is exactly what we'll do in Chapter 2.

CHAPTER 2

Building a Function to Represent a Test Case

In this chapter we'll build a function that encapsulates a single test case. It's called `it` and it looks like this:

```
it("describes a test", () => {
  // ... test commands
});
```

Calls to `it` have two parts: the description and the test code. We'll see how this function can be implemented as the first major piece of the `concise-test` test runner.

By the end of the chapter, you'll have learned

- The history of the `it` function

- The importance of describing tests well

- How to use exceptions to represent test failures

- How to make use of color to add emphasis on the command line

- How to use process exit codes to signal pass or failure to process pipelines

Let's begin by looking at where the `it` function comes from.

D. Irvine, *Build Your Own Test Framework*, https://doi.org/10.1007/978-1-4842-9247-1_2

23

The `it` Function

How about we start with the name? Why do we call it "`it`" and not "`test`"?

The name dates back to 2007 when "it" was introduced as a part of the RSpec tool. At that time, Behavior-Driven Development (BDD) was the new "big idea" in testing.

One of the driving forces behind BDD is the idea that people frequently misunderstand what `test` means, and so they attempted to avoid using that word.

What really *is* a test?

The "it" that `it` refers to is the structure under test. That "structure" could be a function, a class, or a component. It's the thing that's being tested.

I prefer `it` over `test` because `it` immediately puts you in the right frame of mind for describing an *action*: what does this thing *do*?

Sometimes, software is specified via formal(ish) specifications that describe behavior with words like *should, will, may,* and so on.[1] And this was often the way that BDD tests were written:

```
it("should do this thing", () => ...)
```

I don't see much point in the "should"[2]—it's a bit too wordy for my liking. Why not just use a simple, present-tense verb?

In the `concise-test` test runner, the first word of the test description should be a verb, so that the `it` reads in plain English: *it describes a test.* Here are some examples:

adds two numbers prints out a string renders an error message dispatches an action

Concise, isn't it?

---

[1] For example, see RFC 2119, from March 1997, at `https://datatracker.ietf.org/doc/html/rfc2119`

[2] It has some use if you think about the *negative* test cases, which would read: *it should not…*

Common Test Verbs

Figure 2-1 shows a plot of verbs used in one of my React codebases.[3] It includes all terms that appear more than twice in the complete test suite.
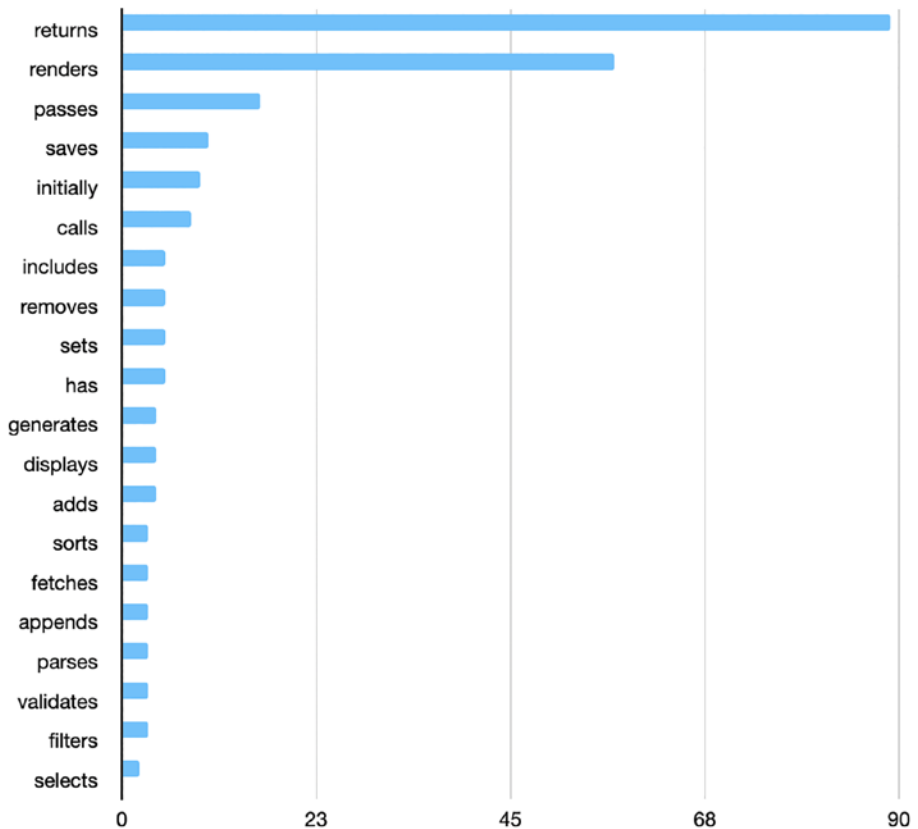
Figure 2-1. *Frequency of starting verbs in test case descriptions*

The clear leader is the verb "returns." How boring! But it makes sense when you think that most of the tests are simply testing the return values of

---

[3] https://github.com/PacktPublishing/Mastering-React-Test-Driven-Development-second-edition

functions. And the next most frequent test verb is "renders," which makes sense given that it's a React codebase.

That's the theory over. Next, it's time to write our implementation of the `it` function.

Using the `it` Function to Group and Describe Test Cases

In this section you'll implement the `it` function in our test runner. Then you'll update the existing `test/tests.mjs` test script to use this new function.

If you squint at the original test script, you'll see the basic structure is a set of `if` statements that possibly throw errors, all ordered one after the other, separated only by new lines.

If you want to figure out what these tests are doing, you'll have to read through the code to figure out what's going on.

The *most important* thing that our test case function must do is *describe* what the test does, so that readers of the test suite don't need to read every line to understand what the tests are doing.

Open `src/runner.mjs` in your editor and add the following definition, just below the `run` function:

```
export const it = (name, body) => body();
```

This is nowhere close to being the final definition of `it`, but this starting point gives us something important: a new scoped context for our variables.

Now go back to the sample project and open `test/tests.mjs`. First, at the top of the file, add an import for `it`:

```
import { it } from "concise-test";
```

Try to run this test (using `npm test` from the sample application directory) to verify that the import has worked. If you get an error about the "main" entrypoint, go back to Chapter 1 and ensure that you ran the instructions in the section "Setting Package Configuration."

Next, take the first test, which should be lines 5–8 if you copied out the previous code sample to the letter, and wrap it in a call to the `it` function.

Listing 2-1. todo-example/test/tests.mjs

```
it("sets completedAt when calling markAsDone", () => {
  const todo = emptyTodo();

  if (!markAsDone(todo).completedAt)
    throw new Error(
      "completedAt not set when calling markAsDone"
    );
});
```

Already this has made a huge difference to the readability of our test. Let's carry on with the next test grouping.

Listing 2-2. todo-example/test/tests.mjs

```
it("throws an exception when adding a todo without a
title", () => {
  const repository = new TodoRepository();

  try {
    repository.add(emptyTodo());
    throw new Error(
      "no error thrown when adding an empty todo"
    );
  } catch (e) {
    if (e.message !== "title cannot be blank")
```

27

```
      throw new Error(
        "wrong message in guard clause when adding an
        empty todo"
      );
  }
});
```

This code sample is interesting because there are *two* checks within this test that codify a logical grouping of information: First, we want an exception to be thrown, and if not, that's a failure. And if the exception *is* thrown, then there's a second check that needs to be performed: that the exception object has the correct content.

You can see that logical groups of expectations have a *nesting effect*. And without due care, test cases can often become needlessly complex. Fortunately, there's a solution for this, which is to build your own matcher. We'll look at that in Chapter 5.

Let's move on to the next test. Here's where things get tricky. If you do exactly the same thing as earlier—wrapping the existing code in an "it" call—you'll get the following code, but unfortunately this won't run.

Listing 2-3. todo-example/test/tests.mjs

```
it("throws errors when adding a repeated todo", () => {
  const newTodo = { ...emptyTodo(), title: "test" };
  repository.add(newTodo);

  const repeatedTodo = { ...newTodo };
  try {
    repository.add(repeatedTodo);
    throw new Error(
      "no error thrown when adding a repeated todo"
    );
  } catch (e) {
```

```
  if (e.message !== "todo already exists")
    throw new Error(
      "wrong message in guard clause when adding an
      existing todo"
    );
  }
});
```

If you run this now, you'll get the following exception from Node:

```
ReferenceError: repository is not defined
```

This error caused our entire test suite to bomb out. Nothing was run past this point. It would be good if each of our tests was run even if one fails; we'll solve that in the next section. But park that thought for now while we focus on completing the task at hand—refactoring the file to use the it function.

When we moved the declaration of const repository into the previous test, we broke this test due to the repository variable going out of scope. Fix it for now by duplicating that definition inside this test. We'll eliminate this duplication when we implement beforeEach in Chapter 4.

Listing 2-4. todo-example/test/tests.mjs

```
it("throws errors when adding a repeated todo", () => {
  const repository = new TodoRepository();
  const newTodo = { ...emptyTodo(), title: "test" };
  repository.add(newTodo);

  const repeatedTodo = { ...newTodo };
  try {
    repository.add(repeatedTodo);
    throw new Error(
      "no error thrown when adding a repeated todo"
```

```
    );
  } catch (e) {
    if (e.message !== "todo already exists")
      throw new Error(
        "wrong message in guard clause when adding an
        existing todo"
      );
  }
});
```

Running tests again, you'll get the same error, but looking at the stack trace shows that the error is further on—this test is now "passing" just fine.

Let's move on to the next test, which is the first test for findAllMatching.

This needs the same trick—we need to duplicate the definition of repository.

Listing 2-5. todo-example/test/tests.mjs

```
it("finds an added todo", () => {
  const repository = new TodoRepository();
  const newTodo = { ...emptyTodo(), title: "test" };
  repository.add(newTodo);

  if (repository.findAllMatching("").length !== 1)
    throw new Error("added todo was not returned");
});
```

On to the next test.

Listing 2-6. todo-example/test/tests.mjs

```
it("filters out todos that do not match filter", () => {
  if (repository.findAllMatching('some other test').
  length !== 0)
    throw new Error("filter was not applied when finding
    matches");
};
```

This test is about the same function, findAllMatching, but you can already see that it has two problems: first, it needs a repository, and second, it needs some setup. If we just added in the repository definition, this test would pass! That's because the "arrange" part of the test—when we set up the test structure—hasn't been completed.

To fix this, just duplicate the setup from the previous test, as shown in the following.

Listing 2-7. todo-example/test/tests.mjs

```
it("filters out todos that do not match filter", () => {
  const repository = new TodoRepository();
  const newTodo = { ...emptyTodo(), title: "test" };
  repository.add(newTodo);

  if (
    repository.findAllMatching("some other test")
      .length !== 0
  )
    throw new Error(
      "filter was not applied when finding matches"
    );
});
```

Wonderful! We've now converted all of our tests to use our new `it` function.

Look how much mileage we got out of that single function. Just to remind you, here it is again:

```
export const it = (name, body) => body();
```

In some respects, all we've done is introduce *indirection*: we've inserted an intermediate function call that does nothing but call another function. But we've also *grouped* the instructions in the body and given them a name.

By changing our tests to use this simple function, we have

- Created clear separation between each test

- Removed interdependencies between code, which means the tests are less brittle[4] than before

- Discovered a logical group of multiple assertions for testing exceptions that could be extracted into an expectation matcher

In addition we've got a few more things we want to address:

- Some of our tests relate to the `add` function, and some relate to the `findAllMatching` function. There's no way to group tests.

- We have duplicated setup in tests.

---

[4] Tests are *brittle* if they are likely to break when updates are made to unrelated parts of the codebase. Tests with shared setup, as in the scenario, are one reason tests can be brittle. Another reason is that tests have too great a surface area. They exercise too much code within one test.

- Our test runner bombs out if any test throws an exception.

- We're not using our test descriptions yet. Our test runner could make use of those.

The first of these we'll address in Chapter 3 when we add the describe function. The second we'll address in Chapter 4 when we add the beforeEach function. But the other two items in our list are fixable right now.

Handling Exceptions

The ideal situation is that each test runs even if it throws an exception. So we'll need to catch exceptions. However, we're still wanting to see the errors produced. When we catch exceptions, we'll use console.error to print the errors out on-screen.

In src/runner.mjs, update the definition of it to read as follows.

Listing 2-8. concise-test/src/runner.mjs

```
export const it = (name, body) => {
  try {
    body();
  } catch (e) {
    console.error(e);
  }
};
```

You can test this out by adding throw new Error("hello, world") or similar to the top of each test in test/tests.mjs. Running tests with npm test should show each error printed out on-screen.

But we aren't using the test name yet. So how can we tell which test produced which error? We can't, at least not yet.

Printing Test Descriptions

We already have a bunch of text for exceptions shown, so we need a way to make the test name appear more prominently than all of the exception detail. Let's print out the name of the failure *in red*. That'll set it apart from the exception test and give some structure to the console report.

For that we need to output some American National Standards Institute (ANSI) escape codes. Escape codes tell your terminal to change certain properties of the text being written to screen, like the color or whether or not the text is bold.

This will only apply if the `npm test` command output is attached to a terminal. For brevity we're going to assume that's the case, but if you've got time, you could add code to stop these escape codes being printed if the command isn't being run on a terminal (e.g., like in CI).

Create a new file in your `concise-test` project called `src/colors.mjs`, and add the following code. By the way, the list of colors here includes all the colors (and text styles like *strike* and *bold*) that are used in later chapters.

Listing 2-9. concise-test/src/colors.mjs

```
const ansiColors = {
  bold: "\u001b[37;1m",
  strike: "\u001b[37;9m",
  cyan: "\u001b[36m",
  green: "\u001b[32m",
  red: "\u001b[31m",
  yellow: "\u001b[33m",
};
```

```javascript
const ansiReset = "\u001b[0m";

export const color = (message) =>
  Object.keys(ansiColors).reduce(
    (message, color) =>
      message
        .replace(
          new RegExp(`<${color}>`, "g"),
          ansiColors[color]
        )
        .replace(
          new RegExp(`</${color}>`, "g"),
          ansiReset
        ),
    message
  );
```

Although most of these colors aren't used within this chapter, we will have used all of them by the end of the book.

The function `color` can be used to output a test failure message like this:

```javascript
console.error(color(`<red>${name}</red>`));
```

It also allows replacements of the same color multiple times on the same line, which is enabled by passing the `"g"` option to the `RegExp` constructor. We'll use this in the next chapter, but here's a sneak peek of how that string looks:

```javascript
`<bold>${describeName}</bold> → <bold>${name}</bold>`;
```

Now I *could* have gone for a simpler approach, by defining a function named `colorRed`:

```
console.error(colorRed(name));
```

But I know that pretty soon I'm going to want to print some green text (for passed tests) and maybe yellow text (for skipped tests). So I've written a single `color` function that will work for all those colors, and I've included all three colors in `ansiColors`. I've also added `bold`, which we'll use in the next chapter.

You might turn around and tell me that this implementation won't work for nested colors:

```
console.error(color(`<red><green>${name}</green></red>`));
```

And I would say to you that you're correct, but to make that work would be to *really* over-engineer the solution if it wasn't over-engineered enough already. I can't envisage any use case where I'd want to include any nested colors like that.

Writing concise software is often about knowing how to walk the fine line between too specific and too generic. The `color` function is more concise than three separate functions for three separate colors, but has obvious limitations to how general it is.

Anyway, that's a digression. Let's update `it` in `src/runner.mjs` to use this new function. First, add an import at the top of the file for the new `color` function:

```
import { color } from "./colors.mjs";
```

Then update it to print out the name of the test failure on error.

Listing 2-10. concise-test/src/runner.mjs

```
export const it = (name, body) => {
  try {
    body();
  } catch (e) {
    console.error(color(`<red>${name}</red>`));
    console.error(e);
  }
};
```

Go ahead and rerun tests for the sample application. You should now see some red text highlighting which tests failed.

Support CI with Correct Exit Codes

When we started suppressing exceptions, we broke something else by accident: the exit code of the process. Your build server will rely on this code to determine if a build should fail or not. We can add that back in by using a Boolean flag to record if any exceptions occurred or not.

Update it in `src/runner.mjs` to read as follows. You'll need to define `anyFailures` outside of it, as a module-level variable.

Listing 2-11. concise-test/src/runner.mjs

```
let anyFailures = false;

export const it = (name, body) => {
  try {
    body();
  } catch (e) {
    console.error(color(`<red>${name}</red>`));
    console.error(e);
```

```
    anyFailures = true;
  }
};
```

At the top of the file, add in a new object, exitCodes, that will define what our exit code is. Success is always 0; anything else is a failure code:

```
const exitCodes = {
  ok: 0,
  failures: 1,
};
```

Then update run to read this variable and return the right code. Add this line to the bottom of the function:

```
process.exit(
  anyFailures ? exitCodes.failures : exitCodes.ok
);
```

If you still have your sample application set up to throw errors, go back to it now and rerun tests with npm test. At the bottom of your test run, you should now see this message:

```
npm ERR! Test failed.   See above for more details.
```

This is Node telling you that it correctly picked up a failure code. You can double-check what the failure code of the last command was by typing the following at your terminal:

```
echo $?
```

With any luck you'll see a 1 returned.

Summarizing a Test Run

To finish off this chapter, let's improve on the `anyFailures` Boolean and return something more useful. Let's record the number of successes and failures instead and then print that out using our `color` function.

Remove the definition of `anyFailures` and replace it with the following two definitions:

```
let successes = 0;
let failures = 0;
```

Update `it` to increment each of these depending on if a test body threw an exception or not.

Listing 2-12. concise-test/src/runner.mjs

```
export const it = (name, body) => {
  try {
    body();
    successes++;
  } catch (e) {
    console.error(color(`<red>${name}</red>`));
    console.error(e);
    failures++;
  }
};
```

Finally, modify `run` by changing the `console.log` statement to say something more useful than just `Test run finished`. And on the last line, you'll also need to replace the `anyFailures` with the `failures !== 0` expression instead.

Listing 2-13. concise-test/src/runner.mjs

```
console.log(
  color(
    `<green>${successes}</green> tests passed,
    <red>${failures}</red> tests failed.`
  )
);
process.exit(
  failures !== 0 ? exitCodes.failures : exitCodes.ok
);
```

We've introduced two module-scope variables here: `successes` and `failures`. Without these, the `it` and `run` functions would need some other way to communicate between them. There are probably other more advanced ways of solving this problem, but this is the simplest way I can think of.

Go ahead and run your tests in the sample application. Update `test/tests.mjs` to have some variation of passing and failing tests just so you can check out how the summary message works.

Exercises

1.   Add a new package command, `npm desc-plot`, that, instead of running your tests, analyzes the test descriptions and counts the occurrences of each starting verb. The output of the command should be CSV data that can be imported into a charting program.

2.   Following on from Exercise 1, draw a bar plot on the command line that looks like Figure 2-1.

Summary

In this chapter we've fleshed out an "`it`" function that records and prints out failures, and we bolstered the `run` function with support for test run summary information and an error exit code for the process if there are any test failures.

We also spotted some immediate places where we can improve our sample application's tests.

Your test runner now has a working implementation of the basic unit of test case encapsulation. The next thing we need is a mechanism for *organizing* test cases, and we'll look at that in the next chapter.

CHAPTER 3

Grouping Tests

In this chapter we'll add the `describe` function to group our tests. This function is a core mechanism for organizing tests.

You'll learn

- The mechanisms in place for organizing tests

- How to implement the `describe` function

- How to build a test report that contains organizational information

By the end of the chapter, you'll have a deep understanding of how all unit test projects are structured.

The Basics of Unit Test Organization

The test script file you've been working with, `test/tests.mjs`, already has *two* types of organization.

First, it's in a file of its own. Test runners can rely on the hierarchical nature of the file system to provide hierarchical test organization. This structure can mirror the file organization of your application code.

Second, the test script has an order: it runs from top to bottom. You can use this to order your tests so that they tell a story, just like this book. In each file, the most important tests should come first: these are the tests that describe the core functionality of your file. As the file progresses, your tests can get more and more detailed, filling in the nuances of your unit.

But there's a *third* mechanism that test runners will generally have, and that's a function we'll call `describe`.

Why is a describe function necessary? It turns out that there's not always a one-to-one mapping between application source file and its tests.

Sometimes–hopefully rarely–some application modules are so complex that the story the tests tell is clearer when it's split into different parts. If you've ever found yourself attempting to make sense of a 3,000-line test file, you'll know exactly what I mean.

The other reason is that you often need a way to *nest* groups, and using file system directories to do this is just not fun–they can't be given verbose descriptions.

Once we've written our `describe` function implementation, we'll look at how the `test/tests.mjs` file can be structured to tell its story effectively.

Figure 3-1 shows how the test runner uses the file system in combination with test suites and test cases to form a complete automated test plan.
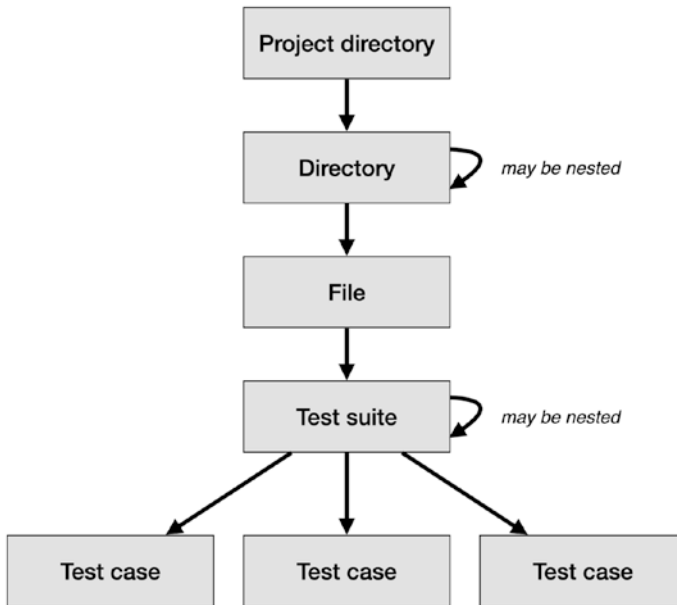
Figure 3-1. *The unit test organizational hierarchy*

That's it for the theory; time to build the function.

A Starting Point for `describe`

As ever, let's start with the simplest thing we can think of. We can write a `describe` function in exactly the same way as we wrote `it` in the last chapter.

In `src/runner.mjs`, add the following function definition at the bottom of the file:

```
export const describe = (name, body) => body();
```

It's the *exact same thing* as the `it` function! (Well, for now, at least.)

Back in our sample application, let's introduce a couple of `describe` blocks. Open `test/tests.mjs`. We'll create two `describe` blocks; the first

is for `todo`, and the second is for `TodoRepository`, which matches the imports at the top of the file. You'll also need to update the `concise-test` import to pull in `describe`.

Listing 3-1. todo-example/test/tests.mjs

```
import { describe, it } from "concise-test";

describe("todo", () => {
  it("sets completedAt when calling markAsDone", ...);
});

describe("TodoRepository",  () => {
  // ... all the remaining tests here ...
});
```

Running these tests should produce the same output as before (I still have a couple of tests left as failing, which is helping me visually verify all the behavior we've coded so far).

Rethinking Test Output

So where do we go from here? Describe blocks are there to organize our output, but we're not actually printing out any test information other than test failures. So perhaps we should start by printing out a list of successful tests and a list of describe block names.

Start with `describe`. Modify the body as shown.

Listing 3-2. concise-test/src/runner.mjs

```
export const describe = (name, body) => {
  console.log(name);
  body();
};
```

This definition is subtly different from the version before; this one no longer returns the value of the describe block.

Before we come on to updating `it`, let's just stop and think about what we're aiming for. We know we absolutely need to see failure output, and we want to see that at the bottom of whatever list we're looking at. It's important to see these failure messages at the bottom because when you're using tests to drive your design, it's the failure text that's most useful to you. And when you're using command-line applications, it's the last bit of text that'll be on-screen after the application exits. All previous output will have scrolled out of view. You don't want to have to scroll up to see detailed failure information.

Now that we've got grouping of tests, it'd be nice to see a well-formatted tree of tests that were included in the run and the result of those tests as they were run.

The test output should have three phases:

1. A list of tests as they are run, with pass or failure result for each

2. A list of test failures

3. A one-line test summary, displaying counts of tests run and failed

To get to this stage, we need to make two changes:

- After a test run, always print out the name with a pass/ fail result.

- Defer printing of failure information until all tests have run.

We'll work on these individually and think through the details as we go.

Printing a Pass/Fail Response for Each Test

Right now we're printing out the failure message in red text. You might think that we could add in passing tests in green text, but this won't work. It's problematic for any displays that don't have color support, like text files. If you ran tests on your build server, you wouldn't be able to look at the build output and tell immediately what tests passed and what tests failed.

We need a visual indicator in addition to color. Before each test name, we'll print out either a green tick (✓) or a red cross (✗).

The following JavaScript snippet uses the Unicode character codes for the tick (✓) and cross (✗) symbols. However, you could simply use the Unicode character itself rather than using the character code.

Update the it function as follows.

Listing 3-3. concise-test/src/runner.mjs

```
const tick = "\u2713";
const cross = "\u2717";
export const it = (name, body) => {
  try {
    body();
    console.log(
      color(`  <green>${tick}</green> ${name}`)
    );
    successes++;
  } catch (e) {
    console.log(color(`  <red>${cross}</red> ${name}`));
    console.error(e);
    failures++;
  }
};
```

I've changed `console.error` to `console.log`. I've done that because the intention of this output is subtly different: it's a summary statement rather than an error. The error message will still be printed, but later on.

If you run tests now, you'll see that our list is now looking well, with our describe block names in place. But our failure details need to be moved to after the test listing.

If you find that the tick and cross marks are not displaying when you run your tests, it's most likely because your terminal font doesn't contain these Unicode characters. If that's the case, you could either replace the characters with the words PASS and FAIL, or you could try downloading and installing a new font.

Ending a Test Report with Test Failure Details

We'll do this by changing `failures` from an integer to an array of `Error` objects.

Start by changing the declaration of failures at the top of the file:

```
let failures = [];
```

Now define a new `printFailure` and `printFailures` function. These build on the functionality we had before, but giving us an informative header and adding some spacing between failures.

Listing 3-4. concise-test/src/runner.mjs

```
const printFailure = (e) => {
  console.error(e);
  console.error("");
};

const printFailures = () => {
  if (failures.length > 0) {
    console.error("");
```

```
    console.error("Failures:");
    console.error("");
  }
  failures.forEach(printFailure);
};
```

Then update `it` to push errors to this list, and remove the second `console.error` statement that prints out the error.

Listing 3-5. concise-test/src/runner.mjs

```
export const it = (name, body) => {
  try {
    body();
    console.log(
      color(`  <green>${tick}</green> ${name}`)
    );
    successes++;
  } catch (e) {
    console.error(color(`  <red>${cross}</red> ${name}`));
    failures.push(e);
  }
};
```

Finally, update `run` to call `printFailures`. It also needs to create a summary message using `failures.length` rather than `failures` itself.

Listing 3-6. concise-test/src/runner.mjs

```
export const run = async () => {
  try {
    await import(
      path.resolve(process.cwd(), "test/tests.mjs")
    );
```

```
  } catch (e) {
    console.error(e);
  }
  printFailures();
  console.log(
    color(
      `<green>${successes}</green> tests passed, ` +
        `<red>${failures.length}</red> tests failed.`
    )
  );
  process.exit(
    failures.length > 0
      ? exitCodes.failures
      : exitCodes.ok
  );
};
```

Go ahead and run tests for your sample application. Things are starting to look rather neat!

```
> todo-example@1.0.0 test
> concise-test

todo
  ✓ sets completedAt when calling markAsDone

TodoRepository
  ✓ throws an exception when adding a todo without a title
✗ throws errors when adding a repeated todo
  ✓ finds an added todo
  ✓ filters out todos that do not match filter

Failures:
```

```
Error: hello, world
    at file:///Users/dan/byotf/todo-example/test/tests.
    mjs:35:13
    at it (file:///Users/dan/byotf/concise-test/src/runner.
    mjs:40:5)
    at file:///Users/dan/byotf/todo-example/test/tests.mjs:34:5
    at describe (file:///Users/dan/byotf/concise-test/src/
    runner.mjs:81:3)
    at file:///Users/dan/byotf/todo-example/test/tests.mjs:16:1
    at ModuleJob.run (node:internal/modules/esm/module_
    job:193:25)
    at async Promise.all (index 0)
    at async ESMLoader.import (node:internal/modules/esm/
    loader:528:24)
    at async run (file:///Users/dan/byotf/concise-test/src/
    runner.mjs:15:5)
4 tests passed, 1 tests failed.
```

There's one thing that doesn't seem quite right, however. We've lost the link between the *name* of a test failure and its exception. The test failures would be much better presented if the name of the test was included.

Saving Test Context Information

To properly identify a failure, we'll need to report not just the failure exception but the test name too. And in addition to the test name, we'll want the describe block name too.

Something like this, for example:

```
TodoRepository → finds an added todo
```

In order to print this information, we must record more than just the failure `Error` object. Let's use the following structure to represent a failure:

```
{
  error: <original Error instance>,
  name: <test name>,
  describeName: <describe block name>
}
```

At the top of `src/runner.mjs`, next to the declarations of `failures` and `successes`, add a third module-level variable:

```
let currentDescribe;
```

Then modify `describe` as follows.

Listing 3-7. concise-test/src/runner.mjs

```
export const describe = (name, body) => {
  console.log(name);
  currentDescribe = name;
  body();
  currentDescribe = undefined;
};
```

In `it`, update the invocation of `failures.push` with the following.

Listing 3-8. concise-test/src/runner.mjs

```
failures.push({
  error: e,
  name,
  describeName: currentDescribe,
});
```

53

Create a new function called `fullTestDescription`:

```
const fullTestDescription = ({ name, describeName }) =>
  `<bold>${describeName}</bold> → <bold>${name}</bold>`;
```

This function won't work if there's no `describe` block–in other words, if the tests are defined without any parent `describe` block–because no `describeName` will be set. We'll fix that in the next section, when we add support for nested describe blocks.

Finally, update `printFailure` to print the test failure name.

Listing 3-9. concise-test/src/runner.mjs

```
const printFailure = (failure) => {
  console.error(color(fullTestDescription(failure)));
  console.error(failure.error);
  console.error("");
};
```

Go back to your sample application and run tests again. Make sure you have a couple of tests throwing `Error` objects just so you can behold the beauty that you've created.

Supporting Nested `describe` Blocks

We're not done with `describe` blocks just yet. Let's add some support for nesting. This is very useful because it allows us to have subgroups of related tests. For example, we can split our `TodoRepository` tests into two sub-describe blocks: one for the `findAllMatching` function and one for the `add` function.

Let's do that now, before we make any further change. In your sample application, open `test/tests.mjs` and add two new `describe` calls in, as follows.

Listing 3-10. todo-example/test/tests.mjs

```
describe("TodoRepository", () => {
  describe("add method", () => {
    it("throws an exception when adding a todo without a
    title", ...)
    it("throws errors when adding a repeated todo", ...)
  });
  describe("findAllMatching method", () => {
    it("finds an added todo", ...)
    it("filters out todos that do not match filter", ...)
  });
})
```

Notice how these tests read like a normal plain-English statement of fact: *"TodoRepository findAllMatching method finds an added todo."* It's a specification.

Go ahead and run the sample application specs with `npm test`. Look closely at the test run list: the describe block names are there, but we don't have an extra level of indent. In the test failures section, we haven't got *both* describe block names listed, just the first one.

So how can we solve this?

We need a stack of describe block names, shown in Figure 3-2. Each time we enter a describe block, we push a name to the stack, and when we leave a describe block, we pop a name off.
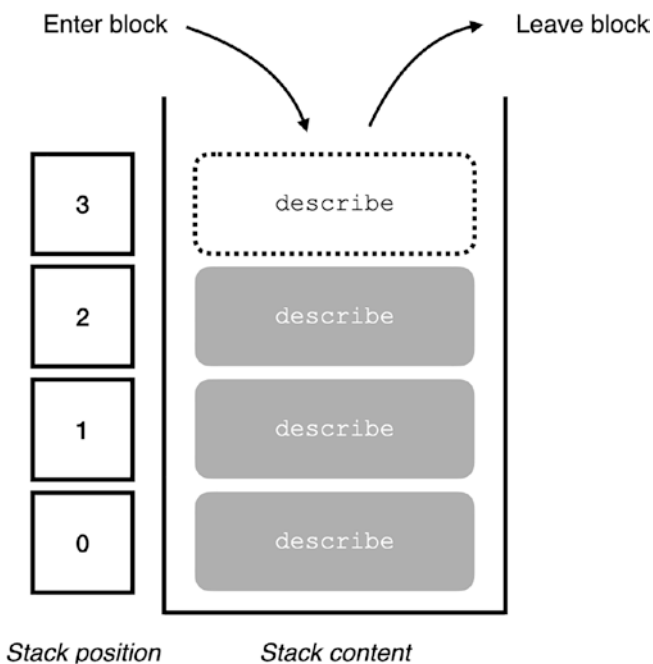
Figure 3-2. *Obligatory diagram of a stack*

The *size* of the stack can then be used to tell us how far to indent any given test case.

Back in `src/runner.mjs`, replace `currentDescribe` with `describeStack`:

```
let describeStack = [];
```

Yes, the stack is a simple array. Update `describe` to push to and pop from this stack.

Listing 3-11. concise-test/src/runner.mjs

```
const withoutLast = (arr) => arr.slice(0, -1);

export const describe = (name, body) => {
  console.log(name);
```

```
  describeStack = [...describeStack, name];
  body();
  describeStack = withoutLast(describeStack);
};
```

The definition of `withoutLast` is important. We'll use it again in Chapter 4.

Update `it` to save a snapshot of this stack along with each failure, shown in the following:

```
failures.push({ error: e, name, describeStack });
```

Finally, update `fullTestDescription` to join each of these segments together:

```
const fullTestDescription = ({ name, describeStack }) =>
  [...describeStack, name]
    .map((name) => `<bold>${name}</bold>`)
    .join(" → ");
```

Now all that's left is correcting the indent. We'll add two spaces of indent for each describe block.

Define a new function called `indent`, which will make use of the `describeStack` variable:

```
const indent = (message) =>
  `${" ".repeat(describeStack.length * 2)}${message}`;
```

This function can make use of `describeStack` because it's used during the test run phase. The previous function, `fullTestDescription`, requires a snapshot of the stack to be saved because it is invoked *after* the test run has completed.

You can now use this function in the `it` function. Remove the two empty spaces at the start of each string and instead insert a call to `indent`.

Listing 3-12. concise-test/src/runner.mjs

```
export const it = (name, body) => {
  // ...
  console.log(
    indent(color(`<green>${tick}</green> ${name}`))
  );
  // ...
  console.log(
    indent(color(`<red>${cross}</red> ${name}`))
  );
  // ...
};
```

Then invoke `indent` in `describe` too:

```
export const describe = (name, body) => {
  console.log(indent(name));
  // ...
};
```

That's us done! Look how pretty our output is now:

```
> todo-example@1.0.0 test
> concise-test

todo
  ✔ sets completedAt when calling markAsDone

TodoRepository
  add
    ✔ throws an exception when adding a todo without a title
    ✗ throws errors when adding a repeated todo
  findAllMatching
```

✓ finds an added todo
✓ filters out todos that do not match filter

Failures:

TodoRepository → add → throws errors when adding a
repeated todo

Error: hello, world
    at file:///Users/dan/byotf/todo-example/test/tests.
    mjs:35:13
    at it (file:///Users/dan/byotf/concise-test/src/runner.
    mjs:40:5)
    at file:///Users/dan/byotf/todo-example/test/tests.mjs:34:5
    at describe (file:///Users/dan/byotf/concise-test/src/
    runner.mjs:80:3)
    at file:///Users/dan/byotf/todo-example/test/tests.mjs:17:3
    at describe (file:///Users/dan/byotf/concise-test/src/
    runner.mjs:80:3)
    at file:///Users/dan/byotf/todo-example/test/tests.mjs:16:1
    at ModuleJob.run (node:internal/modules/esm/module_
    job:193:25)
    at async Promise.all (index 0)
    at async ESMLoader.import (node:internal/modules/esm/
    loader:528:24)

4 tests passed, 1 tests failed.

Exercises

1. Create a synonym of `describe`, "`when,`" that cannot be nested any further.

2. Modify the test reporter to avoid nesting of context names.

3. Modify the test reporter so that it neatly inserts line breaks and tabs if the line is longer than the terminal width.

Summary

In this chapter we've added support for nested describe blocks, and we've tidied up our test report, which is now looking rather awesome.

That just about sums up `describe`–for now. We'll revisit this (and `it`) when we add support for focusing on tests with `describe.only` and `it.only`.

Our code is also starting to get a little messy. As we go into the next chapter, you may wish to think about ways in which you might organize our code a little better.

The next chapter reaches deeper into `describe` support by adding two more incredibly necessary functions for any test framework, `beforeEach` and `afterEach`.

CHAPTER 4

Promoting Conciseness with Shared Setup and Teardown

Our `concise-test` automated test framework is starting to take shape. We've already got something that's usable: we can use the `it` function to write test cases and the `describe` function to group them.

We don't yet have any mechanism for dealing with duplication between tests. And we do have some duplication in our own sample application. In `test/tests.mjs`, for example, three of our tests start in the exact same way, with these three lines:

```
const repository = new TodoRepository();
const newTodo = { ...emptyTodo(), title: "test" };
repository.add(newTodo);
```

In this chapter we'll create a `beforeEach` function that we can use to declare test setup in a single place. This setup code is associated with the containing `describe` block and is run before every single test within the test context.

By the end of the chapter, you'll have learned

- The *Arrange, Act, Assert* (AAA) pattern and why it's a beneficial test structure

- How the `beforeEach` block works and how it's implemented

- The same for the `afterEach` block

Let's start with a look at one of the most useful ideas in automated testing: the *Arrange, Act, Assert* pattern.

The Arrange, Act, Assert Pattern

The 1994 paper in which Kent Beck described his Smalltalk test runner contained the ideas of *Fixtures*, *Cases*, and *Checks*.[1] It turns out that this is a very logical way to think about automated unit test scripts.

In 2001, Bill Wake described the *Arrange*, *Act*, *Assert* pattern[2] (or AAA pattern), which is roughly analogous. And this term seems to have caught on within the community as a concrete principle to apply when writing unit tests.

The popular Cucumber testing tool,[3] with its Gherkin syntax, uses the term *Given*, *When*, *Then*, which are also, in some ways, equivalent in meaning.

So what does it mean?

---

[1] The Smalltalk Report, October 1994

[2] https://xp123.com/articles/3a-arrange-act-assert/

[3] https://cucumber.io

Arrange

Every test starts with getting the system into the right state, which is the "fixture." You "fix" all the pieces together ready for the test to begin. In contrast to some forms of system test, unit tests do *not* start from the application initial state. Instead, they prime the system to get into a certain known and valid state, which is when the test case really begins.

This is what we call the *Arrange* phase of the test.

Is the arranged system state a *valid* system state?

Note the very important assumption that the state of the system at the end of the *Arrange* phase is valid. Otherwise, the next phase (the *Act* bit of the test) won't be valid either. So how do you make sure the *Arrange* steps are valid? Why, writing unit tests for each of *those* steps, of course!

Act

Now we perform the action on the object under test. Generally speaking, this action portion of the test should be as short and succinct as possible: a single function call is all it takes.

Because this is the main part of the test, and it's the bit we're holding under the microscope, it's very important that the function call, the arguments, and any variable storing the result are right there in the test case.

Assert

Finally, after the action has finished, we *check* the state of the system. Did everything that we expected to change actually change? Did the things we expected to stay the same actually stay the same?

We will look at the *Assert* phase in detail in Chapter 5.

Why Is This Pattern Important?

Sometimes developers write tests in a more complicated fashion than they need to be. Take the following example test:

```
it("does something smart", () => {
  setupA();                     // Arrange
  setupB();                     // Arrange
  const result1 = actA();       // Act
  expect(result1).toEqual(X);   // Assert
  const result2 = actB();       // Act
  expect(result2).toEqual(Y);   // Assert
});
```

Note the order:

Arrange ➔ Act ➔ Assert ➔ Act ➔ Assert

Now, proponents of the AAA pattern will tell you that this should actually be two tests:

```
it("does something semi-smart", () => {
  setupA();                     // Arrange
  setupB();                     // Arrange
  const result1 = actA();       // Act
  expect(result1).toEqual(X);   // Assert
});

it("does something also semi-smart", () => {
  setupA();                     // Arrange
  setupB();                     // Arrange
  const result1 = actA();       // Arrange -- no longer Act!
```

```
  const result2 = actB();      // Act
  expect(result2).toEqual(Y); // Assert
});
```

What is the benefit in tests of this form? Well, remember the notion of needing a *valid* test setup, or otherwise the test case itself is invalid?

In the first version of the test, if the first *Act* statement fails or does something unexpected, then you can't sensibly reason about the second half of the test.

Figure 4-1 shows how this "funnel" of Arrange to Act to Assert works and the characteristics of each part of these tests.

Arrange ⟶	Act ⟶	Assert
Clean slate for each test	Single statement	One logical assertion per test
Setup & teardown for dependencies	Application API visible in test	Matchers for reuse and clarity
Simple test doubles	Significant parameters on show	
Factories for reuse and clarity	Insignificant parameters hidden	
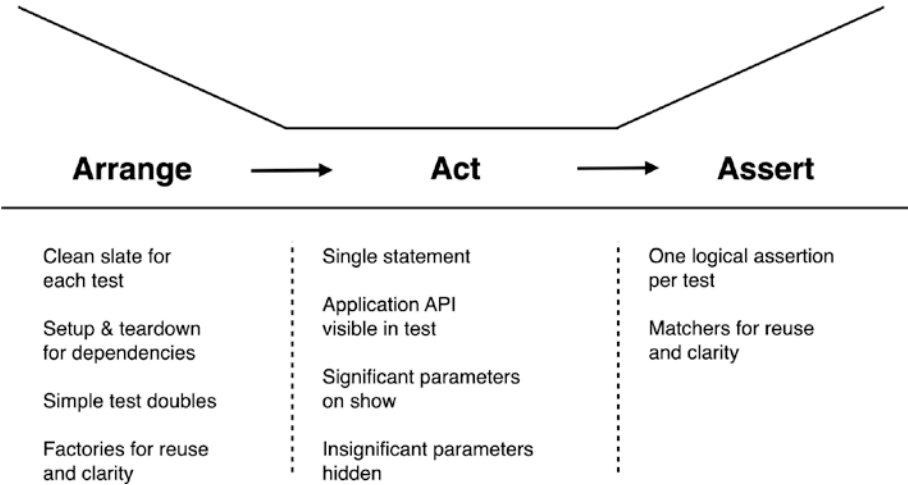
Figure 4-1. *The Arrange, Act, Assert funnel and how each part is used to structure tests*

This is a good indicator of what we mean by a *unit* test vs. a system test or any other type of test. It isn't to do with the structure under test, but rather it's to do with the pincer movement we are making.

Introducing the beforeEach Function

Let's suppose, then, that all our tests will be written in AAA order. Will there be a lot of repetitive *Arrange* phases?

Yes, and that's where the beforeEach function comes in. It takes all the *Arrange* statements and makes sure they are run before every single test case:

```
beforeEach(() => {
  setupA(); // Arrange
  setupB(); // Arrange
});

it("does something semi-smart", () => {
  const result1 = actA();      // Act
  expect(result1).toEqual(X);  // Assert
});

it("does something also semi-smart", () => {
  const result1 = actA();     // Arrange
  const result2 = actB();     // Act
  expect(result2).toEqual(Y); // Assert
});
```

The afterEach Function

Our sample application doesn't have any need for the afterEach function. But it should be clear that this function is run *after* each test case.

In general, the afterEach function is used nowhere near as much as the beforeEach function. It's often there to support undoing changes to global state. But it's good practice to avoid modifying global state at all. (Ironic that we're doing it here in our test framework!)

66

We don't have any need for *multiple* `beforeEach` blocks, but we'll add support for that too. There's no functional need for multiple `beforeEach` blocks, but they can be useful to organize logically different groupings into different things. We'll also see a use case later on in the book when we come to add shared example support with `it.behavesLike`.

> `beforeAll` and `afterAll` are also commonly seen in test frameworks. I'm not including them in this book because I don't think they are necessary for our framework.[4]

Applying beforeEach Blocks to Our Test

We'll need to modify `it` to run all `beforeEach` blocks and all `afterEach` blocks in a specific order.

The order is straightforward: blocks are run in the order that they are declared, with parent blocks being run before nested blocks. Figure 4-2 shows an example of how these functions can be called in order.

---

[4] If you have code you want to run before or after the test, just include it within the `describe` block itself.
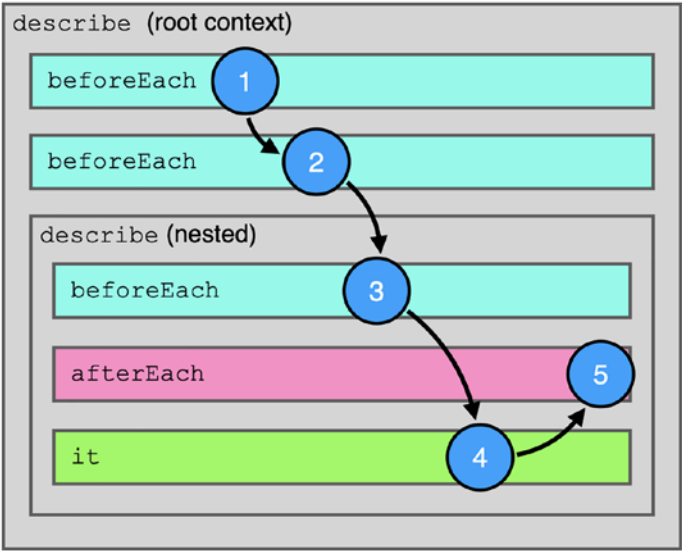
Figure 4-2. *Execution order of a single test. Function bodies are executed in order from 1 to 5*

Our current implementation saves the *name* of each describe block to describeStack. Now we'll modify the stack to store an object that represents each describe block in its entirety:

```
{
  name: <existing entry on the stack>,
  befores: <an ordered array of all beforeEach blocks>,
  afters: <an ordered array of all afterEach blocks>
}
```

We'll then update it to call each of the right functions in turn and fullTestDescription to pull out the test names from this new object.

Refactoring describeStack to an Object

Let's refactor what we've got to an object. describeStack will turn from being an array of strings to an array of objects, each with a name string property.

In src/runner.mjs, modify describe to push a new object, as shown.

Listing 4-1. concise-test/src/runner.mjs

```
export const describe = (name, body) => {
  console.log(indent(name));
  describeStack = [...describeStack, { name }];
  body();
  describeStack = withoutLast(describeStack);
};
```

Then modify fullTestDescription to destructure that object and to wrap the test name in an object itself.

Listing 4-2. concise-test/src/runner.mjs

```
const fullTestDescription = ({ name, describeStack }) =>
  [...describeStack, { name }]
    .map(({ name }) => `<bold>${name}</bold>`)
    .join(" → ");
```

There's a symmetry between describe and it calls. We won't find a generalization in this book, but by the end of the book, we will have written a great deal of very similar (almost duplicated) code between the two. Any potential improvement is left up to you!

Run your sample application tests with npm test and verify that they still behave the same way.

Defining beforeEach

Before we define this function, let's ensure we set each new describe block up with an empty `befores` and `afters` array.

Listing 4-3. concise-test/src/runner.mjs

```
const makeDescribe = (name) => ({
  name,
  befores: [],
  afters: [],
});

export const describe = (name, body) => {
  console.log(indent(name));
  describeStack = [...describeStack, makeDescribe(name)];
  body();
  describeStack = withoutLast(describeStack);
};
```

Let's start with `beforeEach`, and then when we've got that working, we can duplicate it (or pull out an abstraction) for `afterEach`.

Listing 4-4. concise-test/src/runner.mjs

```
const last = (arr) => arr[arr.length - 1];

export const beforeEach = (body) => {
  const newDescribe = {
    ...last(describeStack),
    befores: [...last(describeStack).befores, body],
  };
```

```
  describeStack = [
    ...withoutLast(describeStack),
    newDescribe,
  ];
};
```

Unlike `it`, invoking `beforeEach` outside of a describe block will cause it to blow up. That's because it assumes that `describeStack` has at least one entry.

Okay, very good. We now have an array of `describe` blocks inside `describeStack`, and we have an array of `before` blocks inside `befores`. That means to run each `beforeEach` block means flattening these arrays and invoking each in turn.

Perhaps we should rename `describeStack` as `describes`! Unfortunately, *describes* is already an actual word…

Add the new function `invokeBefores`, just above the `it` function.

Listing 4-5. concise-test/src/runner.mjs

```
const invokeBefores = () =>
  describeStack
    .flatMap((describe) => describe.befores)
    .forEach((before) => before());
```

Well, that was easy!

Let's insert this into our `it` function.

Listing 4-6. concise-test/src/runner.mjs

```
try {
  invokeBefores();
  body();
  // ...
}
```

Did you think that would be harder than it was?! Time to try it out to see if it works as intended.

Updating the Sample Application

Back in your sample application, update `test/tests.mjs` to include `beforeEach` in the imports:

```
import { beforeEach, describe, it } from "concise-test";
```

Then add a `beforeEach` block inside the `TodoRepository` describe block, together with a couple of declarations that we'll discuss after.

Listing 4-7. todo-example/test/tests.mjs

```
describe("TodoRepository", () => {
  const newTodo = { ...emptyTodo(), title: "test" };
  let repository;

  beforeEach(() => {
    repository = new TodoRepository();
  });
  // ... existing tests ...
});
```

The first declaration is `const newTodo`, which has been pulled into the `describe` scope. We could have actually done this earlier. Since the same value is used in each test, we can reuse it throughout each test. Just remember that this value will only ever be instantiated once across all tests.

Some cross-language talk now. In the xUnit test framework pattern for static languages like Java and C#, it's common for a new test suite object to be instantiated for every test that's run. That means you can safely instantiate objects in the test suite constructor, rather than using a before block. That is **not** the case here. The code declared in the `describe` block is run once, when you invoke the `describe` block. The same constants will be used for every single test in the suite.

This is a potential source of bugs. Any value you pull out into a `describe` declaration should be a value that's used to arrange your structure under test only. It shouldn't be modified in anyway during the test.

The `newTodo` value is a great example of this. It's always used to set up our object. None of the tests will change its value.

The second declaration is `let repository`. It's a `let` because we've split the declaration and assignment into two parts. It's declared in the `describe` scope so we can access it in each of our tests, but it's assigned in the `beforeEach`. Not ideal, but it's a good trade-off given the benefit we receive in more concise code.

Now delete those two lines from the start of all four tests in the describe block. The first test actually only uses the first line, but the second line is harmless and won't affect it.

Running the tests should still work.

There's another beforeEach block we can remove, and we can test our nested logic in the process. In the findAllMatching nested describe block, both of our tests call repository.add(newTodo) as part of their *Arrange* phase. So pull that out as a beforeEach block too.

Listing 4-8. todo-example/test/tests.mjs

```
describe("findAllMatching", () => {
  beforeEach(() => {
    repository.add(newTodo);
  });

  // ... existing tests here ...
});
```

This beforeEach will be called *after* the parent describe block's beforeEach, so repository will have already been instantiated by the time this block is run.

Remove the same line from the two tests and rerun the tests. You should find everything still passes.

It's important to realize that we only pulled this out because the call is part of the *Arrange* **phase**. One of the preceding two tests also has exactly the same line: repository.add(newTodo). However, in that test the call to repostory.add is the *Act* **phase** of the test. Generally speaking, the Act phase of a test should never be in a beforeEach block.

Defining afterEach

Now that we have manually tested the `beforeEach` block, how can we reuse what we've done with `afterEach`?

Sure, we *could* just duplicate the code. But how about we try to generalize what we've got instead? Both `beforeEach` and `invokeBefores` seem like they have a lot of useful knowledge that could be pulled out and reused for `afterEach` and `invokeAfters`.

Generalizing beforeEach

Add these two functions, `currentDescribe` and `updateDescribe`, just above `beforeEach`.

Listing 4-9. concise-test/src/runner.mjs

```
const currentDescribe = () => last(describeStack);

const updateDescribe = (newProps) => {
  const newDescribe = {
    ...currentDescribe(),
    ...newProps,
  };
  describeStack = [
    ...withoutLast(describeStack),
    newDescribe,
  ];
};
```

Now `beforeEach` becomes a single-liner:

```
export const beforeEach = (body) =>
  updateDescribe({
    befores: [...currentDescribe().befores, body],
  });
```

After making that change, make sure you run the tests for the sample application again.

And `afterEach` is now a much shorter affair:

```
export const afterEach = (body) =>
  updateDescribe({
    afters: [...currentDescribe().afters, body],
  });
```

Generalizing invokeBefores

This is already a short and sweet function. There are a number of options for sucking out code from this, but the one I'd go with is this.

Listing 4-10. concise-test/src/runner.mjs

```
const invokeAll = (fnArray) =>
  fnArray.forEach((fn) => fn());

const invokeBefores = () =>
  invokeAll(
    describeStack.flatMap((describe) => describe.befores)
  );
```

Then `invokeAfters` is as simple as

```
const invokeAfters = () =>
  invokeAll(
```

```
    describeStack.flatMap((describe) => describe.afters)
  );
```

You can then make the final change to update `it` to run both of these functions.

Listing 4-11. concise-test/src/runner.mjs

```
export const it = (name, body) => {
  try {
    invokeBefores();
    body();
    invokeAfters();
    console.log(
      indent(color(`<green>✓</green> ${name}`))
    );
    successes++;
  } catch (e) {
    console.log(indent(color(`<red>✗</red> ${name}`)));
    failures.push({ error: e, name, describeStack });
  }
};
```

Exercise

1. Extend the test runner so that if an exception occurs during the first run of a `beforeEach` block, no further tests within the entire `describe` context will run and instead the `describe` block is marked as failing. Ensure that the command output makes it clear that the error was an exception in the `beforeEach` block and not a test failure.

Discussion Question

1. The `beforeEach` block is useful for the *Arrange* phase of the *Arrange, Act, Assert* pattern. But how does the `afterEach` block fit into this pattern? It is not part of the *Act* or *Assert* phase. Can you argue that it is part of the *Arrange* phase, or is it something else? Is its use an anti-pattern? What are some valid use cases of `afterEach` that you've come across in your own work?

Summary

This chapter began with a look at the *Arrange, Act, Assert* pattern and why it's a good structure for writing tests.

Then we built support for `beforeEach` and `afterEach` functions. These functions (particularly the `beforeEach` function) are crucial for writing short and concise testing.

Our test runner is taking shape. We just have one more core function to build, and that's the `expect` function and its associated matchers, which we'll build in the next chapter.

CHAPTER 5

Improving Legibility with Expectations and Matchers

In this chapter we'll build out the expect function and its associated *matcher functions*. This is the *Arrange* phase of the test, which can be viewed as one or more expectations.

An *expectation* is a type of assertion about the state of the system that reads in plain English, like these examples:

```
expect(todo.createdAt).toBeDefined();

expect(repository.findAllMatching("")).toHaveLength(1);

expect(() => repository.add(emptyTodo())).toThrow(
  new Error("title cannot be blank")
);
```

The argument to expect is the *actual* value that exists after the *Act* phase of the test is complete. The bit on the right–toBeDefined(), toHaveLength(1)–is the *matcher expression*. The matcher is one of the fundamental mechanisms for keeping test suites maintainable.

By the end of the chapter, you'll have learned

- How judicious use of matchers improves the maintainability of test suites

- How to build a user-defined matcher

- How to support multiple `expect` function calls within a single test

Let's start with a closer look at matchers in a real-world test project.

Using Matchers for Clarity and Reuse

A matcher can be used to test a logical concept. The name of the matcher should make it instantly clear to the reader what the matcher is doing.

The most basic test systems have an `assert` statement that takes a single argument and fails the test if the argument resolves to false:

```
assert(x == y)
```

This is roughly equivalent to the `toEqual` matcher on an expectation:

```
expect(x).toEqual(y);
```

Beyond equality, however, matchers have distinct advantages to expressions passed into assert calls:

- They are written in plain English so they can be read quickly without having to scrutinize details.

- They can encapsulate complex business logic in a single statement.

- Each matcher can have its own failure description logic to aid in diagnosing failures when they occur.

Figure 5-1 shows a plot of all the matchers used in a small React project.
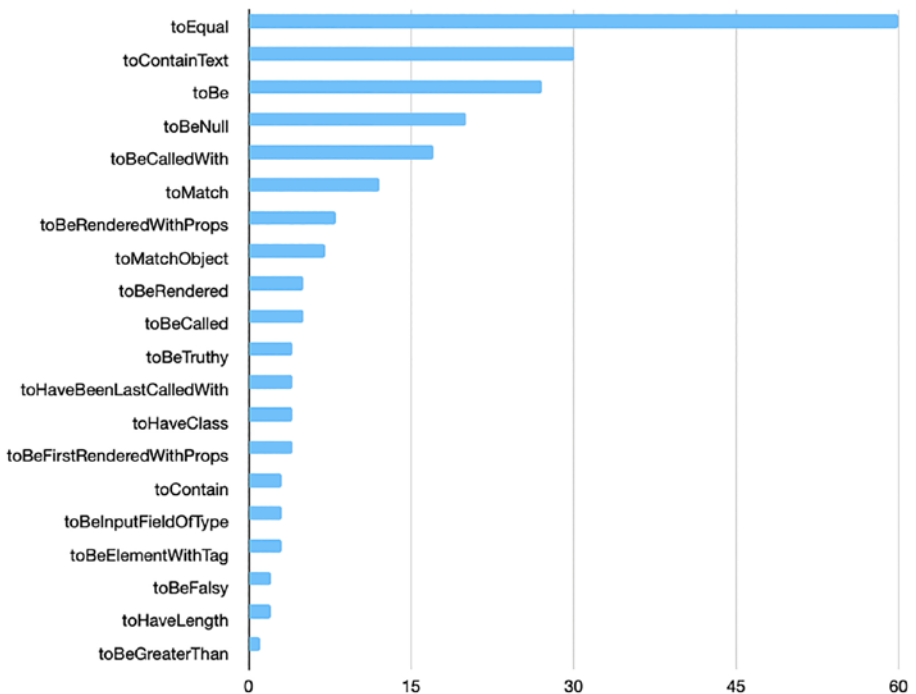
Figure 5-1. *Frequencies of matchers in a sample project*

Although most test runners come prepackaged with a wide variety
of matchers, it's also possible to write your own *user-defined matchers*,
and indeed this is often beneficial because you can write matchers in
your application's test suite that are aware of your own domain logic:
`toBeShopOrder`, `toContainTodoItems`, and so on.

Building the First Matcher: toBeDefined

We'll build these using a *proxy object* and a matchers module. When a
test calls `expect`, it returns a proxy object that forwards all calls on to the
matchers module. That module defines a set of exports with the same
names as the proxied calls but with an additional first parameter, which is
the parameter passed to `expect`.

Sounds like a lot? Conceptually I suppose, but thankfully it's not much code. Let's get going.

Create a new file in the `concise-test` project called `src/matchers.mjs`, with the following content.

Listing 5-1. concise-test/src/matchers.mjs

```
export const toBeDefined = (actual) => {
  if (actual === undefined) {
    throw new Error(
      "Expected undefined value to be defined"
    );
  }
};
```

In this implementation, I've made the error message more specific about the value it's complaining about. This particular matcher doesn't need a detailed message because the matcher name itself is very useful.

What *would* be useful in this case is just showing the expectation code from the test itself. For that we'll need access to the source file, which we'll come back to in Chapter 6.

First though, let's get this particular implementation working. Open `src/runner.mjs` and add an import for our new file, at the top. We'll use a named module import to pull in *all* the defined matchers at once. This way, the runner doesn't need knowledge of individual matchers:

```
import * as matchers from "./matchers.mjs";
```

Now at the bottom of the file, define `expect` in the following way. First, we define `matcherHandler`, which is a proxy handler that calls the right matcher, and then `expect` creates a new proxy object using that handler.

Listing 5-2. concise-test/src/runner.mjs

```
const matcherHandler = (actual) => ({
  get:
    (_, name) =>
    (...args) =>
      matchers[name](actual, ...args),
});

export const expect = (actual) =>
  new Proxy({}, matcherHandler(actual));
```

The named module export allows us to look up a matcher using
matchers[name]. The name value comes from the proxy object and will
be set to the name of the property that has been called. So if you call
expect(todo.completedAt).toBeDefined(), we will look up the matcher
matchers["toBeDefined"].

I've skipped ahead (again) here by including the ...args parameter.
In this case the matcher only needs one argument, the actual value,
but the matchers we'll write next will need a second value, which is the
expected value.

That means that all matchers have the same interface: the first
argument is the actual value of whatever is being expected, and the
remaining arguments match the arguments passed via the expect
invocation in the test.

Let's update the sample application tests to see if this works. In test/
tests.mjs, update the import to include expect.

Listing 5-3. todo-example/test/tests.mjs

```
import {
  beforeEach,
  describe,
```

```
  expect,
  it,
} from "concise-test";
```

Then update the first test to use the new matcher.

Listing 5-4. todo-example/test/tests.mjs

```
it("sets completedAt when calling markAsDone", () => {
  const todo = emptyTodo();

  expect(markAsDone(todo).completedAt).toBeDefined();
});
```

Rerun your tests; hopefully you get the same result as always. This would be a good point to test the failure message, too, so make the test fail by commenting out the line in `src/todos.mjs` that sets `completedAt`.

I've omitted file locations from my test output, but this is what I'm now getting:

```
Error: Expected undefined value to be defined
    at Module.toBeDefined
    at Proxy.<anonymous>
    at file:///...
```

This is useful but it's also noisy. We can cut out the lines of text that aren't relevant. However, before tackling that, I'd like some more information on what kinds of error messages *would* be useful. So let's write a second matcher, this time for the second test.

That's this one:

```
expect(() => repository.add(emptyTodo())).toThrow(
  new Error("title cannot be blank")
);
```

Back in the concise-test project, open src/matchers.mjs and add the following matcher.

Listing 5-5. concise-test/src/matchers.mjs

```
export const toThrow = (fn, expected) => {
  try {
    fn();
    throw new Error(
      `Expected ${fn} to throw exception but it did not`
    );
  } catch (actual) {
    if (expected && actual.message !== expected.message)
      throw new Error(
        `Expected ${fn} to throw an exception, but the thrown
        error message` +
          ` did not match the expected message.` +
          EOL +
          `  Expected exception message: ${expected.message}` +
          EOL +
          `    Actual exception message: ${actual.message}` +
          EOL
      );
  }
};
```

For this to work, you'll need to import EOL, which prints your platform's end-of-line character:

```
import { EOL } from "os";
```

> This matcher doesn't require an expected argument to be passed to it, in which case it only confirms that an error was thrown, but not the message or type. See the "Exercises" section to continue this work.

In the sample application, update `test/tests.mjs` with this new version of the second test.

Listing 5-6. todo-example/test/tests.mjs

```
it("throws an exception when adding a todo without a
title", () => {
  expect(() => repository.add(emptyTodo())).toThrow(
    new Error("title cannot be blank")
  );
});
```

While you're at it, you may as well update the third test because it uses the same matcher.

Listing 5-7. todo-example/test/tests.mjs

```
it("throws errors when adding a repeated todo", () => {
  repository.add(newTodo);
  const repeatedTodo = { ...newTodo };
  expect(() => repository.add(repeatedTodo)).toThrow(
    new Error("todo already exists")
  );
});
```

To ensure these fail, go into `src/TodoRepository.mjs` and add some nonsense characters into the error messages. Then run tests so you can see the failure. Here's what I'm getting:

```
Error: Expected () => repository.add(repeatedTodo) to throw an
exception, but the
thrown error message did not match the expected message.
  Expected exception message: todo already exists
    Actual exception message: 123todo already exists
```

This is a lot of text. Colors will be useful in making values stand out, such as using green for the expected value and red for the actual value.

It's also useful to note that in this case, the actual value is *computed* by our matcher, rather than being directly passed in as a value. We're lucky that printing a lambda expression returns that actual source text. It makes the matcher output more readable: we can see the *source* in addition to the *actual* and *expected* values. If every failure message looked like that, we'd be winning.

Let's try to improve this for our third matcher. Change `src/matchers.mjs` to import `colors`:

```
import { color } from "./colors.mjs";
```

Now write the third matcher, `toHaveLength`.

Listing 5-8. concise-test/src/matchers.mjs

```
export const toHaveLength = (actual, expected) => {
  if (actual.length !== expected) {
    throw new Error(
      color(
        `Expected value to have length <bold>${expected}</bold>
        but it was` +
          ` <bold>${actual.length}</bold>`
      )
    );
  }
};
```

Back in `test/tests.mjs`, update the final two tests to use this new matcher.

Listing 5-9. todo-example/test/tests.mjs

```
it("finds an added todo", () => {
  expect(repository.findAllMatching("")).toHaveLength(1);
});

it("filters out todos that do not match filter", () => {
  expect(
    repository.findAllMatching("some other test")
  ).toHaveLength(1);
});
```

To test these, modify the tests with an error value–for example, change `toHaveLength(0)` to `toHaveLength(999)`. Here's the output I get:

```
Error: Expected value to have length 0 but it was 999
```

Although it won't show in the preceding printed code snippet, the bold text really helps here. Also we can see that the error message format doesn't match the one for `toThrow` or indeed `toBeDefined`, so it feels important that all error messages have the ability to craft their own message format.

That being said, let's pull out an abstraction.

Creating an Error Subtype

Looking at the last matcher, the biggest issue I have is that there are two methods of inserting data into the string–there's our application-specific `color` function syntax, and then there's also standard string interpolation. So any new subclass of `Error` should solve that.

Node already has its own subclass just like this, called
`AssertionError`. But where's the fun in using that? Let's build
our own.

In addition, the stack traces for these expectation failures are not really
all that useful. They are indeed still useful for non-expectation failures,
since in those cases the failure is unexpected and the the programmer will
want to know where the problem lies.

For expectations, what's useful is the source code for the expectation
itself: both the value that lies between `expect(` and `)` and the matcher
together with the values passed to it.

Create a new file in the `concise-test` project, called `src/`
`ExpectationError.mjs`. Give it the following content.

Listing 5-10. concise-test/src/ExpectationError.mjs

```
import { color } from "./colors.mjs";

export class ExpectationError extends Error {
  constructor(message, { actual, expected, source }) {
    super(
      "Expected " +
        color(
          message
            .replace("<actual>", `<red>${actual}</red>`)
            .replace(
              "<expected>",
              `<green>${expected}</green>`
            )
```

```
        .replace("<source>", `<bold>${source}</bold>`)
      )
    );
  }
}
```

Now update `src/matchers.mjs` to use this new class in each of the matchers. You can also remove the `colors` import at the top of the file.

Before looking at the solution given in the next paragraph, go ahead and try yourself first, and make sure to watch the tests fail before you consider yourself done.

Here are the updates I made.

Listing 5-11. concise-test/src/matchers.mjs

```
throw new ExpectationError("<actual> to be defined", {
  actual,
});
```

```
throw new ExpectationError(
  "<source> to throw exception but it did not",
  {
    source,
  }
);
```

```
throw new ExpectationError(
  "<source> to throw an exception, but the thrown error message
   did not match the" +
    " expected message." +
    EOL +
    "  Expected exception message: <expected>" +
    EOL +
    "    Actual exception message: <actual>" +
```

```
    EOL,
  {
    source,
    actual: actual.message,
    expected: expected.message,
  }
);

throw new ExpectationError(
  "value to have length <expected> but it was <actual>",
  { actual: actual.length, expected }
);
```

Allowing Multiple Failures per Test

For the final part of this chapter, let's extend our test runner to capture multiple test failures per test. This is useful because if you're formatting your tests correctly, then all calls to expect come after the *Act* phase of your test. None of the calls to expect should change program state. That means their pass or fail results are independent of each other. Because of that, even if they all fail, the information about each failure is still valid.

Long story short: If your test has three failing expectations, you want all three results, not just the first.

We'll do this by suppressing expectation exceptions within the expect proxy handler. That way, non-expectation exceptions will still cause our test to bomb out, and they will still be recorded as errors.

But now we have a design issue. Does each failure count as a separate test failure? Or should a single test have its failures grouped up and reported as a single failure?

Making a Mess

Recall that the `failures` array from Chapter 4 holds all the test cases that have failed. Each item in this array has an `error` property that is the `Exception` object that was caught.

In this section, we'll convert the `error` property (which holds the `Exception` object) into an `errors` property, which will hold an array of `Exception` objects. Every time an exception occurs, we'll push the exception into this array, but then continue on with processing the test.

We'll need to make sure that the `matcherHandler` function no longer throws these exceptions, but instead collects the errors in a *holding object* that can be initialized before running any expectation. This object is named `currentTest`. It can then be placed into the `failures` array later on should it contain any exceptions. Otherwise, the holding object is discarded.

This implementation is starting to feel messy to me. Perhaps it is for you too. We have an array within an array, which is a bit ugly. Don't worry. We're going to overhaul it in Chapter 8 when we split out the runner into different phases.

In `src/runner.mjs`, start by defining a new module-level variable, just under your definition of `successes`:

```
let currentTest;
```

Create a new function named `makeTest`, just above the definition of `it`.

Listing 5-12. concise-test/src/runner.mjs

```
const makeTest = (name) => ({
  name,
  errors: [],
  describeStack,
});
```

Rewrite the it function to set the currentTest variable when it starts
and to push failures to the errors array instead of directly to failures and
then, if we've collected any errors at all during the test run, push the whole
object to failures.

Listing 5-13. concise-test/src/runner.mjs

```
export const it = (name, body) => {
  currentTest = makeTest(name);
  try {
    invokeBefores();
    body();
    invokeAfters();
  } catch (e) {
    currentTest.errors.push(e);
  }
  if (currentTest.errors.length > 0) {
    console.log(
      indent(color(`<red>${cross}</red> ${name}`))
    );
    failures.push(currentTest);
  } else {
    successes++;
    console.log(
```

```
      indent(color(`<green>{$tick}</green> ${name}`))
    );
  }
```

That change will have broken our application; we need to update `printFailure` to deal with multiple `errors` rather than a single `error`. It's only the middle line of the function that changes.

Listing 5-14. concise-test/src/runner.mjs

```
const printFailure = (failure) => {
  console.error(color(fullTestDescription(failure)));
  failure.errors.forEach((error) => {
    console.error(error);
  });
  console.error("");
};
```

That's our existing refactoring complete. Now, to support multiple expectation failures requires changing `matcherHandler`. First, import `ExpectationError` at the top of the file:

```
import { ExpectationError } from "./ExpectationError.mjs";
```

Then update `matcherHandler`.

Listing 5-15. concise-test/src/runner.mjs

```
const matcherHandler = (actual) => ({
  get:
    (_, name) =>
    (...args) => {
      try {
        matchers[name](actual, ...args);
      } catch (e) {
```

```
      if (e instanceof ExpectationError) {
        currentTest.errors.push(e);
      } else {
        throw e;
      }
    }
  },
});
```

This is the second place that we push to `currentTest.errors`; the `it` function already does that. You might wonder why we collect the failures for non-`ExpectationError`s in here too. The reason is we'd risk double-counting them: the `it` function *must* record to `currentTest.errors` in order to catch errors from `beforeEach` blocks, `afterEach` blocks, and the test function itself. So we must be careful here not to push those errors in `matcherHandler`.

That's undoubtedly a behavior that would be nice to have as an automated test!

Testing Your Solution

There are two new behaviors to test: first, that expectation failures are grouped, and second, that non-expectation exceptions cause the test to bomb out.

To test the first, go to `test/tests.mjs` and add a nonsense expectation at the end:

```
expect(markAsDone(todo).completedAt).toBeDefined();
expect(markAsDone(todo).foobar).toBeDefined();
```

Running your tests now should show that there's still only one test failure, but two expectations are printed out.

To test the second behavior, add the following line to the top of the first test in `test/tests.mjs`:

```
throw new Error("foobar");
```

When you run tests, you should see this error printed, but neither of the expectations has been run.

I'm starting to wish we'd followed TDD. Knowing that we have a big refactor coming, we'll have to manually retest *all* of this.

Exercises

1. Write a utility like the one created in Chapter 2, Exercise 1, that counts and lists the frequencies of matchers used within your test suites. It should output CSV data that can be imported into a charting program and used to produce a graph like the one in Figure 5-1.

2. Update the `toThrow` matcher to fail if the `Error` subtype doesn't match the one defined in the test.

3. Reimplement test failures to work without relying on exceptions. Instead, every matcher should return some value that can then be collected by the matcher handler and dealt with appropriately.

4. Some test frameworks offer *negated* matchers, for example:

```
expect(true).not.toEqual(false);
```

Given your solution to Exercise 3, go ahead and implement `not`.

5. Think of some of the matchers you've used with Jest or other test frameworks that support expectations. Pick one and reimplement it within your test framework. If you have time, carry on with implementing more.

6. If the developer accidentally mistypes a matcher name or attempts to use a matcher that isn't registered, the test runner should helpfully inform them of the problem. Implement this feature.

Discussion Questions

1. Common development wisdom tells us that exceptions should be used for exceptional program behavior–meaning *unexpected* behavior–and not failure scenarios that are *expected* behavior. Do you think our use of exceptions in the test framework is valid use for them? What are the pros and cons of this approach vs. relying on any other mechanism?

2. If you hadn't implemented your solution in Exercise 3 and your system still relied on throwing exceptions, how would you have implemented negated matchers?

Summary

In this chapter you've added the `expect` function and three matchers: `toBeDefined`, `toThrow`, and `toHaveLength`.

You've learned the importance of user-defined matchers and how they can be used to simplify and improve the maintainability of your tests.

In the next chapter, we'll zone in on how test failures are reported.

PART II

Constructing a Usable Framework

In this part, you will extend the test framework to include ergonomic features that make the `concise-test` runner into a viable competitor to mainstream test frameworks.

In Chapter 6, "Formatting Expectation Errors," we write a formatter for stack traces to help pinpoint failures quickly.

In Chapter 7, "Automatically Discovering Test Files," we'll add test file discovery to our test runner and add support for running a single file through a command-line argument.

In Chapter 8, "Focusing on Specific Tests," we split out the runner into two phases: a discovery phase and an execution phase. In between them, we insert a filter phase to support running only a subset of tests.

In Chapter 9, "Supporting Asynchronous Tests," we add the ability to wait on tests that return `Promise` objects and timing out tests with `it.timesOutAfter`.

In Chapter 10, "Reporting," we use a pub-sub model to build a plugin system for reporters.

CHAPTER 6

Formatting Expectation Errors

One of the signs of a "good" test is that when it fails, it very quickly pinpoints to you *why* it failed so that you can get to fixing the problem quickly.

To that end, our current expectation errors are not as helpful as they could be. In Chapter 5, you saw how matchers are used to output "pretty" expectation failure messages. However, when a failure occurs, we'll *also* get an exception stack trace printout, and it's this that we can improve on.

In this chapter, we'll take the stack trace and rework it into something that's directly useful to your test runs.

By the end of the chapter, you'll have seen how to dig into the V8 API, which underpins Node, to extract the constituent parts of a stack trace and format them appropriately for a terminal.

Utilizing Stack Traces with Testing Workflows

A standard stack trace includes a list of *call sites* that highlight each entry in the call stack where an exception originated from. Each call site is made up of four parts:

- A function name, if the call site is within a function

- A file path, if the function was defined within a script

- A line number

- A column

For expectation errors, only *one* of these call sites is ever useful: the call site where the user's test code calls an `expect` matcher. All other call sites will be within the test runner's call sites, and they are not meaningful at all to the user.

Figure 6-1 shows an example of how each line of a stack trace relates to a specific point in the code: the first item being with the `concise-test` framework and the second line being within the test suite code itself. It's this second line that we're most interested in as it pinpoints where the expectation that failed.
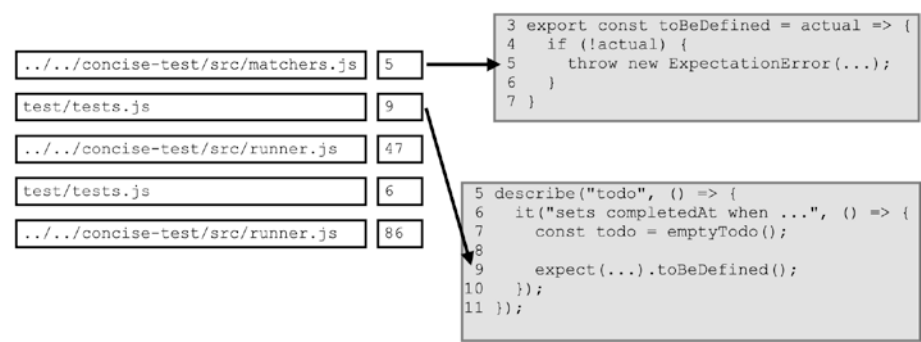

Figure 6-1. *How our stack trace looks with expectation errors*

With this in mind, here's a list of specific things we can tackle in this chapter:

- As stated previously, only list the *first* call site that is within the user's test code.

- Format the file path as a relative path so it's easier to read.

- Add colors to improve readability.

102

- Include a segment of text from the original source file that shows the surrounding context of where a failure occurred.

- Use a ^ (caret) character to highlight where the failure occurred.

The last part is something I first saw from the Jest test runner,[1] which I really like, because it acts as a backup explanation of why a test failed, if the expectation message isn't enough.

For example, take our `toBeDefined` matcher. In the case of an undefined value–which is the only time that this matcher will break–the expectation message will be

```
Expected undefined to be defined
```

This is unhelpful to say the least. But add in a stack trace, and suddenly it's more understandable:

```
> todo-example@1.0.0 test
> concise-test

todo
  ✗ sets completedAt when calling markAsDone
TodoRepository
  add
    ✓ throws an exception when adding a todo without a title
    ✓ throws errors when adding a repeated todo
  findAllMatching
    ✓ finds an added todo
    ✓ filters out todos that do not match filter
```

---

[1] https://jest.io

```
Failures:

todo → sets completedAt when calling markAsDone
Expected undefined to be defined

  in test/tests.mjs:

   7 |      const todo = emptyTodo();
   8 |
   9 |      expect(markAsDone(todo).completedAt123).
            toBeDefined();
     |                                              ^
  10 |    });
  11 | });

4 tests passed, 1 tests failed.
```

We'll use a slightly different development approach to this work. Rather than break it down into multiple small features and implement those in turn, we're going to build out a stackTraceFormatter module and write it out from top to bottom.

I didn't initially write it this way, but instead made a mess and then worked it into a shape that I'm happy with. I'm sparing you that process and showing you the end result.

I think this is beneficial because it highlights the functional style of coding at its best: lots of little functions that tell a story that flows down the page, starting from small utility functions and ending with an exported function that pulls everything together.

If you've never seen this style before, it can feel quite different. I love writing code this way.

Building stackTraceFormatter

We're going to build a source file that ends with a single function: formatStackTrace. To get there we have to write a bunch of utility functions. In total, these functions perform the following actions:

- Filter the given stack trace to remove any call sites that originated within three locations: the test runner project, node_modules, and Node internals.

- From the remaining call site entries, take the first entry. This should relate to the place where an expect matcher was called.

- Read that source file and filter it to the five lines that surround the matcher invocation that failed.

- Return first an "intro" line that includes just a file location that is relative to the project root.

- Then return the five source lines, including line numbers and a caret location pointing to the column location of the call site.

- Indent all of the returned content by two characters to set it apart from the failing test name and expectation message.

Create a new file in your test runner project at the location src/stackTraceFormatter.mjs. Start by adding the imports:

```
import path from "path";
import fs from "fs";
import { EOL } from "os";
import { color } from "./colors.mjs";
```

Then, define the first function, `findFailureCallSite`. It needs a constant called `ignoredFilePatterns`, which we'll define just above it.

Listing 6-1. concise-test/src/stackTraceFormatter.mjs

```
const ignoredFilePatterns = [
  "/node_modules/",
  path.dirname(new URL(import.meta.url).pathname),
  "^internal/",
];

const findFailureCallSite = (stack) =>
  stack.find((callSite) => {
    const fileName = callSite.getFileName();
    return (
      fileName &&
      !ignoredFilePatterns.some((pattern) =>
        fileName.match(pattern)
      )
    );
  });
```

This function is relying on the V8 `CallSite` API (https://v8.dev/docs/stack-trace-api). The `stack` parameter is a list of `CallSite` objects. Each one of those has a `getFileName` function that returns either `null` or a file path.

But what we want is the *first* call site from within the set of test source files. That won't be the first in the array because that will be somewhere in the test framework's code.

Here's how we do it–using `Array.find`, we take the first call site that doesn't match any of the regular expressions in `ignoredFilePatterns`:

- /node_modules/ is self-explanatory and removes any path that has a directory named node_modules within it. This will catch any source folders that happen to be called node_modules, but if you're naming your source folders that, then you are asking for trouble.

- The call to path.dirname uses the import.meta.url value to get the directory of this currently executing file (in other words, stackTraceFormatter.mjs). This allows us to remove any files that are within the test runner directory. This is useful when you're running the test runner via a local package, just as we are in this book. For anyone pulling the package from NPM, this case should be covered by the /node_modules condition.

- The final entry removes any Node internal entries, which appear as files within the internal/ directory.

The V8 engine offers a function that *could* have been useful to us: Error.captureStackTrace. This function is used to cut out any call sites that occur *after* a given call site. So, for example, we could have called Error.captureStackTrace(it) where it is a reference to the it function in src/runner.mjs. This would have blocked out any call stack entries that came before the user's call to it. However, this doesn't buy us anything, because have to filter out call stack entries *above* the expectation call in addition to ones below it. Using Error.captureStackTrace would have meant adding more code, not removing any.

107

Next up, let's convert anyway from the V8 `CallSite` API to something a little more user-friendly.

Listing 6-2. concise-test/src/stackTraceFormatter.mjs

```
const relative = (fileUrl) =>
  path.relative(
    process.cwd(),
    fileUrl.replace("file://", "")
  );

const getFailureLocation = (stack) => {
  const failureLocation = findFailureCallSite(stack);
  if (failureLocation) {
    return {
      fileName: relative(failureLocation.getFileName()),
      lineNumber: failureLocation.getLineNumber(),
      column: failureLocation.getColumnNumber(),
    };
  }
};
```

This returns us a plain ol' JavaScript object with three properties: `fileName`, which is converted to a relative location to the current working directory; `lineNumber`, which is the line number of the call site within that file; and `column`, which is the column number of the call site within that line.

I prefer the name `column` over `columnNumber` because I think it's clear when you write `column` that you're referring to a number. It wouldn't make sense any other way. The same isn't true for `lineNumber`: `line` could mean a line of text. You could argue that this isn't consistent. Feel free to call it `columnNumber` if you prefer.

Next up, we've got some functions to format a set of file lines with line numbers and a caret (^) pointing at the column failure. As a reminder, here's what we're aiming for:

```
 7 |     const todo = emptyTodo();
 8 |
 9 |     expect(markAsDone(todo).completedAt123).
         toBeDefined();
   |                                           ^
10 |   });
11 | });
```

To start with, let's define a function `pipeSeparatedValues` that joins a bunch of columns with a pipe (|) separator:

```
const pipeSeparatedValues = (...columns) =>
  columns.join(" | ");
```

Something interesting about this definition is that it'll work for any number of columns, not just two. But we only needed it to work for two columns. Have I over-engineered this?

I like to write the most general form of a function that is still straightforward to describe. In this case, I can liken "pipe-separated values" to comma-separated values.[2] So I can argue that this definition is "simpler" but only if I assume that you're familiar with CSV.

General functions like this have the advantage that they can be reused should the need arise.

---

[2] https://en.wikipedia.org/wiki/Comma-separated_values

Now let's define a function `withLineNumbers` that adds line numbers to a bunch of lines. It takes two parameters, the `lines` array and the `start` line number, and returns a new array of lines, which is a set of lines formatted with `pipeSeparatedValues`.

Listing 6-3. concise-test/src/stackTraceFormatter.mjs

```
const withLineNumbers = (lines, start) => {
  const numberColumnWidth = (
    lines.length + start
  ).toString().length;

  return lines.map((line, index) => {
    const number = (start + index)
      .toString()
      .padStart(numberColumnWidth);
    return pipeSeparatedValues(number, line);
  });
};
```

This code uses the variant of `map` that passes the index of each value as the second parameter to the map function. We use that to print out the line number, offset from the `start` line number.

The most interesting thing about this code is that it right-aligns the numbers in the line number column. It does that by first calculating the width of the largest number—that's `numberColumnWidth`—and then it uses `padStart` to ensure each line number has that width.

Then we'll define `pointerAt`, which builds the line with the caret pointer.

Listing 6-4. concise-test/src/stackTraceFormatter.mjs

```
const pointerAt = (column, maxLineNumber) => {
  const padding = maxLineNumber.toString().length;

  return pipeSeparatedValues(
    " ".repeat(padding),
    `${" ".repeat(column - 1)}<bold>^</bold>`
  );
};
```

The complication here is that this function needs to have knowledge of the width of the first column. It gets that by way of the maxLineNumber being passed in. This is clearly different from how withLineNumbers did it, which takes a start line number and adds on the index of each line from lines, but neither lines nor start is useful to us here in this function, so there's no point passing them in just to be consistent.

There are almost certainly other, perhaps clearer, ways to share knowledge of this width between withLineNumbers and pointerAt, perhaps by creating some kind of table definition object that specifies the width of each line, but this implementation is good enough for me.

Another place this could be useful is for lines that extend over the width of the current terminal window. I don't tend to hit this issue because I like my source code lines to be short, but any long line will cause the terminal to wrap its content. For me this is so much of an edge case that I don't want to bother spending any time fixing it. But if you disagree, feel free to extend the implementation here at this point.

Okay, one more helper function before we start joining things up:

```
const boundedSlice = (array, from, to) =>
  array.slice(
    Math.max(from, 0),
    Math.min(to, array.length - 1)
  );
```

This modifies the `Array.slice` function so that it doesn't wrap. We want to take the five lines that surround the call site–two lines above and three lines below–but what happens if the error occurs on line 1? In that case we'd only want to grab lines 1–4. If we use the regular `Array.slice` function, it will wrap around, so `boundedSlice` makes sure that doesn't happen. Similarly, if an exception occurs on the last line of a source file, we don't want to send invalid line numbers to `slice`. `boundedSlice` takes care of that for us.

Note again that `boundedSlice` is a generalized form of a function that could be used elsewhere in another context, should the need arise.

Let's make use of all the functions we just defined to format the source.

Listing 6-5. concise-test/src/stackTraceFormatter.mjs

```
const highlightedSource = ({
  fileName,
  lineNumber,
  column,
}) => {
  const allLines = fs
    .readFileSync(fileName, { encoding: "utf8" })
    .split(EOL);
```

```
  const fromLine = lineNumber - 3;
  const toLine = lineNumber + 2;

  const highlightedLines = withLineNumbers(
    boundedSlice(allLines, fromLine, toLine),
    fromLine + 1
  );

  return [
    ...highlightedLines.slice(0, 3),
    pointerAt(column, toLine),
    ...highlightedLines.slice(3, 5),
  ];
};
```

This function takes as input a failure location as defined by `getFailureLocation`. It then reads that file and builds a set of formatted lines, with line numbers and a pointer line added in.

We've now achieved the majority of what we set out to do. Here's what's left:

- Adding in an intro line

- Indenting everything by two characters

Start by defining `indentLine`, which very simply prepends two spaces:

```
const indentLine = (line) => `  ${line}`;
```

There are a couple of issues with this tiny function. The first is that the knowledge that an "indent" is two spaces is not clear, probably because it's hard to count (or even notice!) empty spaces. It might be better for us to have an explicit `const indentSize = 2` and then use the `String.repeat` function to build that indent, just as we've done before in `runner.mjs`.

The second is that, yes, we've already built something similar in `runner.mjs`. Perhaps there's some refactoring of that code to use a general `indentLine` function? Again, I'm leaving that as an exercise for you! I've declined to do it because I think these functions are so short and simple that it's almost not worth trying to find a more general form.

Finally then, let's put it all together.

Listing 6-6. concise-test/src/stackTraceFormatter.mjs

```
export const formatStackTrace = (error, stack) => {
  const failureLocation = getFailureLocation(stack);
  if (!failureLocation) return;

  const { fileName } = failureLocation;

  const introLine = `in <bold>${fileName}</bold>:`;

  const allLines = [
    "",
    introLine,
    "",
    ...highlightedSource(failureLocation),
  ];

  return color(allLines.map(indentedLine).join(EOL));
};
```

There we have it. Beautiful! Perhaps the most complicated piece of this is the guard clause that returns nothing if a failure location isn't found. That might happen, for example, if we have an unexpected error in the test runner itself. In that case we shouldn't bother showing the user anything, because it's not an error with their own code.

Joining Up Our Formatter with the Runner

Finally then, we need to pull this code into `src/runner.mjs`. Add the following import at the top of the file:

```
import { formatStackTrace } from "./stackTraceFormatter.mjs";
```

Just below that, add the following line:

```
Error.prepareStackTrace = formatStackTrace;
```

This allows us to overwrite the mechanism by which an array of `CallSite` objects is converted to a string. With this line in place, any time an error occurs in the runner, it'll run through our formatter instead.

There's a subtle issue with this change; if you run `npm test` now with an intentional failure, you'll see that the error `message` is no longer printed, only the stack trace.

We need to update our use of `console.error` to explicitly print both the `error` and the `stack` properties.

In `src/runner.mjs`, update `printFailure` as shown in the following.

Listing 6-7. concise-test/src/runner.mjs

```
const printFailure = (failure) => {
  console.error(color(fullTestDescription(failure)));
  failure.errors.forEach((error) => {
    console.error(error.message);
    console.error(error.stack);
  });
  console.error("");
};
```

And in addition to that, also change the use of `console.error` in the run function.

Listing 6-8. concise-test/src/runner.mjs

```
export const run = async () => {
  try {
    ...
  } catch (e) {
    console.error(e.message);
    console.error(e.stack);
  }
  ...
};
```

There *is* a default `Error.prepareStackTrace` defined when our Node application starts. You could, if you wish, extend `formatStackTrace` to call this original function when it can't find a failure location, rather than simply returning undefined.

And that's it. Run your tests–again making sure you have something set up to fail–and you should see the new format in action.

If you've made any mistake in your error-reporting code, it will unfortunately block any errors from being reported. If this happens to you, I suggest you wrap the `formatStackTrace` body in a try-catch and use `console.error` to print out the exception message. See Exercise 2 if you'd like to work on fixing this issue.

Exercises

1. Implement a new *interactive* mode (meaning it waits for input from you before quitting) for your test runner that allows the user to explore the stack trace of a failure. Use Figure 6-1 as a guide for how this should work: display a table of file names and line numbers. The user can select a row from this table to show the function definition at this point, right within the terminal.

2. We used the `Error.prepareStackTrace` function to override stack traces in a way that works for test failures. However, this function is now used for *all* errors, including runtime errors like syntax issues or missing imports. Change the design so that it uses the built-in `Error.prepareStackTrace` function for anything other than test failures.

3. Look through the V8 API for call frames to determine what additional information you might want to output when a test fails. Experiment with the test runner by implementing any of the ideas you uncover.

Discussion Questions

1. One of the worst problems a developer can have is to be spending time tracking down *why* any given test has failed. In an ideal world, a test failure would itself make it immediately obvious why it has failed. What strategies can you use when writing tests to ensure that they're quick to pinpoint the area of application code that is causing a test to fail?

2. How useful is it to have contextual information,
    like the source code you printed out in Exercise
    1, available to you within the test runner? If your
    test runner was integrated into your IDE, what
    contextual information do you think would be
    useful to you?

3. Imagine you were able to print out the value of
    variables within the current scope, at the point of
    a test failure. Do you think this would be useful to
    see as part of the test runner output or would be
    too *noisy*? Think through some examples to help
    you decide.

Summary

In this chapter we took a little journey into printing better expectation
failures, digging deep into the V8 API.

That just about finishes up the `expect` function. In the next chapter,
we'll switch focus away from the details of a test case and start generalizing
the test runner, starting with discovery of test files.

Automatically Discovering Test Files

Up to this point, our tests have had to exist within the `test/tests.mjs` file. This is not ideal. Surely we want the ability to place our test suites wherever we want.

In this chapter you'll learn

- Typical testing workflows during local development

- How to run multiple test suites sequentially

- How to run just a single, specific file by including the file path in the launch command

Let's begin with a look at some common usage patterns when running test runners locally.

Development Workflows and Test Runners

We know that we can run tests with this command:

```
npm test
```

This command of course runs *all* our tests. That has a couple of disadvantages: First, it can be slow, and so it makes it hard to integrate into any personal workflow that requires running the command frequently. Second, if you're busy making changes to a codebase, it may end up returning a *lot* of failures if you're mid-refactor, and at that point you probably aren't interested in *all* the failures. That's just noise.

The primary occasion that it's critical to run the full test suite is on any Continuous Integration (CI) environment before any change is accepted into a production branch.

Beyond this usage of the test runner, there are alternatives.

The most obvious is to run a single file:

```
npm test foo.tests.mjs
```

In general, a single test suite is going to take a short amount of time to run–maybe less than a minute, to put a figure on it. That means it can be used as a part of a personal workflow to get feedback quickly on code changes, be it new feature work or refactoring. It works really well with test-first styles of development, like Test-Driven Development (TDD). This is a command variant that I use most in my day-to-day work.

When you're running tests in just a single file, you are purposefully excluding other test suites, which may or may not contain failures. So it's always important to follow up with a full test run at some point–perhaps before you commit your changes.

You could also specify a set of files on the command line:

```
npm test foo.tests.mjs bar.tests.mjs
```

This command is useful if you know there's a set of test suites that act on the area of the production code you're working on.

A variant of this is to set a *tag* on each of the test suites that you're interested in and then run all test suites with that tag. For more on this approach, see Chapter 12.

In Chapter 8 we'll look at another method of selecting a specific set of files, with the `only` modifier that can be applied to the `describe` and `it` functions.

What About Watch Mode?

Some test runners can be started in *watch* mode that will keep the test runner "live," listening for any changes to the test files and production code.

I don't find this useful for a couple of reasons:

- Often I like to see the last failure on-screen while I'm working on fixes. I don't want it to disappear on me if I happen to save the file I'm working on and the test run to start again. I want that test output to stay put until I've finished with my set of changes.

- Reloading test and production code in a running process is not simple. Some changes get missed, and tests are no longer running from a clean slate. I never feel like I can fully *trust* watch mode–and trust is critically important for any test runner.

What I find works better for me is

- Using a screen windowing setup that allows me to see my application code, test code, and test runner output all at the same time

- Making it very easy to switch between my editor and the terminal so I can rerun the test command with just one or two keystrokes

If you haven't used this kind of watch mode before, it's worth trying it out. You might like it!

Discovering Files

With all that said, let's start making a change to our test runner.

At the moment, when the `npm test` command is invoked, the test runner loads tests in a single file, `test/tests.mjs`. Let's modify our test runner to find all files that match the pattern `test/*.tests.mjs`.

In `src/runner.mjs`, first create a new import for `fs`:

```
import fs from "fs";
```

Then create a new function called `discoverTestFiles`, just above `run`.

Listing 7-1. concise-test/src/runner.mjs

```
const discoverTestFiles = async () => {
  const testDir = path.resolve(process.cwd(), "test");
  const dir = await fs.promises.opendir(testDir);
  let testFilePaths = [];
  for await (const dirent of dir) {
    if (dirent.name.endsWith(".tests.mjs")) {
      const fullPath = path.resolve(
        dir.path,
        dirent.name
      );
      testFilePaths.push(fullPath);
    }
  }
  return testFilePaths;
};
```

Finally, update `run` to import each of these test files and await the result of each. We use `Promise.all` to wait for each individual dynamic import to have succeeded.

122

Listing 7-2. concise-test/src/runner.mjs

```
export const run = async () => {
  try {
    const testFilePaths = await discoverTestFiles();
    await Promise.all(
      testFilePaths.map(async (testFilePath) => {
        await import(testFilePath);
      })
    );
  } catch (e) {
    console.error(e.message);
    console.error(e.stack);
  }
  printFailures();
  console.log(
    color(
      `<green>${successes}</green> tests passed, ` +
        `<red>${failures.length}</red> tests failed.`
    )
  );
  process.exit(
    failures.length > 0
      ? exitCodes.failures
      : exitCodes.ok
  );
};
```

That's it. Time to test. In your sample application, split `test/tests.mjs` into two files: `test/todo.tests.mjs` and `test/TodoRepository.tests.mjs`, each one with its own describe block.

With any luck, all your tests are still found and still passing.

Running a Single File

Sometimes it's nice to just run the tests in a single file–perhaps because your full test suite takes a while to run. So let's add that support by allowing the user to run a single file on the command line, like this:

```
npm test test/todo.tests.mjs
```

Running File Sets

It's often useful to run a specific set of files together because they test a similar set of functionality. If you have a file that often changes when a certain other file changes—that is, they are coupled in some way—then it's useful to run tests for both of them together.

In the JavaScript world, this might happen with UI components and reducers, which change in lockstep. In the Rails world, this often happens where one logical entity has a model, a controller, and a serializer and all should be run.

I don't think the command line is the right place to specify these file sets. Instead, these connections should be written into your tests. Tagging test suites is one way to do it, which we'll implement in Chapter 12.

Back in your test runner, create the following functions, just above `run`.

Listing 7-3. concise-test/src/runner.mjs

```
const isSingleFileMode = () => process.argv[2];

const getSingleFilePath = async () => {
  const filePathArg = process.argv[2];
  try {
```

```
    const fullPath = path.resolve(
      process.cwd(),
      filePathArg
    );
    await fs.promises.access(fullPath);
    return [fullPath];
  } catch {
    console.error(
      `File ${filePathArg} could not be accessed.`
    );
    process.exit(exitCodes.cannotAccessFile);
  }
};
```

We'll use these functions in `discoverTestFiles`, but there's one more thing we need to do. There's a try-catch in `getSingleFilePath`: it protects us against mistyped file paths. If the file passed in can't be found or can't be read, we exit the process with a new error code.

We didn't bother to do a similar check in our original `discoverTestFiles` implementation. Why not? It's down to the fact that `getSingleFilePath` takes user input and uses that as the basis of its search. It's very easy to mistype a file path, so it's likely to be quite common for this error to occur. That's entirely different from `discoverTestFiles`, which asks the file system for a directory listing.

In the definition of `exitCodes`, add in a third entry:

```
const exitCodes = {
  ok: 0,
  failures: 1,
  cannotAccessFile: 2,
};
```

Let's create a new function `chooseTestFiles`:

```
const chooseTestFiles = () =>
  isSingleFileMode()
    ? getSingleFilePath()
    : discoverTestFiles();
```

Hook it into `run` by switching out the call to `discoverTestFiles` with a call to `chooseTestFiles`:

```
const testFilePaths = await chooseTestFiles();
```

Time to test. Switch back to your sample application directory and run the command `npm test test/todo.tests.mjs`. You should see just one test running:

```
> todo-example@1.0.0 test
> concise-test test/todo.tests.mjs

todo
  ✓ sets completedAt when calling markAsDone
1 tests passed, 0 tests failed.
```

Now try running `npm test test/notfound.tests.mjs`. What happens?

Exercises

1. Update your code to handle *symlinks*, particularly symlinks that link back to parent folders and would cause a cycle to occur within your test discoverability code.

2. Update the test runner to take multiple file paths, and run each of them individually.

3. Update the test runner to work with globs. For example, the following command should work:

    ```
    npm test tests/**/Foo*
    ```

4. Implement a source code analysis feature that uses module imports to determine which application code files are covered by tests and which aren't. Use this to report to the user which application code files do not have any test coverage. (Assume that importing a file within a test suite is enough to provide full coverage.)

Discussion Question

1. Our test application has a `tests` directory that is separate from the application code. You could also colocate your tests with the application code so that they appear in the same directory. Another approach would be to have the tests appear in the exact same file as the application code. What are the pros and cons of each of these approaches? What do you prefer? Is it simply a personal preference, or is there an objective reason as to why one approach is better than the other? Does your answer depend on the programming language?

Summary

In this chapter we've started to explore support for test suites spanning multiple files.

We also only have the beginnings of a command-line parser.

A problem we now face is that `src/runner.mjs` has a bunch of new functions for handling files. They'd be better off in a namespace of their own. We'll tackle that in the next chapter by rebuilding our test functions within their own module.

CHAPTER 8

Focusing on Specific Tests

The core of our system is essentially complete. We can now write complete test suites and get beautiful reports on-screen.

But there's one **BIG** unanswered question that'll require a serious rethink: How do we a run a *single* test?

This is known as *focusing* on one (or more) test or describe block. The test runner does this by only running tests that are created using `it.only` rather than `it`. If no tests are marked as `.only`, then we run all the tests, just as we do now.

This applies to `describe` too: if one or more `describe` blocks are marked `describe.only`, then only those blocks get run. If any of *those* blocks contain `it.only`, the same logic is applied to the test set.

This is pretty complex to think about: there's some kind of recursive selection of focus going on.

In this chapter, we'll redesign our test runner to handle this use case. By the end of the chapter, you'll have learned how to

- Judge when a software design is no longer fit for purpose and needs rework.

- Use a stack data structure to split parsing a test suite from its execution.

- Filter out tests using the `it.only` function call.

We start with a look at one of the primary scenarios in which focusing tests is useful.

The Refactor Workflow

In the last chapter, we discussed how the test runner takes a key role in your personal development workflows. For example, if you're rerunning tests each time you make a change, you might wish to avoid running your whole test suite and concentrate on single files instead.

The `only` modifier for test cases and test suites is another way to do this. It can be used when you have a set of tests failing that you want to keep track of for your next changes. This is especially useful when you're refactoring code.

The process would work a little like this:

1.   Make a set of changes to the application code.

2.   Run all tests with `npm test` and observe that multiple tests are failing across multiple files.

3.   Mark each of these tests with the `only` modifier.

4.   Then rerun `npm test` and get to work on fixing those tests.

By doing this you're reducing the time spent running your test suite and you're reducing noise.

Introducing a Parse Step

Let's think about how we can support the new `only` modifier in our test runner code.

In order to avoid running some tests, we need to parse *all* of the tests so that we know if we're focusing on any tests or not. But at the moment, we run tests as soon as they are parsed. We need to split this process out into parsing and running.

What we'll do in this chapter is build a three-stage process, shown in Figure 8-1, consisting of *parse*, *filter*, and *run* phases.

Figure 8-1. *The three phases of a test run*

The *parse* phase will store tests. We'll use the same structures we have already but add a new `run` function that is used to defer running of tests. The result of this phase is a tree of objects: each object has a `children` array that is a set of tests and describe blocks.

The *filter* phase cuts this tree down to only the focused tests, if any at all are focused.

Finally, the *run* phase simply calls *run* on everything left after filtering.

Implementing the Parse Phase

We'll use this as an opportunity to rewrite our test functions in a new module, called `src/testContext.mjs`. We will rewrite `it` and `describe` within this file and rewrite the shared context between the two. We'll then move `beforeEach` and `afterEach` over as well.

In this way we'll begin to clean up `src/runner.mjs`, which at 160+ lines is starting to get quite big. (Anything over 100 lines in a single file is too much, in my humble opinion.)

Some frameworks, like RSpec, feature another mechanism for focusing tests, which is to specify the line number of a test to run on the command line. If we wanted to implement something like this, we'd still need to split out the parsing and running of specs, *and* we'd also need a way of discovering call site information as each test is run, so I believe this solution would be more complex for us than adding `.only` functions.

Jasmine uses a `fit` function to focus on tests. I prefer to use the `.only` modifier to tests because it means one fewer import: `it.only` is accessible as soon as you've imported `it`, because it's implemented as a property on `it`, not as a separate function.

One final thing to mention: Splitting out parsing and running of tests will allow us to support randomized tests, which we'll get to in Chapter 14.

Parsing the describe Function

In your test runner project, create a new file called `src/testContext.mjs`, and add the following. It's not all that different from what we had before; we'll cover the differences afterward.

Listing 8-1. concise-test/src/testContext.mjs

```
let currentDescribe;

const makeDescribe = (name) => ({
  name,
  befores: [],
  afters: [],
  children: [],
});
```

```
currentDescribe = makeDescribe("root");

const describeWithOpts = (name, body, options) => {
  const parentDescribe = currentDescribe;
  currentDescribe = makeDescribe(name, options);
  body();
  currentDescribe = {
    ...parentDescribe,
    children: [
      ...parentDescribe.children,
      currentDescribe,
    ],
  };
};

export const describe = (name, body) =>
  describeWithOpts(name, body, {});
```

What's changed from before? Here's the list:

- We got rid of the describeStack, because it's used only when running tests. We'll come back to this in just a second.

- Each describe block now has a children property and extensible options.

- describeWithOpts is a new "inner" describe that isn't exported, but takes a third argument, options.

- describeWithOpts does not print any console output.

- describe simply calls through to describeWithOpts but with an empty options.

- We introduced a root describe block.

A little later we'll add `describe.only`, which will call
`describeWithOpts` but with `options` set to `{ focus: true }`.

Why the removal of `describeStack` and the addition of a `root` block?

`describeStack` was used purely during running tests: for knowing
how much to indent test names in a summary list and also for knowing the
execution order of `befores` and `afterEach` blocks. We will need this same
behavior later on, but not during the parse phase.

`root` is needed because now we're collecting tests *and* describe blocks.
Since `describe` can now become `describe.only`, we need some way of
collecting the top-level describe blocks. We do that by having a "hidden"
root-level describe block.

We are going to use `options` to pass through an option of `focus`. If this
is set to `true`, then we know the block should be focused.

We'll pass this through implicitly when we define `describe.only`,
which is coming up soon.

Adding the it Function

Let's continue on with the addition of `it`. Remember, this won't run any
test but instead simply store off the body of the `it` within the `children` of
the describe block.

Listing 8-2. concise-test/src/testContext.mjs

```
const makeTest = (name, body, options) => ({
  name,
  body,
  ...options,
  errors: [],
});

const itWithOpts = (name, body, options) => {
  currentDescribe = {
```

```
      ...currentDescribe,
      children: [
        ...currentDescribe.children,
        makeTest(name, body, options),
      ],
    };
  };

export const it = (name, body) =>
  itWithOpts(name, body, {});
```

All our new `it` does is push a new test object to the `currentDescribe` array.

Adding the beforeEach and afterEach Functions

These actually simplify since they use `currentDescribe` directly.

Listing 8-3. concise-test/src/testContext.mjs

```
export const beforeEach = (body) => {
  currentDescribe = {
    ...currentDescribe,
    befores: [...currentDescribe.befores, body],
  };
};

export const afterEach = (body) => {
  currentDescribe = {
    ...currentDescribe,
    afters: [...currentDescribe.afters, body],
  };
};
```

Implementing the Run Phase

Time to think about running our code. `describe` and `it` both require different options, so first we need to be able to tell them apart. We can do that by the presence of body—only `it` has one of these:

```
const isIt = (testObject) =>
  testObject.hasOwnProperty("body");
```

Now let's define `runDescribe`. This sets up the current `describeStack` and then calls `runBlock` on each of its `children`.

Listing 8-4. concise-test/src/testContext.mjs

```
let describeStack = [];

const indent = (message) =>
  `${" ".repeat(describeStack.length * 2)}${message}`;

const withoutLast = (arr) => arr.slice(0, -1);

const runDescribe = (describe) => {
  console.log(indent(describe.name));
  describeStack = [...describeStack, describe];
  describe.children.forEach(runBlock);
  describeStack = withoutLast(describeStack);
};
```

Then, `runIt`. First, import `color`:

```
import { color } from "./colors.mjs";
```

Now define `runIt` at the bottom of the file.

Listing 8-5. concise-test/src/testContext.mjs

```
let successes = 0;
let failures = [];

const runIt = (test) => {
  global.currentTest = test;
  currentTest.describeStack = [...describeStack];
  try {
    invokeBefores(currentTest);
    currentTest.body();
    invokeAfters(currentTest);
  } catch (e) {
    currentTest.errors.push(e);
  }
  if (currentTest.errors.length > 0) {
    console.log(
      indent(
        color(`<red>${cross}</red> ${currentTest.name}`)
      )
    );
    failures.push(currentTest);
  } else {
    successes++;
    console.log(
      indent(
        color(`<green>${tick}</green> ${currentTest.name}`)
      )
    );
  }
  global.currentTest = null;
};
```

One important aspect of this is the setting of a global variable, currentTest. This allows currentTest to be accessed by all the modules within our system. Most importantly, expect needs to access this in order to push failures to it. The expect function won't live within this file.

The Global currentTest Variable

It's important that once runIt has set global.currentTest, all future usages of the test object go via this global, rather than the parameter test. In Chapter 12, we will write a function named it.timesOutAfter that depends on this behavior.

Another thing to be aware of is this line:

```
currentTest.describeStack = [...describeStack];
```

This line ensures that all failures have their describe context captured so that it can be printed out on-screen.

For the next change, move across invokeAll, invokeBefores, and invokeAfters from src/runner.mjs. These functions don't need to change.

Listing 8-6. concise-test/src/testContext.mjs

```
const invokeAll = (fnArray) =>
  fnArray.forEach((fn) => fn());

const invokeBefores = () =>
  invokeAll(
    describeStack.flatMap((describe) => describe.befores)
  );

const invokeAfters = () =>
  invokeAll(
    describeStack.flatMap((describe) => describe.afters)
  );
```

Now all that's left is a way to select between the two; that's the `runBlock` function:

```
const runBlock = (block) =>
  isIt(block) ? runIt(block) : runDescribe(block);
```

Finally, we can write a wrapper around `run` to allow it to all of the `children` of the root describe block and output the results:

```
export const runParsedBlocks = () => {
  currentDescribe.children.forEach(runBlock);
  return { successes, failures };
};
```

Updating the Runner

Now it's time to move back to `src/runner.mjs`. Start by adding the following import:

```
import { runParsedBlocks } from "./testContext.mjs";
```

Update the `run` function to call the imported function.

Listing 8-7. concise-test/src/runner.mjs

```
export const run = async () => {
  try {
    const testFilePaths = await chooseTestFiles();
    await Promise.all(
      testFilePaths.map(async (testFilePath) => {
        await import(testFilePath);
      })
    );
```

```
    const { failures, successes } = runParsedBlocks();
    printFailures(failures);
    console.log(
      color(
        `<green>${successes}</green> tests passed, ` +
          `<red>${failures.length}</red> tests failed.`
      )
    );
    process.exit(
      failures.length > 0
        ? exitCodes.failures
        : exitCodes.ok
    );
  } catch (e) {
    console.error(e.message);
    console.error(e.stack);
    process.exit(exitCodes.parseError);
  }
};
```

Update `printFailures` to take a `failures` parameter.

Listing 8-8. concise-test/src/runner.mjs

```
const printFailures = (failures) => {
  if (failures.length > 0) {
    console.error("");
    console.error("Failures:");
    console.error("");
  }
  failures.forEach(printFailure);
};
```

Then delete all of these functions: `indent`, `describe`, `it`, `beforeEach`, `afterEach`, `makeTest`, `makeDescribe`, `withoutLast`, `last`, `currentDescribe`, and `updateDescribe`.

Delete the module-level variables: `successes`, `failures`, `currentTest,` and `describeStack`.

The preceding `run` function uses a new exit code: `parseError`. Define that now:

```
const exitCodes = {
  ok: 0,
  failures: 1,
  cannotAccessFile: 2,
  parseError: 3,
};
```

Next, we need to update `package.json` to use `src/testContext.mjs` as the file with default exports:

```
"exports": {
  ".": "./src/testContext.mjs"
}
```

Moving the expect Function

The switch to the default exports will have "broken" `expect`. Let's move that function into its own file and then import and re-export it from within `src/testContext.mjs`.

Create a new file named `src/expect.mjs` and move both `matcherHandler` and `expect` into it, plus the imports for `ExpectationError` and `matchers`.

Then back in `src/testContext.mjs`, re-export this:

```
export { expect } from "./expect.mjs";
```

If you run your tests now for your sample application, you should (with any luck) see the same result as before.

Now it's time to support focus.

Adding the Focus Functions

Here's the fun part. In `src/testContext.mjs`, add a function `addModifier` that will allow us to define secondary functions as properties that hang off the main function. These secondary functions will call the original but with an additional `options` parameter. We'll call this type of function a *modifier function*, and we'll add more later in the book: `timesOutAfter` in Chapter 9, `behavesLike` in Chapter 11, and `skip` in Chapter 13.

Listing 8-9. concise-test/src/testContext.mjs

```
const addModifier = (object, property, fn, options) =>
  Object.defineProperty(object, property, {
    value: (...args) => fn(...args, options),
  });

addModifier(it, "only", itWithOpts, { focus: true });

addModifier(describe, "only", describeWithOpts, {
  focus: true,
});
```

You can see from this how a test object ends up with an option of `focus` set to true. We'll use this to filter out tests.

Filtering Tests

Create a new file called `src/focus.mjs` and add the following function. Its job is bubble up any `{ filter: true }` to the root describe block, cutting out non-focused tests along the way. It does this by recursively calling `focusedOnly` on child nodes in a depth-first fashion. The leaf nodes–the tests–are passed back as is. But further up the tree, a `describe` block can have its `focus` changed to `true` *and in addition* have its children cut down to focused children as long as one of them is focused.

By the time this function call returns, it will either be returning either *exactly the same* tree if nothing was focused or only a *subset of tests and describe blocks* that were explicitly set as focused by the user.

Listing 8-10. concise-test/src/focus.mjs

```
export const focusedOnly = (block) => {
  if (!block.children) {
    return block;
  }
  const focusedChildren = block.children.map(focusedOnly);
  if (focusedChildren.some((child) => child.focus)) {
    return {
      ...block,
      focus: true,
      children: focusedChildren.filter(
        (child) => child.focus
      ),
    };
  } else {
    return block;
  }
};
```

I'd like to draw attention to one very subtle thing happening here. If a `describe` block is focused but some of its tests are *also* focused, then the `describe` block's tests will still be filtered.

Time to enable it. Back in `src/testContext.mjs`, import the new function:

```
import { focusedOnly } from "./focus.mjs";
```

Then update `runParsedBlocks` to filter these tests out:

```
export const runParsedBlocks = () => {
  const withFocus = focusedOnly(currentDescribe);
  withFocus.children.forEach(runBlock);
  return { successes, failures };
};
```

That's everything done. Time to test. Here's some manual test cases you can try out:

- Focus a single `it` by changing `it` to `it.only`. Only that one test should be run.

- Focus two `it` calls in a single `describe` block. Both tests should be run.

- Focus a single `describe` block to true, but no tests within it. All tests in that describe block should be run.

- Focus a single `describe` block and then focus a single test within it. Only that one test should be run.

- Focus a nested `describe` block but not the parent. All the tests within the nested describe block should be run.

- Focus a nested `describe` block *and* its parent. All the tests within the nested describe block should be run.

- Focus a `describe` block with nested `describe` blocks. All the tests within those nested describe blocks should be run.

Exercises

1. Add the ability to run a single test by specifying a line number on the command line, along with the file name, for example:

```
npm test test/tests.mjs:43
```

This should look for the `it` function that is defined on line 43 or, if there isn't one on that line, then the first `it` function that appears walking backward from line 43.

2. Add a new mode, `retest`, that reruns only the *last set* of test failures. This means that a test run will need to save these test failures to a file (of your choosing). This new mode should then read that file, load only the required test files, and run only the specified tests from within those suites.

Discussion Question

1. Given that test failures very often result in adjusting test code, how useful do you think the `retest` mode that you implemented in Exercise 3 will be in practice? Is there much benefit to having this mode when you can simply use your terminal's history to rerun the last terminal command?

Summary

In this chapter we've written all of our test functions so that there are two phases of a test run: a parse phase and a run phase. Once we'd refactored our code to use that approach, the change to filter out focused tests boiled down to a 15-line function.

We've now covered what is–in my opinion–the most difficult piece of the framework. But we're still a few features short of a usable test framework. In the next chapter, we'll add a key requirement for testing: the ability to run asynchronous tests.

CHAPTER 9

Supporting Asynchronous Tests

Sometimes we want to test async stuff:

```
it("does something async", async () => {
  await something();
  expect(thing).toBe("foo");
});
```

For this to work, notice that the second argument to the `it` function call is now marked as `async`. This is a hint to how the test runner must change: it must assume that the test body returns a `Promise` object, which is what the `async` method means.

But with our current test runner, our tests will complete, reporting no failures for any test that awaits a `Promise`, because our test runner won't bother to wait for it to complete.

In this chapter we'll add support for tests like these. You will

- Learn why asynchronous behavior is so important to the JavaScript runtime environment.

- Update the test runner to wait for asynchronous operations.

- Deal with tests that do not complete in a reasonable amount of time.

Before rushing into an implementation, it's worth refreshing our memories of how the JavaScript runtime environment handles asynchronous behavior.

Event Loops in Runtime Environments

Our test runner is a command-line utility that exits as soon as it is done processing your test run. It doesn't wait for any user input or any system events. But many programs *do* wait for external input, and these programs in general use an architectural pattern known as a message or *event loop*.

While the program runs, messages are placed on an internal message queue and are processed one by one. These messages might be user events like button presses or keyboard input, or they may be network events, like how an HTTP server processes incoming HTTP requests.

These messages can also be initiated by your own application code. You do this when some of your code will need to wait for a period of time–for example, if it's made an HTTP request and is waiting for a response. In the browser runtime environment (and also desktop environments and mobile platforms), the message loop runs on the main thread, which is also processing user input events. So to avoid *blocking* the user interface, you can dispatch this long-running event back to the message queue. The system is then free to process the next message while it waits for your event to finish.

Figure 9-1 shows how these concepts link together in *the* browser runtime environment. *Messages are continually taken off the message queue, and the associated code block is executed. Those code blocks can themselves queue more messages. Messages can also arrive* in *the* form of *DOM* events that are created by the browser.
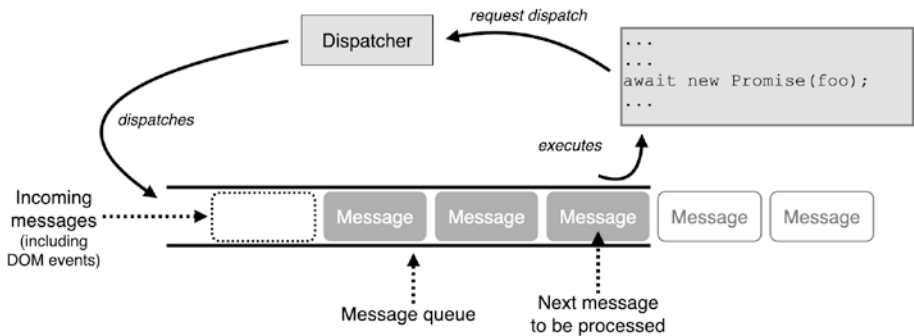
Figure 9-1. *A view of the event loop*

Part of the beauty of this system is that you are relieved from thinking about any of the multi-threading concerns. Take, for example, this code:

```
const response = await fetch("www.example.com");
```

The internals of the browser's Fetch API[1] take care of opening and waiting on the network connection, and the await keyword packages the rest of your code and ensures it runs once the fetch call message has been processed by the message queue.

Synchronicity and the Test Runner

But… isn't the test runner *synchronous*? As noted in the last section, the only input is the command-line arguments, and the process terminates once the test run is complete. There are no other external messages to process.

So why do we need asynchronous code support in the test environment?

It turns out that some other programming environments allow asynchronous requests to be converted to synchronous code, and this

---

[1] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

can be beneficial because it can simplify how you write your tests. Your test code doesn't need to wait for anything, because it knows that once a statement has finished, the execution has completed. Simple.

It can also speed up your tests. Imagine, for example, you had this in your application code:

```
await new Promise((resolve) => setTimeout(resolve, 4000));
```

In your running application, this code might be perfectly valid. But in your test runner, this still has the effect of pausing all execution for 4 seconds. Four whole seconds! Nobody needs their test runner to stop like that.

In fact, during an automated test run, you want to *flatten time* as much as possible. The ideal time for a test run is 0 seconds. Now obviously it's physically possible for your test suites to execute in 0 seconds, but we can still aim for it by cutting out all time offenders, like the preceding line.

On the front end, this kind of time offense happens a *lot* because we tend to use component packages that have all sorts of lovely animations. (Wouldn't it be great if we could tell the test runner to *never wait* on those?)

In the JavaScript browser environment, controlling time and removing asynchronicity is actually quite difficult and is beyond the scope of this book.

So we'll settle with ensuring that we honor the message loop and can wait for asynchronous tasks to complete.

Waiting for Promise Completion

With a test body marked `async`, as in the preceding example, the returned value is a `Promise`. All we need to do in that case is await the result.

In `src/testContext.mjs`, define a new function, executedAndWaitForBody, as shown in the following. We'll call this from the existing `runIt` function.

Listing 9-1. concise-test/src/testContext.mjs

```
const runBodyAndWait = async ({ body }) => {
  const result = body();
  if (result instanceof Promise) {
    await result;
  }
};
```

The `if` is necessary because JavaScript won't allow us to `await` a non-Promise value.

Now update `runIt` to use that function. Make sure to mark the function definition as `async` too.

Listing 9-2. concise-test/src/testContext.mjs

```
const runIt = async test => {
  // .. previous implementation
  try {
    invokeBefores(currentTest);
    await runBodyAndWait(currentTest.body);
    invokeAfters(currentTest);
  }
  // .. previous implementation
};
```

We need to pull that async behavior all the way up the run chain. That means `runDescribe` too. This one is tricky because we want to run tests sequentially, meaning we need to await within a loop. We can do that with a standard `for` loop rather than a `forEach`.

Listing 9-3. concise-test/src/testContext.mjs

```
const runDescribe = async (describe) => {
  console.log(indent(describe.name));
  describeStack = [...describeStack, describe];
  for (let i = 0; i < describe.children.length; ++i) {
    await runBlock(describe.children[i]);
  }
  describeStack = withoutLast(describeStack);
};
```

Then on to `runParsedBlocks`.

Listing 9-4. concise-test/src/testContext.mjs

```
export const runParsedBlocks = async () => {
  const withFocus = focusedOnly(currentDescribe);
  for (let i = 0; i < withFocus.children.length; ++i) {
    await runBlock(withFocus.children[i]);
  }
  return { successes, failures };
};
```

In `src/runner.mjs`, it's a simple matter of finding the invocation of `runParsedBlocks` and putting `await` in front of it. The containing function, `run`, is already `async`:

```
const { failures, successes } = await runParsedBlocks();
```

Testing It Out

The sample application tests don't have any async behavior for testing, but you can modify the tests in our sample application. Add a new describe block at the bottom of `test/todo.tests.mjs`.

Listing 9-5. todo-example/test/todo.tests.mjs

```
describe("async tests", () => {
  it("waits for a little bit", async () => {
    await new Promise((resolve, _) =>
      setTimeout(resolve, 5000)
    );
  });
});
```

The middle line causes the test to pause for 5 seconds. If you run `npm test` now, you should see the test runner waiting, passing the test, and then continuing on.

Catching Exceptions

You can also test it with a promise rejection:

```
it('waits for a little bit', async () => {
  await new Promise((_, reject =>
    setTimeout(() => reject(new Error('error!')), 5000));
});
```

This should work straight away without any further change.

If you don't pass any parameters to `reject`, our test runner will blow up. That's because the runtime `throws` whatever you pass to `reject`. Our test try-catch always expects an object of type `Error`, but in this case it would receive `undefined`. Try it out!

What would you do to fix this?

Timing Out Tests After a Period of Time with the it.timesOutAfter Modifier

What should happen if an async test runs for too long? We can set up our tests to fail after a certain period.

We'll create a function called `it.timesOutAfter` that takes a number of milliseconds as an argument. If the test doesn't complete, within that time, it fails:

```
it.timesOutAfter(10000);
```

We'll set this up on `it` itself, just like with `it.only`. This time, however, the function should be run during the run phase, unlike `it.only` that is run during the parse phase.

First, let's define a new exception type called `TestTimeoutError`. Create a new file, `src/TestTimeoutError.mjs`, and add the following.

Listing 9-6. concise-test/src/TestTimeoutError.mjs

```
export class TestTimeoutError extends Error {
  constructor(timeout) {
    super(`Test timed out after ${timeout}ms`);
    this.timeout = timeout;
  }
}
```

```
createTimeoutPromise() {
  return new Promise((_, reject) =>
    setTimeout(() => reject(this), this.timeout)
  );
}
}
```

Back in `src/testContext.mjs`, import that class:

```
import { TestTimeoutError } from "./TestTimeoutError.mjs";
```

We'll start by defining a *default timeout*. In fact we're going to save off an `Error` object. We do that because we can then capture the stack trace at the moment `it.timesOutAfter` was called, as we'll soon see.

Update `makeTest` to include this default error.

Listing 9-7. concise-test/src/testContext.mjs

```
const makeTest = (name, body, options) => ({
  name,
  body,
  ...options,
  errors: [],
  timeoutError: new TestTimeoutError(5000),
});
```

Now add the following new function. I've placed this just below the definition for `runIt`.

Listing 9-8. concise-test/src/testContext.mjs

```
const appendTimeout = (timeout) => {
  currentTest = {
    ...currentTest,
```

```
    timeoutError: new TestTimeoutError(timeout),
  };
};

addModifier(it, "timesOutAfter", appendTimeout, {});
```

How do we make use of this? We use `Promise.race`: pitting our existing `Promise` against one that times out at the current timeout.

Update `runBodyAndWait` to do that. It'll need a new helper, `timeoutPromise`.

Listing 9-9. concise-test/src/testContext.mjs

```
const timeoutPromise = () =>
  currentTest.timeoutError.createTimeoutPromise();

const runBodyAndWait = async (body) => {
  const result = body();
  if (result instanceof Promise) {
    await Promise.race([result, timeoutPromise()]);
  }
};
```

The `timeoutPromise` must make use of `currentTest` rather than the version of `test` that `runIt` stores. That's because the body can itself be the code that sets the timeout on `currentTest`. So we need to ensure we take that latest version.

This feels like something of a design smell to me. It's not clear from the code that this is happening.

That's everything complete!

Testing It Out

Update your sample application test to call `timesOutAfter` with a setting of 2000.

Listing 9-10. todo-example/test/todo.tests.mjs

```
it("waits for a little bit", async () => {
  it.timesOutAfter(2000);
  await new Promise((resolve, _) =>
    setTimeout(resolve, 5000)
  );
});
```

You should get this output:

```
async tests → waits for a little bit
Test timed out after 2000ms

  in test/todo.tests.mjs:

  12 | describe('async tests', () => {
  13 |   it.only('waits for a little bit', async () => {
  14 |     it.timesOutAfter(2000);
     |         ^
  15 |     await new Promise((resolve, _) =>
           setTimeout(resolve, 5000));
  16 |   });
```

Lovely! Try it out in a `beforeEach` block too.

Exercise

1. Propose a solution for running promises synchronously, including package code, like React. The benefit of this is that you are not at the mercy of any timing issues, for example, due to animations running when controls are manipulated (like a dropdown that takes half a second to unfold its options). This might involve writing a Babel plugin to convert async/await and promise code to your own implementation. What problems do you foresee? Proceed with the implementation until you get blocked.

Discussion Questions

1. Imagine you had a construct that allowed *synchronous* execution of your test code. All promises are run inline, including DOM events. Think about some of the UI tests you've written in JavaScript. How would having synchronous tests affect the running of your tests?

2. Adding the modifier `it.timesOutAfter` makes the definition of `addModifier` a bit of a mess: the `name` property is used to pass along the timeout. Partly this is because `it.timesOutAfter` is not used to define a test, but instead is used to modify test options *during* a test run. Design a better solution to this problem that does *not* use the `addModifier` function but instead uses a new function,

`addDuringExecutionModifier`, that achieves the
same thing but only while a value has been set for
`global.currentTest`.

Summary

We've now covered all the basic functionality required to support
asynchronous tests.

There are still some missing pieces. We already discovered a couple of
places where the design is lacking, like what happens if a promise rejects
with no error.

But we'll move on, because there are bigger (and arguably more
important) issues to tackle.

In the next chapter, we'll shift focus again and look in detail at the test
runner's mechanism for building test reports.

Reporting

Currently, the test runner's reporting code is pretty fixed. There's no way to configure it or switch it out. This might be a problem, for example, with sending output to a text file on a Continuous Integration (CI) environment, since text files don't support ANSI escape codes.

In this chapter we'll implement a system for supporting different output mechanisms for test reports. We'll do this by writing a basic event dispatcher that will allow us to dispatch events to a reporter.

You'll learn

- How the observer pattern can be applied to a JavaScript codebase

- How to refactor code to split out test runner processing events from console I/O

- How to implement more processing events

- How to use the process exit code to signal success or failure to other parts of the system

We'll begin with a look at the observer design pattern.

The Observer Pattern, in a Half-Baked Style

This is a pattern as old as the hills and is simply a way of saying, "When stuff happens to me, I want to let other objects know about it, and they can elect whether or not they are interested."

When you work in a functional environment such as JavaScript, an *observer* can simply be a function that is stored by the *subject* and invoked whenever an event happens. Figure 10-1 shows how this might look.
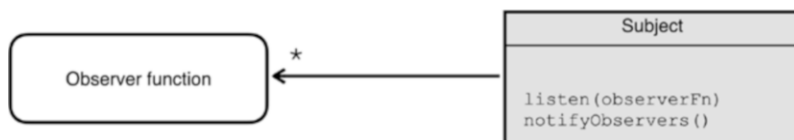
Figure 10-1. *The observer pattern when applied to a subject, with each observer existing as a single function*

It is the subject's job to maintain the list of listening observers.[1]

In our test runner, we'll complicate this a little by having multiple *events* such as beginningDescribe and finishedTest that the observer can listen to individually.

That means our subject (the test runner) will need to maintain multiple lists of observers, one for each event.

Because of that, we'll invent a type of "event dispatcher," triggered by a call to a dispatch function, that takes an event name and a set of arguments, then selects the right set of observers to notify, and then notifies them of the event.

Adding an Event Dispatcher

In your test runner project, create a new file src/eventDispatcher.mjs, and add the following content.

Listing 10-1. concise-test/src/eventDispatcher.mjs

```
import process from "process";

const observers = {};
```

---

[1] In some systems you may need to have an *unlisten* or *unsubscribe* method too.

```
export const listen = (eventName, handler) => {
  if (!observers[eventName]) {
    observers[eventName] = [handler];
  } else {
    observers[eventName] = [
      ...observers[eventName],
      handler,
    ];
  }
};

const dispatchNow = (eventName, ...args) =>
  observers[eventName].forEach((observer) => {
    try {
      observer(...args);
    } catch {}
  });

export const dispatch = (eventName, ...args) => {
  if (observers[eventName])
    dispatchNow(eventName, ...args);
};
```

This new namespace exports two functions: `listen` and `dispatch`. Any part of the system, even reporters themselves, can dispatch new events. The events are executed inline.

Any exception that occurs in the observer is simply swallowed. This is probably a good thing; observer success is of a lower "priority" than our own runner code, and it shouldn't impact the execution of our test run.

To see how this API is used, let's create a new reporter and listen for our first event.

Dispatching a Begin Suite Event

We'll start with an event named `beginningDescribe`, which correlates to the `runDescribe` event being called in `src/testContext.mjs`.

Create a new file in a subdirectory[2] of `src`: `src/reporters/default.mjs`. Add the following content.

Listing 10-2. concise-test/src/reporters/default.mjs

```
import { listen } from "../eventDispatcher.mjs";

const indent = (stack, message) =>
  `${" ".repeat(stack.length * 2)}${message}`;

export const installReporter = () => {
  listen(
    "beginningDescribe",
    (describeStack, { name }) => {
      console.log(indent(describeStack, name));
    }
  );
};
```

The `installReporter` function is the one we'll use to plug the reporter into place.

You'll notice this is almost a like-for-like copy of the current output for `runDescribe`. The difference is we can no longer use the `describeStack` module-level variable, so we have to pass that through.

---

[2] This is in anticipation of you adding more reporters of your own!

164

Now in `src/testContext.mjs`, add the following import:

```
import { dispatch } from "./eventDispatcher.mjs";
```

Update `runDescribe` to replace the `console.log` with a call to `dispatch`.

Listing 10-3. concise-test/src/testContext.mjs

```
const runDescribe = async (describe) => {
  dispatch("beginningDescribe", describeStack, describe);
  describeStack = [...describeStack, describe];
  for (let i = 0; i < describe.children.length; ++i) {
     await runBlock(describe.children[i]);
  }
  describeStack = withoutLast(describeStack);
};
```

Open `src/runner.mjs` and install the reporter. Start with the import:

```
import { installReporter } from "./reporters/default.mjs";
```

At the top of `run`, call `installReporter`.

Listing 10-4. concise-test/src/runner.mjs

```
export const run = async () => {
  installReporter();
  // ... existing code here ...
};
```

Time to test it out. Go back to your sample application and run `npm test`. You should see exactly the same output as before. This is a good sign! Now all that remains is replacing the rest of the events.

Dispatching a Test Finished Event

Let's add another event: `finishedTest`. This event will print out the summary of the test failure, and it will *also* save off its own copies of `failures` and `successes`. We'll use these with the next event, `finishedTestRun`.

In `src/reporters/default.mjs`, start by adding a new import for `color`:

```
import { color } from "../colors.mjs";
```

At the top of the definition of `installReporter`, add in two new values of `successes` and `failures`:

```
export const installReporter = () => {
  let successes = 0;
  let failures = [];
  // ... existing code here ...
};
```

Then add the following new `listen` call at the bottom of `installReporter`.

Listing 10-5. concise-test/src/reporters/default.mjs

```
const tick = "\u2713";
const cross = "\u2717";
listen("finishedTest", (test) => {
  if (test.errors.length > 0) {
    console.log(
      indent(
        test.describeStack,
        color(`<red>${cross}</red> ${test.name}`)
      )
    );
    failures.push(test);
```

```
    } else {
      successes++;
      console.log(
        indent(
          test.describeStack,
          color(`<green>${tick}</green> ${test.name}`)
        )
      );
    }
});
```

We're ready to replace the existing code. Open `src/testContext.mjs` and replace `runIt` as shown.

Listing 10-6. concise-test/src/testContext.mjs

```
const runIt = async (test) => {
  global.currentTest = test;
  currentTest.describeStack = [...describeStack];
  try {
    invokeBefores(currentTest);
    await runBodyAndWait(currentTest.body);
    invokeAfters(currentTest);
  } catch (e) {
    currentTest.errors.push(e);
  }
  dispatch("finishedTest", currentTest);
  if (currentTest.errors.length > 0) {
    failures.push(currentTest);
  } else {
    successes++;
  }
  global.currentTest = null;
};
```

You can now remove the `indent` function.

In the preceding snippet, I haven't yet removed the use of `failures` and `successes`; we'll do that in the next section.

For now, go ahead and rerun the sample application tests. With any luck they still contain the same output!

Adding finishedTestRun

Time to finish off the output of `successes` and `failures`.

In `src/reporters/default.mjs`, add the next invocation of `listen` to the `installReporter` function.

Listing 10-7. concise-test/src/reporters/default.mjs

```
listen("finishedTestRun", () => {
  printFailures(failures);
  console.log(
    color(
      `<green>${successes}</green> tests passed, ` +
        `<red>${failures.length}</red> tests failed.`
    )
  );
});
```

You will now need to transfer across `printFailures`, `printFailure`, and `fullTestDescription` from `src/runner.mjs`. They can be placed above `installReporter`.

In `src/runner.mjs`, add the following import. You can also delete the import for `color` since we'll be removing its use in the next step:

```
import { dispatch } from "./eventDispatcher.mjs";
```

Update `run` to dispatch this new event. Remove the call to `printFailures` and `console.log` too.

Listing 10-8. concise-test/src/runner.mjs

```
export const run = async () => {
  installReporter();
  try {
    const testFilePaths = await chooseTestFiles();
    await Promise.all(
      testFilePaths.map(async (testFilePath) => {
        await import(testFilePath);
      })
    );
    const failed = await runParsedBlocks();
    dispatch("finishedTestRun");
    process.exit(
      failed ? exitCodes.failures : exitCodes.ok
    );
  } catch (e) {
    console.error(e);
    process.exit(exitCodes.parseError);
  }
};
```

Be careful with that call to `runParsedBlocks`: we can get rid of the use of `successes`, but not `failures`, since we use that for the process exit code.

Go ahead and run your tests; they should still be showing the same output.

Getting Rid of Failures and Successes

We've done the majority of work now, but we have one problem still to solve: that of the process exit code. It'd be great to get rid of the `failures` count already, but how else would we know the right error code to return?

We *could* ask the reporter to do it. But is that the concern of the reporter? I don't think so. Our `runner` is the thing that knows about the CLI and the `process`, so it should be responsible for exiting.

Instead, what we'll do is parse our final test context tree for any non-zero `error` arrays.

Open `src/testContext.mjs` and remove the two declarations of `successes` and `failures`.

Then update `runIt` to remove the updates to those removed variables.

Listing 10-9. concise-test/src/testContext.mjs

```
const runIt = async (test) => {
  global.currentTest = test;
  currentTest.describeStack = [...describeStack];
  try {
    invokeBefores(currentTest);
    await runBodyAndWait(currentTest.body);
    invokeAfters(currentTest);
  } catch (e) {
    currentTest.errors.push(e);
  }
  dispatch("finishedTest", currentTest);
  global.currentTest = null;
};
```

Now add a new function, named `anyFailure`, just above `runParsedBlocks` at the bottom.

Listing 10-10. concise-test/src/testContext.mjs

```
const anyFailed = (block) => {
  if (isIt(block)) {
    return block.errors.length > 0;
```

```
  } else {
    return block.children.some(anyFailed);
  }
};
```

Then finally update `runParsedBlocks` to use this new function, rather than `successes` and `failures`.

Listing 10-11. concise-test/src/testContext.mjs

```
export const runParsedBlocks = async () => {
  const withFocus = focusedOnly(currentDescribe);
  for (let i = 0; i < withFocus.children.length; ++i) {
    await runBlock(withFocus.children[i]);
  }
  return anyFailed(withFocus);
};
```

Back in `src/runner.mjs`, update run as follows.

Listing 10-12. concise-test/src/runner.mjs

```
export const run = async () => {
  installReporter();
  try {
    const testFilePaths = await chooseTestFiles();
    await Promise.all(
      testFilePaths.map(async (testFilePath) => {
        await import(testFilePath);
      })
    );
    const failed = await runParsedBlocks();
    dispatch("finishedTestRun");
```

```
   process.exit(
     failed ? exitCodes.failures : exitCodes.ok
   );
 } catch (e) {
   console.error(e.message);
   console.error(e.stack);
   process.exit(exitCodes.parseError);
 }
};
```

That's it! Go ahead and run your sample application and see if the exit code is still working. You'll be able to tell by the presence of the `npm  ERR` line at the bottom. Make sure you test both success and failure cases.

Exercises

1. Add a CLI argument that allows a user-defined listener to be used at runtime. Ensure that your subject protects against exceptions that may occur.

2. Write a new "concise" listener that prints a series of green dots for each successful test case and a red "F" for any failing test. Once the test run is complete, it should list out all the tests that failed together with their failure information.

3. Rewrite the dispatcher so that it maintains only a single array of observers. The implementation of the observer will need to change to support this. One way to do this would be to convert the observer from a function to an object with the event names being properties on the object.

4. The *Test Anything Protocol (TAP)* is a protocol for reporting test runs. It dates from 1987 when it was used in the automated test suite for the first version of Perl. Write a reporter that adheres to the latest TAP specification. It can be found at the `https://testanything.org` website.

Summary

Adding a `dispatch` function and using it to separate out reporting from our test runner has significantly improved the readability of our code.

There's an important distinction here between code that is core to our execution flow and code that isn't: observers that listen for events cannot have an impact on the flow of test execution, as is evident of our inclusion of an empty catch block that suppresses errors.

Plenty of improvements await you: be it more runners, more events, or a more battle-hardened implementation.

This chapter wraps up what might be called "core" functionality of a test runner.

In this next chapter, we'll add a feature that isn't common in JavaScript test runners but still very useful: *shared examples*.

PART III

Extending for Power Users

In this part, you will add some unique features that are borrowed from other programming environments.

In Chapter 11, "Shared Examples," we borrow an important feature from Ruby: inheritance for `describe` blocks, which gives us a flexible mechanism for removing duplication between test groups.

In Chapter 12, "Tagging Tests," we create a mechanism for running a group of tests based on their tag.

In Chapter 13, "Skipping Tests," we introduce a number of ways to skip tests, including tests without bodies, and `it.skip` and `describe.skip` functions.

In Chapter 14, "Randomizing Tests," we add a runtime flag for randomizing the order of tests, which is a useful technique for catching invisible dependencies between tests.

CHAPTER 11

Shared Examples

With our test runner in an extremely functional state, it's time to look at some more interesting features.

In this chapter we'll implement a pair of functions: the `describe.shared` function and the `it.behavesLike` function. They can be used to apply a test suite to multiple different objects, each of which is expected to implement the same behavior.

By the end of the chapter, you'll have discovered

- The various methods of reuse that are available to test suites

- How to build variants of `describe` and `it` on top of the existing test runner design

Let's start with a look at methods of reuse.

Methods of Reuse

It's a given that we don't want to litter our codebases with repeated code. We want to find ways to *reuse* code snippets in many places.

It's crucially important to realize that application code and test code have very different means of reuse.

This comes about because they have entirely different outcomes. The application code is the thing that manipulates the system in complex ways. It is mainly a battle of *abstractions*: finding the elegant ways to express

changes and updating those abstractions as new features are added (and removed). But test suites are *scripts*: they run from top to bottom and tell a story.

This chapter introduces *shared examples*. Table 11-1 shows this as a mechanism for reusing code, alongside the other mechanisms you've already learned.

Table 11-1. *Methods of code reuse for application code vs. test suites. Test suites need a completely different set of practices to ensure future maintainability*

Application Code	Test Code
Functions	`beforeEach` and `afterEach`
Objects	Matchers
Modules	Factories and builders
Design patterns	Shared examples

The concept of "shared" examples is a little bit like inheritance in classes. In fact, they are most often used to test class hierarchies. We can use them to repeat tests in subclasses that define behavior that the superclass provides.

Just like how subclasses can share data with their superclasses via a call to `super` in the constructor, shared examples are passed contextual data from the describe block that pulls them. Let's take a look at how that works.

Here's the new content of `test/TodoRepository.tests.mjs`, which you can update now:

Listing 11-1. todo-example/test/TodoRepository.tests.mjs

```
describe("TodoRepository", () => {
  const newTodo = { ...emptyTodo(), title: "test" };
  let repository;
```

```
  beforeEach(() => {
    repository = new TodoRepository();
  })

  it.behavesLike("a list", () => ({
    type: repository
    entry: newTodo
  }));

  // ... existing tests here ...
});
```

It's important that the context is specified by a function. That means it can be re-evaluated for each test in the shared example, after each of the beforeEach blocks has run. In the preceding case, that means that the shared examples always get a new repository instance.

What you can see clearly from the preceding example is that "a list" identifies the shared examples to use.

So how can we define "a list"? The following listing shows the content of test/list.shared.tests.mjs, which you can create now in the sample application repository:

This sample makes use of the toBe matcher, which hasn't been covered by the book but is included in the starting code for this chapter.

Listing 11-2. todo-example/test/list.shared.tests.mjs

```
import { describe, it, expect } from "concise-test";

describe.shared("a list", () => {
  describe("adding entries", () => {
    it("returns true on success", ({ type, entry }) => {
      expect(type.add(entry)).toBe(true);
    });
```

```
  it("throws error when adding undefined", ({
    type,
  }) => {
    expect(() => type.add(undefined)).toThrow();
  });
});
});
```

Here's how that will look by the time we've completed this chapter:

```
TodoRepository
  a list (shared)
    adding entries
      ✓ returns true on success
      ✓ throws error when adding undefined
  add
    ✓ throws an exception when adding a todo without a title
    ✓ throws errors when adding a repeated todo
  findAllMatching
    ✓ finds an added todo
6 tests passed, 0 tests failed.
```

Implementing a Shared Example Repository

Create a new file src/sharedExamples.mjs and add the following:

Listing 11-3. concise-test/src/sharedExamples.mjs

```
let sharedExamples = {};

export const registerSharedExample = (name, body) => {
  sharedExamples = {
    ...sharedExamples,
```

```
    [name]: body,
  };
};
```

```
export const findSharedExample = (name) =>
  sharedExamples[name];
```

These two functions make up the shared example registry. It's a very simple map, keyed by a string. We'll wire up `describe.shared` to call `registerSharedExample`, and we'll wire up `it.behavesLike` to call `findSharedExample`.

Before we do that, we've got two more functions to define in this file (only one of which is exported):

Listing 11-4. concise-test/src/sharedExamples.mjs

```
const findContextFn = (stack) =>
  stack
    .map((block) => block.sharedContextFn)
    .find((fn) => fn);
```

```
export const buildSharedExampleTest = ({
  body,
  describeStack,
}) => {
  const sharedContextFn = findContextFn(describeStack);
  return sharedContextFn ? () => body(contextFn()) : body;
};
```

The `buildSharedExampleTest` will be used by `runWithIt` to produce a special `body` that can be slotted into the existing design: notice how it calls a `sharedContextFn` before running the `it` body. This `contextFn` is set up with the call to `it.behavesLike`.

Open up `src/testContext.mjs,` and let's get on with implementing `describe.shared`. First, import the three new functions we've just defined:

Listing 11-5. concise-test/src/testContext.mjs

```
import {
  registerSharedExample,
  findSharedExample,
  buildSharedExampleTest,
} from "./sharedExamples.mjs";
```

Then, just after the timeout code, add this line:

```
addModifier(describe, "shared", registerSharedExample);
```

That's all that's necessary for `describe.shared`. The `it.behavesLike` function is slightly more complicated:

Listing 11-6. concise-test/src/testContext.mjs

```
const behavesLike = (name, sharedContextFn) =>
  describeWithOpts(name, findSharedExample(name), {
    sharedContextFn,
  });

addModifier(it, "behavesLike", behavesLike, {});
```

We introduce the `sharedContextFn` for the first time. This is passed as an option to `describeWithOpts`, which is the same functionality we used for focus behavior.

Next, update `runIt` to call our build function. In the following example, we create a new `wrappedBody` and use that in place of `body`:

Listing 11-7. concise-test/src/testContext.mjs

```
const runIt = async (test) => {
  global.currentTest = test;
  currentTest.describeStack = [...describeStack];
  const wrappedBody = buildSharedExampleTest(currentTest);
```

182

```
  try {
    invokeBefores(currentTest);
    await runBodyAndWait(wrappedBody);
    invokeAfters(currentTest);
  } catch (e) {
    currentTest.errors.push(e);
  }
  dispatch("finishedTest", currentTest);
  global.currentTest = null;
};
```

Let's update our reporter to highlight a shared example block. Open `src/reporters/default.mjs` and update the listener for the `beginningDescribe` event as follows:

Listing 11-8. concise-test/src/reporters/default.mjs

```
listen(
  "beginningDescribe",
  (describeStack, { name, sharedContextFn }) => {
    if (sharedContextFn) {
      console.log(
        indent(
          describeStack,
          color(`${name} (<cyan>shared</cyan>)`)
        )
      );
    } else {
      console.log(indent(describeStack, name));
    }
  }
);
```

You can use the examples at the start of the chapter to test this out. Make sure you use `npm test`, however.

What happens if you try to run just this test suite, with the command `npm test test/TodoRepository.tests.mjs`?

```
> todo-example@1.0.0 test
> concise-test test/TodoRepository.tests.mjs

body is not a function

  in test/TodoRepository.tests.mjs:

  13 |    });
  14 |
  15 |    it.behavesLike("a list", () => ({
     |        ^
  16 |      type: repository,
  17 |      entry: newTodo
```

When running a single file, the shared examples–noted by the `"a list"` argument–haven't been loaded by the test runner.

Importing Shared Examples

If you want to use `it.behavesLike` when using single-file support, you'll need to import the shared examples file explicitly, like this:

```
import * as shared from "./list.shared.tests.mjs";
```

This is a bit of a trap that unwitting developers will undoubtedly fall into. Your IDE will have no indication of the link between the shared examples name (`"a list"`) and any imports (from "`./list.shared.tests.mjs`").

So how about we improve our implementation with a helpful error message?

Listing 11-9. concise-test/src/testContext.mjs

```
const behavesLike = (name, sharedContextFn) => {
  const sharedExample = findSharedExample(name);
  if (!sharedExample)
    throw new Error(
      `The shared context "${name}" was not found. Have
      you imported the file containing the shared context
      definition?`
    );
  describeWithOpts(name, findSharedExample(name), {
    sharedContextFn,
  });
};
```

With this change, running the command `npm test test/` `TodoRepostiory.tests.mjs` will result in the following:

```
> todo-example@1.0.0 test
> concise-test test/TodoRepository.tests.mjs

The shared context "a list" was not found. Have you imported
the file containing the shared context definition?

  in test/TodoRepository.tests.mjs:

 13 |   });
 14 |
 15 |   it.behavesLike("a list", () => ({
    |      ^
 16 |     type: repository,
 17 |     entry: newTodo
```

That's much better, isn't it? The situation is still not ideal, but at least we've anticipated and mitigated this potential trap for our users.

Exercise

1. Implement a new mode for your test runner that takes the *name* of a shared context and runs all the test suites that include it. The command would look as follows:

```
npm test --behavesLike "a list"
```

Discussion Questions

1. With great power comes great responsibility. Why do you think shared examples can make test suites *less* maintainable?

2. Imagine that you have a set of objects that all have common behavior that you'd like to test using shared examples. How could you modify your *application code* so that the problem disappears and you can easily test your application without relying on shared examples?

Summary

This chapter has provided an example of how making test suites maintainable can be vastly different from making application code maintainable, because the methods of reuse are different.

Shared examples aren't a feature you'll be reaching for every day, but they're an important feature nonetheless.

In the next chapter, we'll look at a substantially different feature and one that isn't all that common in JavaScript frameworks: tagging.

Tagging Tests

When test suites become large, they can become slow to run. *Tagging* gives us a mechanism for slicing and dicing test suites in ways that allow them to be run differently depending on the run context. You can associate a group of tests with a tag of your choosing (`ui`, `e2e`, `slow`, etc.) that you can then use on the command line to run only these tests (or to *avoid* running these tests).

This is also useful because a generalized automated test runner, like the one we're building, can be used for many different types of test, ranging from unit tests that require no special setup to full system tests that require multiple other processes to be running that the test runner interacts with using some form of inter-process communication (IPC).

In this chapter, we'll build new forms of the `describe` and `it` functions that allow test suites and test cases to be annotated with one or more tags.

You'll see how to

- Implement function overloads with differing parameters.

- Use the command line to supply tags for filtering tests.

Let's take a closer look at tags when applied to test suites.

Another Way of Slicing Tests

In any sufficiently complex codebase, you will come across groups of files that are all related in some way. They are *coupled*: when one of those files changes, the others are likely to change too; when one test suite breaks, the others are likely to break too.

For example, React components often have a set of files that are needed for the component to function, like files for API calls or files related to a Redux store.

Wouldn't it be great if we could run *all* the files related to the single component?

We already have the ability to run all files or a single file. In this chapter, we'll add the ability to specify a tag (or multiple tags):

```
npm test --tags foo
```

This command runs only tests and describe blocks that are tagged with `"foo."`

Figure 12-1 shows examples of common tags that you might encounter when working in modern application test suites.
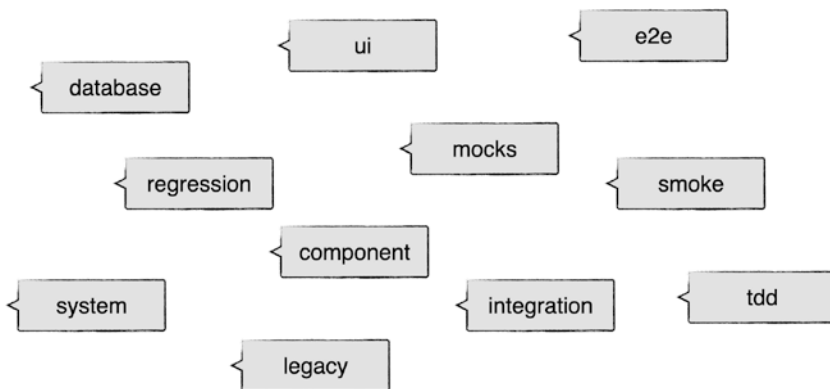
Figure 12-1. *Examples of common tags that are used in automated test suites*

If you think through your current usage patterns of your automated test runner, you may be surprised how many can be simplified with the use of tags.

Thinking Through Design Options

Jest's Approach

Jest has a feature that allows you to run tests that have a name matching a given regular expression. This allows you to encode your own tagging system within that name. However, I can't think of a use for this feature other than tagging, and in that case it feels better to me to make tags a defined part of the design.

Adding tags in a test case description can encourage bad test naming practices. Good test names should be human-readable strings that accurately describe the behavior that the test case exercises. A tag is an entirely separate thing. Terse, descriptive, thoughtful, and well-written names are one of the weapons you have for quickly diagnosing failures. Don't shoot yourself in the foot!

How about the approach shown as follows?

```
describe.tagged("my block", { tags: [ "foo" ] }, () => {
  // ... describe body here
}

it.tagged("my test", { tags: [ "foo" ] }, () => {
  // ... test body here
})
```

There are some big problems with this approach. How would you focus tests with tags? What about shared examples? Do we somehow add a complex mechanism for chaining modifiers?

```
it.only.tagged("my test", { tags: ["foo"] }, () => {
  // ... test body here
});
```

Chaining modifier functions doesn't sound like a great idea to me—the implementation will be anything but simple. The modifiers we've created already—only, shared, and behavesLike—are all mutually exclusive. In other words, it's unlikely you'll want to combine them. But you can imagine that you *would* want to combine tags and the only modifier, for example.

(And in the next chapter, we'll add another modifier, skip, which is essentially the opposite of only. They'd never be combined.)

Let's look at another design idea. In Chapter 9, we wrote a helper like this:

```
it.timesOutAfter(5000);
```

So how about a helper just like that?

```
it.isTaggedAs("foo");
```

Now this is more like it. But the problem with this style of modifier is that it's invoked as part of the test, and our tags need to be parsed *before* the test runs.

Here's another option, which isn't ideal for a simple reason I'll outline in the following, but it's better than the preceding two solutions. It redefines describe and it to take three parameters rather than two: the first parameter is still the descriptive name, but the second is now a set of options, and the third becomes the body of the test:

```
describe("my block", { tags: [ "foo" ] }, () => {
  // ... describe body here
}

it("my test", { tags: [ "foo" ] }, () => {
  // ... test body here
})
```

Assuming we want the original forms to still work, these new forms of the `describe` and `it` functions are now *polymorphic* in their second and third arguments. Since JavaScript is a dynamic language and doesn't have function overloads–where the same function can be defined with multiple versions that differ on the parameter types–we have to do a type check:

```
if (secondArg instanceof Function)
  ...
```

Yuck! A type check! A rule of good programming is avoid type checks. Type checks are a conduit for complexity. But another rule of good programming is everything is a trade-off. And based on all the other solutions we've considered, this seems like the best of all evils.

This way, you'll be able to use modifiers and specify tags at the same time.

In the next sections, here's what we'll build:

- New forms of the `describe` and `it` functions that take an `opts` object as the second argument, followed by the body function as the third argument

- Filtering test runs based on the tags property in the new `opts` object

- Reading tags from the process arguments

Finally, we'll finish up with updating the sample application with some tags.

Supporting Polymorphic Calls to Test Functions

There are two types of calls to the describe and it functions that we want to support. The first is the traditional form:

```
describe(
  "test name",
  () => { /* test body */ }
)
```

The second is the new form, where the second parameter becomes our options:

```
describe(
  "test name",
  { tags: [ "ui"] },
  () => { /* test body */ }
)
```

It's generally unwise to design any public interface where types can be *either* one thing *or* another thing. Dynamic languages like JavaScript make it all too easy to do this, but it can leave your code in a horrendous mess of conditionals.

A good tactic to avoid that is to do the conditional work *immediately* upon the function call, converting the parameters into a standardized form.

That's what we'll do here, by defining a chooseOptions function and a chooseBody function. These are the functions that contain the type checks, and they are called right at the start of the function processing.

In src/testContext.mjs, add the following two definitions, just below the imports.

Listing 12-1. concise-test/src/testContext.mjs

```
const chooseOptions = (eitherBodyOrOpts) => {
  if (eitherBodyOrOpts instanceof Function) {
    return {};
  } else {
    return eitherBodyOrOpts;
  }
};

const chooseBody = (eitherBodyOrOpts, bodyIfOpts) => {
  if (eitherBodyOrOpts instanceof Function) {
    return eitherBodyOrOpts;
  } else {
    return bodyIfOpts;
  }
};
```

Note how chooseOptions returns an empty object if eitherBodyOrOpts is a body function, rather than null or undefined. That will help keep the rest of our implementation simple.

Next, redefine the describe function as shown in the following. It makes use of the two new helpers, passing them into the exiting describeWithOpts function. You'll recall that we built this function to handle *modifier* options. We're now reusing that function with our *user* options (named as such because they're specified by the user).

Listing 12-2. concise-test/src/testContext.mjs

```
export const describe = (
  name,
  eitherBodyOrOpts,
  bodyIfOpts
) =>
```

```
describeWithOpts(
  name,
  chooseBody(eitherBodyOrOpts, bodyIfOpts),
  chooseOptions(eitherBodyOrOpts)
);
```

Then do the same for the it function.

Listing 12-3. concise-test/src/testContext.mjs

```
export const it = (name, eitherBodyOrOpts, bodyIfOpts) =>
  itWithOpts(
    name,
    chooseBody(eitherBodyOrOpts, bodyIfOpts),
    chooseOptions(eitherBodyOrOpts)
  );
```

Okay, now for the tricky bit. What if we're using the new form of it but with a modifier function, like the it.only example shown previously?

We need to *merge* the modifier options and the user options. To do that we'll define a new function, mergeModifierOptsIntoUserOpts, that you add just above the definition of the addModifier function.

Listing 12-4. concise-test/src/testContext.mjs

```
const mergeModifierOptsIntoUserOpts = (
  eitherBodyOrUserOpts,
  modifierOpts
) => ({
  ...chooseOptions(eitherBodyOrUserOpts),
  ...modifierOpts,
});
```

Then make use of that by redefining the `addModifier` function as shown in the following.

Listing 12-5. concise-test/src/testContext.mjs

```
const addModifier = (
  object,
  property,
  fn,
  modifierOpts
) =>
  Object.defineProperty(object, property, {
    value: (name, eitherBodyOrOpts, bodyIfOpts) =>
      fn(
        name,
        chooseBody(eitherBodyOrOpts, bodyIfOpts),
        mergeModifierOptsIntoUserOpts(
          eitherBodyOrOpts,
          modifierOpts
        )
      ),
  });
```

This is hairy, that's for sure. But at least we've contained the hairiness to a dozen lines of code, and we developed language (modifier and user options) that hopefully clears up what's going on.

The modifier options take precedence, so if you've specified `{ focus: false }` as a user option but also used `describe.only`, then your describe block will still be focused.

Now that we can pass user options in, let's create a handler for a `tags` option.

Filtering Test Runs Using the tags Property

In Chapter 8, you saw one way of filtering tests, with the focusedOnly
function. Let's copy that approach with a new taggedOnly function.

Create a new file called src/tags.mjs and within it add the following
function, taggedOnly. The purpose of this function is to recursively walk
through a tree of describe and it blocks, filtering out any block that is not
tagged with one of the input tags.

Listing 12-6. concise-test/src/tags.mjs

```
const tagMatch = (tagsToSearch, tagsToFind) =>
  tagsToSearch.some((tag) => tagsToFind.includes(tag));

export const taggedOnly = (tags, block) => {
  if (!tags) {
    return block;
  }

  if (block.tags && tagMatch(block.tags, tags)) {
    return block;
  }

  if (!block.children) {
    return null;
  }

  const taggedChildren = block.children
    .map((child) => taggedOnly(tags, child))
    .filter((child) => child);

  if (taggedChildren.length > 0) {
    return {
      ...block,
      children: taggedChildren,
```

```
    };
  } else {
    return null;
  }
};
```

If a block is marked with a filtered tag, then it is immediately returned along with all its children. But if a block doesn't have the filtered tag, it might still be returned, if any of its children have the tag. That's why we need to recursively call taggedOnly on its children.

Now it's time to redefine runParsedBlocks to make use of the new taggedOnly function. Open src/testContext.mjs and import the new function:

```
import { taggedOnly } from "./tags.mjs";
```

Next, modify the runParsedBlocks function to take a new tags parameter and change const withFocus to be let filtered. Once we've focused tests, we then call taggedOnly.

Listing 12-7. concise-test/src/testContext.mjs

```
export const runParsedBlocks = async ({ tags }) => {
  let filtered = focusedOnly(currentDescribe);
  filtered = taggedOnly(tags, filtered);
  for (let i = 0; i < filtered.children.length; ++i) {
    await runBlock(filtered.children[i]);
  }
  return anyFailed(filtered);
};
```

The final step is to make sure runParsedBlocks is called with the new tags argument, which is was we do in the next section.

Reading Tags from the Process Arguments

Recall that we want to read the process arguments and use those as the tags. We look for a first argument of `--tags` and then expect the next argument to be a comma-separated list of tags.

Open `src/runner.mjs` and create a new function, readTags.

Listing 12-8. concise-test/src/runner.mjs

```
const readTags = () => {
  const tagArgIndex = process.argv.findIndex(
    (t) => t === "--tags"
  );
  if (tagArgIndex > -1) {
    return process.argv[tagArgIndex + 1]
      .split(",")
      .map((tag) => tag.trim());
  }
};
```

I've also chosen to call `trim` so that the user can use their shell's command-line support for formatting this argument as they wish. It gives them a little bit more freedom, for no design pain.

If our command-line arguments got more complicated than this, I'd consider using a library to parse command-line arguments.

Update the run function to call `readTags`:

```
const failed = await runParsedBlocks({
  tags: readTags(),
});
```

There's just one more modification that needs to be made. We've now broken the function `isSingleFileMode`. Update that to return false if the first parameter begins with a hyphen:

```
const isSingleFileMode = () =>
  process.argv[2] && !process.argv[2].startsWith("-");
```

And we're done. Time to (manually) test.

Adding Tags to the Sample Application

The first test you'd want to make is to tag the `todo` and `TodoRepository` describe blocks as `todo`, leaving the async tests alone. So in `test/todo.tests.mjs`:

```
describe("todo", { tags: ["todo"] }, () => {
  // .. existing code
});
```

Then go ahead and try

```
npm test --tags todo
```

There's a whole bunch of other manual tests you might want to try:

- Assigning multiple tags to a single test should still run the test.

- Focusing on a tagged test should still run the test.

- Focusing on a non-tagged test should run nothing.

- Assigning a non-`todo` tag to a parent describe should still run the tagged test within the parent describe.

Exercises

1. Add the ability to filter *out* specific tags by using the ~ operator, like this:

   ```
   npm test --tags ~ui ~e2e
   ```

2. While the API and overall design feels sound, there are a bunch of loose ends that can be improved. Improve upon the code presented in this chapter, using these questions as a starting point:

   - Are there better abstractions lurking within our codebase?

   - Can we deal with command-line arguments in a simpler way?

   - How can we report on which tags were run?

Discussion Question

1. Look at the tags shown in the "tag cloud" in Figure 12-1. Which of these tags would you find useful in your current project's test suites?

Summary

This chapter has shown how tags can be added to both test suites and tests as a way to support multiple "types" of test.

There is an analogy here with test directories, and to an extent you could use directories as a simple form of tagging.

For example, you could place some tests in the `test/unit` directory and some in the `test/system` directory, to signify the difference in test files contained in each directory.

But while a test suite can exist in only one directory, it can have multiple tags: `system`, `ui`, `long-running`, and so on.

In the next chapter, we'll implement another modifier to the `describe` and `it` functions: skipping tests with `describe.skip` and `it.skip`.

Skipping Tests

This chapter contains an implementation for a new modifier function that is invaluable when using automated tests as part of your daily workflow: the ability to skip tests.

The idea is that the user can avoid a test being run (perhaps because they haven't written it yet) by renaming `it` to `it.skip`. The same can be done for `describe`, which becomes `describe.skip`.

When the tests are reported, we'll make this obvious to the user by using strikethrough text and a yellow question mark next to the test name.

Finally, we'll add a modifier function that allows us to skip both `describe` and `it` by simply not passing a body. This is useful if you want to write yourself a *reminder* to write a test, but defer writing it until later.

In this chapter, you will uncover

- Why skipping tests is important

- How to implement `describe.skip` and `it.skip` modifier functions

We start with a look at a typical developer workflow that makes use of skipping tests.

Taming the Test Runner

In Chapter 7 we discussed developer workflows and how the test runner has an important role to play in simplifying your work.

Skipping tests is another one of these useful operation modes that helps when refactoring. It is essentially the opposite of *focusing* tests that we look at it in Chapter 8, building out the `only` modifier. Figure 13-1 shows how your own workflow might look.
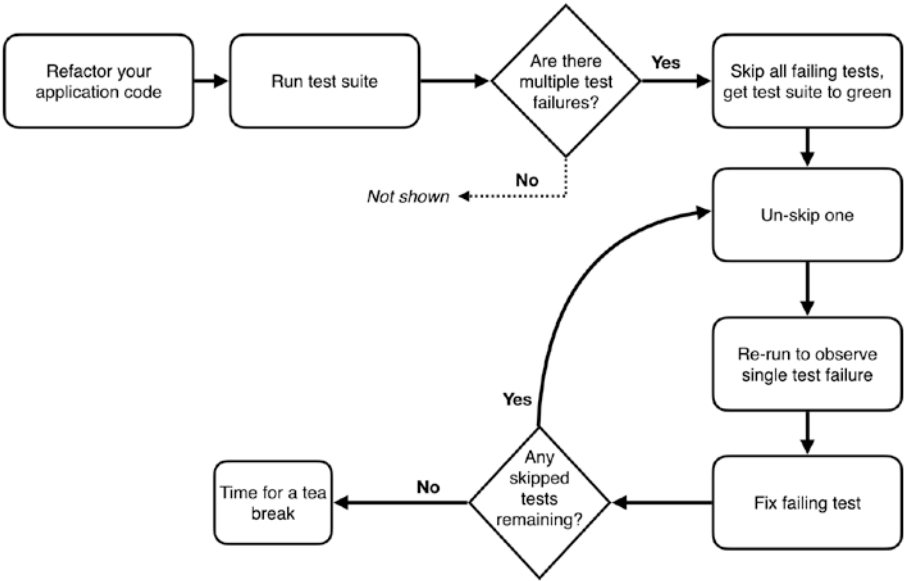
Figure 13-1. *A workflow showing how to make use of skipped tests when refactoring application code*

Interestingly, the ability to skip tests is often the most abused of all test runner features.

It's very common to find test suites that have permanently skipped tests. This is the test suite's version of the pesky `TODO` comment. It's almost as if you write it, you're guaranteeing that the code will never be written. And when you skip a test because it doesn't work *today* but might *tomorrow* if someone just finds the time to fix it, then you're almost guaranteeing that the test will remain skipped forever.

In this scenario, you should reach for the Delete key instead. You can always get the test back from your commit history.

Adding the it.skip Modifier

Let's get started with adding the new modifier function.

In `src/testContext.mjs` add the following modifier function definitions:

```
addModifier(it, "skip", itWithOpts, { skip: true });
addModifier(describe, "skip", describeWithOpts, {
  skip: true,
});
```

Update `runDescribe` to break early if the `skip` property is set. At the same time, dispatch a `skippingDescribe` event.

Listing 13-1. concise-test/src/testContext.mjs

```
const runDescribe = async (describe) => {
  if (describe.skip) {
    dispatch("skippingDescribe", describeStack, describe);
    return;
  }
  // ... existing code here ...
};
```

Now we repeat that exercise with `runIt`. Add the guard clause right at the top of the function, before `currentTest` is set.

Listing 13-2. concise-test/src/testContext.mjs

```
const runIt = async (test) => {
  if (test.skip) {
    dispatch("skippingTest", test);
    return;
  }
  // ... existing code here ...
};
```

Over in `src/reporters/default.mjs`, we need to listen for the new events.

Listing 13-3. concise-test/src/reporters/default.mjs

```
listen("skippingDescribe", (describeStack, { name }) => {
  console.log(
    indent(
      describeStack,
      color(`<strike>${name}</strike>`)
    )
  );
});

listen("skippingTest", ({ describeStack, name }) => {
  console.log(
    indent(
      describeStack,
      color(`<yellow>?</yellow> <strike>${name}</strike>`)
    )
  );
});
```

Believe it or not… that's it!

Testing It Out

In your sample application, open `test/todo.tests.mjs` and add `.skip` to the first `it` and to the second `describe`.
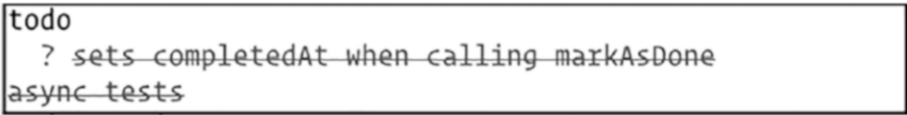
Listing 13-4. todo-example/test/todo.tests.mjs

```
describe("todo", { tags: ["todo"] }, () => {
  it.skip("sets completedAt when calling markAsDone", () => {
    const todo = emptyTodo();

    expect(markAsDone(todo).completedAt).toBeDefined();
  });
});

describe.skip("async tests", () => {
  it("waits for a little bit", async () => {
    it.timesOutAfter(2000);
    await new Promise((resolve, _) =>
      setTimeout(resolve, 5000)
    );
  });
});
```

Given the preceding change, Figure 13-2 shows what you should see on-screen when you run your tests with the npm test command.[1]

```
todo
  ? sets completedAt when calling markAsDone
async tests
```

Figure 13-2. *Skipped tests in a test run*

---

[1] Not all terminals support strikethrough text, unfortunately.

Supporting Empty-Body describe and it Calls

This is also quite straightforward, if a little messy.

Back in `src/testContext.mjs`, update the `runIt` conditional to check for an undefined body:

```
if (test.skip || !test.body) {
  dispatch("skippingTest", test);
  return;
}
```

Support for `describe` works slightly differently because the `describe` body is invoked at parse time. That means we have to take some action during the parse phase.

Instead, we use the absence of a body to set an initial value for the `skip` property.

Listing 13-5. concise-test/src/testContext.mjs

```
const describeWithOpts = (name, body, options) => {
  const parentDescribe = currentDescribe;
  currentDescribe = makeDescribe(name, {
    skip: body === undefined,
    ...options,
  });
  if (body) {
    body();
  }
  currentDescribe = {
    ...parentDescribe,
    children: [
      ...parentDescribe.children,
```

```
      currentDescribe,
    ],
  };
};
```

Let's try it out. Create a new file in your sample application, `test/skipped.tests.mjs`, with the following content.

Listing 13-6. todo-example/test/skipped.tests.mjs

```
import { describe, it } from "concise-test";

describe("skipped describe");

describe("with skipped children", () => {
  it("skipped 1");
  it("skipped 2");
});
```

Run your tests with `npm test`. With any luck you'll see the two new test suites listed and marked as skipped.

Exercises

1. Update the no-body version of `it` and `describe` so that they are *not* shown in test output, except for the final reporting line, thereby solving the issue highlighted in the preceding section.

2. Implement an alias of the no-body feature from Exercise 1, named `it.todo` and `describe.todo`.

3. Write unit tests for the `it` function.

4. Write unit tests for the `describe` function.

Discussion Question

1. Very often, teams get in the habit of skipping tests that don't *currently* pass. Usually this happens when there's an intention to fix these test cases in the future. More often than not, these tests then become noise that litter the test runner output every time the tests run. In this case, do you think it's better to delete the test case entirely? What other solutions can you think of?

Summary

In this chapter you have learned why skipping tests is an important part of any test workflow, and you've seen a basic implementation of `describe.skip` and `it.skip` functions.

The `runDescribe` and `runIt` functions are starting to become quite complicated. These functions would surely benefit from having automated tests to specify their behavior. Plus, there's plenty of opportunity to refactor and simplify.

In the next chapter, we'll add support for randomizing test runs, which is a useful check that our tests are indeed independent.

Randomizing Tests

No matter how experienced you are with test writing, it's all too easy to inadvertently write interdependent tests that share *state*.

Shared state can mean global variables, database records, and so on.

Having interdependent tests can mean that your tests pass but *only* because of the setup or actions done in a previous test. You might think your test is working correctly, but in actual fact it's not doing anything.

That can mean you end up with test failures or tests that don't test what you think to do, because the *Arrange* phase doesn't accurately describe the setup of the test.

Your most important weapon to fight test interdependence is to reduce all shared state between tests. Have no shared state if possible. Any test setup should be done in your `beforeEach`, and your production code shouldn't rely on any global state or any objects that aren't instantiated in either a `beforeEach` or the test itself.

That's one reason we haven't built a `beforeAll` block–using it is something of a risk, for this very reason.

Another weapon against this is randomizing test runs. Instead of running your tests in the order they appear in a file, they are first parsed and then shuffled, so that they run in a random order. That means that their initial state will differ between each test run. Any inadvertent state pollution will be more likely to fail during a random test run. More likely, but not certain.

In this chapter we'll add a new command-line flag, `--randomize`, that switches the order of our tests.

By the end of the chapter, you will

- Understand the scenarios in which test randomization is useful and when it isn't.

- Have implemented a CLI flag to randomize test order using the `Math.random()` function.

Let's start with looking at testing workflows (again).

Test Runner Use Cases

In previous chapters we discussed the role of the test runner in support of personal developer workflows. Chapter 7 covered how to use file path specifiers to choose specific sets of tests. Chapter 8 covered the `only` modifier to support refactoring. Chapter 13 covered the `skip` modifier, which is the opposite of `only`.

Besides personal developer workflows, the test runner also supports another user: the Continuous Integration (CI) environment.

Randomizing Tests in CI Environments

The primary responsibility of the test runner in a CI environment is to ensure that *all* tests pass and fail builds that do not. This should block any breaking changes from being merged into the production branch.

It's in this context–running *all* tests–that randomization of tests makes sense. It helps catch against developer errors, where some tests might accidentally depend on the result of a previous test. It turns out that randomization is actually quite effective at doing this, because chances are high that any two tests will return in a different order than they were written in a test file.

Randomizing Tests on Developer Machines

For personal developer workflows, which are very often focused on a single file or single set of tests, randomization can be annoying. Often, you're interested in the test that *first* failed in a file, since test dependencies tend to move downward from top to bottom.

Tests lower down in a file are more likely to depend on higher tests than the other way round. Also, tests higher up are likely to be *simpler* than tests lower down, which tend to be the edge cases or the unhappy paths.

So it follows that if you've got a bunch of failures in a single test suite, you should start by fixing the *first* failing test.

Therefore, randomization can get in the way of helping you see which test you should be fixing first.

Adding the Flag

In this section we'll add the new `randomize` flag to the test runner.

Open `src/runner.mjs` and add the following new function.

Listing 14-1. concise-test/src/runner.mjs

```
const readRandomFlag = () => {
  if (process.argv.find((t) => t === "--randomize")) {
     return true;
  }
};
```

Then update the call to `runParsedBlocks` in `run` to pass through this flag:

```
const failed = await runParsedBlocks({
  tags: readTags(),
  shouldRandomize: readRandomFlag(),
});
```

213

To make use of this flag, we'll define a new type of filter. Create a new file named `src/randomize.mjs` and add the following content. This uses a Fisher-Yates shuffle to take the `children` array and shuffle it–for both describe blocks and tests.

Listing 14-2. concise-test/src/randomize.mjs

```
const shuffle = (arr) => {
  const newArr = [...arr];
  for (let i = newArr.length - 1; i > 0; --i) {
    let j = Math.floor(Math.random() * (i + 1));
    [newArr[i], newArr[j]] = [newArr[j], newArr[i]];
  }
  return newArr;
};

export const randomizeBlocks = (
  shouldRandomize,
  block
) => {
  if (!shouldRandomize) {
    return block;
  }

  if (!block.children) {
    return block;
  }

  const randomizedChildren = block.children.map((child) =>
    randomizeBlocks(shouldRandomize, child)
  );

  return {
    ...block,
```

```
  children: shuffle(randomizedChildren),
  };
};
```

Now all that's left is to update `runParsedBlocks` to call this new filter. Open `src/testContext.mjs` and add the new import:

```
import { randomizeBlocks } from "./randomize.mjs";
```

Then update runParsedBlocks.

Listing 14-3. concise-test/src/testContext.mjs

```
export const runParsedBlocks = async ({
  tags,
  shouldRandomize,
}) => {
  let filtered = focusedOnly(currentDescribe);
  filtered = taggedOnly(tags, filtered);
  filtered = randomizeBlocks(shouldRandomize, filtered);
  for (let i = 0; i < filtered.children.length; ++i) {
    await runBlock(filtered.children[i]);
  }
  return anyFailed(filtered);
};
```

Testing It Out

Go to your sample application and run tests with the following command:

```
npm test -- --randomize
```

You should see everything in a shuffled order. Hopefully this is enough to convince you that they are in fact running in a different order!

Exercises

1. Implement a feature that saves the test run order to a file on disk, but *only* if there were test failures. This file can then be read to rerun tests in the *same* order, so that the failure can be reproduced in the scenario that the failure only occurs *because of* that test order.

2. Our test runner now has three clear phases: parse, filter, and run. However, there is no real explicit recognition of that in the codebase. If you were reading the codebase for the first time, you might not pick up on this architecture. Refactor the code so that it makes it clear how the three-phase approach is used during a test execution run.

Discussion Question

1. It can be argued that randomizing test runs during local development (i.e., not on a CI environment) does more harm than good. Why would this be the case? How does it affect a development workflow such as the Test-Driven Development (TDD) workflow?

Summary

In this chapter you've seen how to implement a new CLI flag that can randomize the order that tests are run, and you've seen why this is important.

Adding this feature has solidified the "filter" phase of our tests, and our test runner is looking complete.

For the final chapters of the book, we'll switch track and look at what's arguably the most misunderstood feature of automated testing: test doubles.

PART IV

Test Doubles and Module Mocks

We end the book with a look at unit testing's most misunderstood feature, the humble test double.

In Chapter 15, "Deep Equality and Constraining Matchers," we do the groundwork for supporting test double verifications.

In Chapter 16, "Test Doubles," we explore the theory behind test doubles and implement a spy function, together with a matcher for verifying spy invocations.

In Chapter 17, "Module Mocks," we finish the book with a look at how entire modules can be replaced by using a Node custom loader.

CHAPTER 15

Deep Equality and Constraining Matchers

JavaScript is one of many languages that makes it easy to create object-like map structures with expressive syntax:

```
{
  customerId: 123,
  order: {
    time: 234,
    value: 56.99,
    items: [
      { id: 345, unitPrice: 5.00, quantity: 2 },
      { id: 456, unitPrice: 46.99, quantity: 1 }
    }]
  }
}
```

Often, objects like this are returned from functions and API calls, which means it helps to have a matcher that can match the values within this entire structure.

Even more importantly, it helps to have a matcher that can match *some* of this structure. This is crucially important to building maintainable tests, because often large return values have logical groupings of values (like *customer*, *order*, and *items* in the preceding example).

In this chapter you will

- Build an `equals(l, r)` function that returns `true` if `l` and `r` are *deeply* equal, by recursively working its way through objects and arrays checking that all keys and primitive values have equal values.

- Build the `anObjectContaining` constraint function that can be used with `equals`.

This chapter is mainly a preparation for Chapter 16 that makes use of the `anObjectContaining` constraint function in the testing of a `global.fetch` call.

However, this topic is useful as a standalone topic because it has a number of Exercises and Discussion Questions. The building of the `toEqual` matcher itself is Exercise 1, which you will be able to do with the knowledge from Chapter 5.

How Are Constraining Functions Useful?

We use *constraining functions* to zone in on the bit of an object that we're interested in. The next two tests show what tests might look like for the object structure shown at the start of this chapter:

```
it("returns the customer id", () => {
  expect(apiResponse).toEqual(
    anObjectContaining({ customerId: 123 })
  );
});
```

```
it("returns the order items", () => {
  expect(apiResponse).toEqual(
    anObjectContaining({
      order: anObjectContaining({
        items: [
          { id: 345, unitPrice: 5.0, quantity: 2 },
          { id: 456, unitPrice: 46.99, quantity: 1 },
        ],
      }),
    })
  );
});
```

The two other constraining functions that are generally useful are anArrayContaining, which matches any array that contains the same items in any order, and anArrayContainingExactly, which matches arrays that contain *only* these items and no others. We won't make use of them in this book.

Let's get started with the equals function.

Implementing the equals Function

Aside from the recursion, there are three essential operations in a deep equality function:

1. Checking that two primitive values (numbers, strings, Booleans) are equal. In JavaScript, this can be done with the === operator.

2. Checking that two arrays are equal. This is done by checking they are of equal size and that the two items at each index are the same.

3. Checking that two objects are equal. This is done by
    checking that they have the same number of keys
    and that each keyed item in the first collection is
    equal to the same keyed item in the second.

In this section we will start by building an arraysWithSameLength
function. Then we'll build an objectsWithSameKeys function and then
finish it off with the equals function itself.

Checking if Two Arrays Have Equal Size

Create a new file in the concise-test project named src/equals.mjs and
give it the following content.

Listing 15-1. concise-test/src/equals.mjs

```
const arraysOfEqualLength = (l, r) =>
  Array.isArray(l) &&
  Array.isArray(r) &&
  l.length === r.length;
```

That's all there is to it, but note the use of l and r as shorthands for
left and right.

Checking if Two Objects Have Equal Keys

Next, add the following two functions.

Listing 15-2. concise-test/src/equals.mjs

```
const keysEqual = (l, r) => {
  const lks = Object.keys(l);
  const rks = Object.keys(r);
```

```
  if (lks.length !== rks.length) return false;

  return lks.every((lk) => rks.includes(lk));
};
const objectsWithSameKeys = (l, r) =>
  typeof l === "object" &&
  typeof r === "object" &&
  keysEqual(l, r);
```

We use a separate function named keysEqual here to check the key arrays.

It should be clear that both arraysWithSameLength and objectsWithSameKeys don't actually check the values. For that we need to bring in the recursive function.

Recursively Checking Element Equality

Finally, implement equals as shown in the following. It's marked as an export because it's what we'll make available to matchers. (And it's also what you'll write unit tests against in Exercise 2.)

Listing 15-3. concise-test/src/equals.mjs

```
export const equals = (l, r) => {
  if (l === r) return true;

  if (!l) return false;
  if (!r) return false;

  if (
    arraysOfEqualLength(l, r) &&
    l.every((lv, i) => equals(lv, r[i]))
  )
    return true;
```

```
  if (
    objectsWithSameKeys(l, r) &&
    Object.keys(l).every((lk) => equals(l[lk], r[lk]))
  )
    return true;

  return false;
};
```

One interesting thing to note is the *falsy* value check happens *after* the value equality (===) check. This is so that two equal falsy values will still return true:

```
equals(null, null)   // should be true
equals(null, false)  // should be false
```

That completes the basic form of the `equals` function. However, we need to extend it to support our constraint function.

Implementing the contains Constraint

As a reminder, we're trying to implement a function `anObjectContaining` that can be used with `equals` like this:

```
{
  order: anObjectContaining({
    items: [
      { id: 345, unitPrice: 5.0, quantity: 2 },
      { id: 456, unitPrice: 46.99, quantity: 1 },
    ],
  });
}
```

It's certainly not obvious, at first glance, how this should work.

The important observation is that anObjectContaining is *not* recursive: it applies only to the top-level set of keys. This suggests an approach: we can replace the equals(l, r) function for this object with a new contains(l, r) function that returns true if (and only if) r is a *subset* of l, making use of the equals function to check the deep equality of keyed items.

Add the definition of contains to the src/equals.mjs file.

Listing 15-4. concise-test/src/equals.mjs

```
const contains = (l, r) => {
  if (!l) return false;
  if (!r) return false;

  if (typeof l !== "object") return false;
  if (typeof r !== "object") return false;

  return Object.keys(r).every((rk) =>
    equals(l[rk], r[rk])
  );
};
```

The next observation is that our equals function doesn't take any notice of *function* values, but if we let it *execute* any function values, then that gives us a mechanism for introducing the contains function.

This is in fact exactly what anObjectContaining does, which you can now write in src/equals.mjs, as shown. Note this is also an exported constant:

```
export const anObjectContaining =
  (expected) => (actual) =>
    contains(actual, expected);
```

Then, update equals to look for values of r that have type function and execute them, if found.

Listing 15-5. concise-test/src/equals.mjs

```
export const equals = (l, r) => {
  if (l === r) return true;

  if (!l) return false;
  if (!r) return false;

  // The following conditional enables use of
  // "constraining" functions like anObjectMatching.
  if (typeof r === "function") {
    return r(l);
  }

  ...
};
```

Finally, update src/testContext.mjs to re-export the equals and anObjectContaining functions:

```
export * from "./equals.mjs";
```

We will stop our implementation here. It suffices for our needs, but it is not truly complete: we don't want just *any* function to match, but only *constraint* functions. Check out Exercise 3 if you're interested in building a more general solution.

Exercises

1. Use the equals function to build the toEqual matcher.

2. Write unit tests for the equals function.

3. Extend your solution so that only functions that are explicitly *constraint* functions are invoked by `equals`. Since constraint functions should be extensible by any project using `concise-test`, you will either need to determine a way to label these functions at definition or maintain a registry of constraint functions, rather than having a hard-coded list of functions within `equals` itself.

Discussion Questions

1. The second version of our `equals` function, which added support for the `anObjectMatching` constraint, has a very obtuse piece of code:

```
if (typeof r === "function") {
  return r(l);
}
```

Look at the full definition of the `equals` function again, and notice how *out of place* this code snippet appears. Do you agree that if you were a newcomer to this codebase, this piece of code would be difficult to comprehend? What options do you have to improve the maintainability of this snippet? Do you think the unit tests from Exercise 2 are enough? What are the other options available to you?

2. Let's assume your preferred solution to Discussion Question 1 was "add an explanatory comment above the code." What would that comment say? What problems can you see with this approach? Would it be possible to rewrite the comment as a unit test, and if so, what is the benefit of doing that?

Summary

This chapter has covered deep equality and its role in making maintainable test suites. You saw how to implement the `equals` function and also a typical constraint: the `anObjectMatching` function.

In the next chapter, you'll make use of this function as you build the `toBeCalledWith` matcher.

Test Doubles

A *test double* is a special type of function (or object) that replaces a dependency of the structure under test with a "blank" version, which avoids doing any of the real work of the dependency.

In this chapter we'll look at the common scenario of using test doubles to replace the `global.fetch` function that makes HTTP requests.

You'll learn

- The different types of test double

- How to implement a spy function and its matcher

By the end of the chapter, you'll be confident with the use of test doubles.

How Are Test Doubles Useful?

Usually these dependencies are doing *external* things that don't make sense to do in unit tests, like making network requests.

A key characteristic of *unit* test suites is that they avoid doing *external* things because they may be out of our control and therefore cause unexpected failures and also because anything out of process will cause a vast slowdown of test runs.

The other reason you'd use test doubles is to isolate complex test suites from one another. If an object *A* depends on a function *B* but *B* itself is complicated, then you can write one test suite for *B* and then one for *A* with a test double replacing B.

This has two advantages for *A*'s test suite: first, it can make the *Arrange* phase of the test suite simpler, because you can prime the test double very easily without having to interact with *B* in the usual way, and second, it means that any failures in *A*'s test suite *must* be due to errors in object *A*, not object *B*. If they were in *B*, you'd be getting failures in *B*'s test suite instead.

You can see how this can help you write more maintainable tests.

The big downside is that this seam you've introduced can mean you're no longer testing the interaction between A and B. If you're interested in how to solve this problem, see Discussion Question 2.

Spies and Stubs

The most basic form of a test double is a *stub*, which is a hand-rolled function that does nothing other than return a hard-coded value:

```
const dummyResponse = ... // the dummy value
global.fetch = () => dummyResponse
```

The simplicity of this approach means it's often forgotten, but it should always be the first port of call when writing any test involving test doubles.

But what if you wanted to check the *arguments* passed to the function? Now you need a *spy*: which is just like a stub except it makes a record of arguments it's invoked with. Figure 16-1 shows the relationship between the spy and the subject under test.
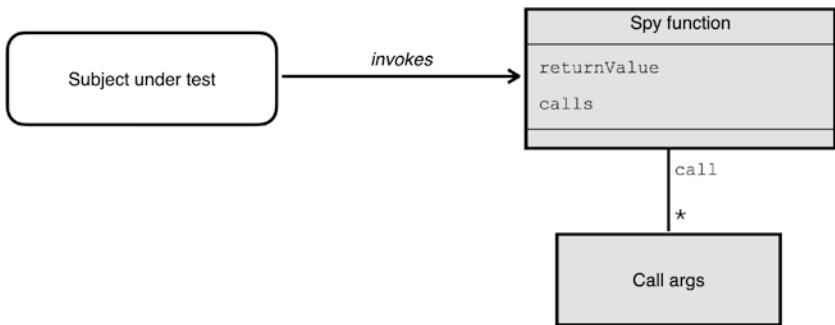
Figure 16-1. *A spy function stores each of its invocation and returns a stubbed value*

You can hand-roll your own spy objects, but most often you'll find that unit test frameworks include their own, together with matchers that instrument them.

The spy function that we'll build in this chapter is used like this:

```
global.fetch = spy(() => ({ ok: true }));
```

The matcher is used like this:

```
expect(global.fetch).toBeCalledWith(
  "/todos",
  anObjectContaining({
    body: { todo },
  })
);
```

Notice the use of anObjectContaining that was the subject of Chapter 15. It enables us to focus on just a small part of a value in a single test, which is really important when making API requests with global. fetch, which tend to give and receive large clumps of data.

I usually split my tests by *things that are likely to change together*. So, for example, the request configuration (headers, HTTP method, etc.) is unlikely to change when the contents of the request body change. And the configuration is unlikely to *ever* change, or very rarely, so it's nice to siphon that off into its own test.

With the theory behind us, it's time to get to work on the new code.

Adding the New Tests

The `todo-example` repository contains a file, `src/api.mjs`, that so far isn't used and isn't tested. Add a new test suite for that now, in the file `test/api.tests.mjs`, with the content shown in the following.

Listing 16-1. todo-example/test/api.tests.mjs

```
import {
  describe,
  beforeEach,
  it,
  expect,
  spy,
  anObjectContaining,
} from "concise-test";

import { saveTodo } from "../src/api.mjs";

describe("api", () => {
  const todo = "todo text";

  beforeEach(() => {
    global.fetch = spy(() => ({ ok: true }));
  });
```

```
it("calls POST /todos with headers", () => {
  saveTodo(todo);
  expect(global.fetch).toBeCalledWith(
    "/todos",
    anObjectContaining({
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
    })
  );
});

it("calls POST /todos with the todo body", () => {
  saveTodo(todo);
  expect(global.fetch).toBeCalledWith(
    "/todos",
    anObjectContaining({
      body: { todo },
    })
  );
});
});
```

Note how it has two tests that interact with global.fetch, which has been set up to be a test double in the beforeEach block.

For these tests to work, we'll need to head back to the concise-test project and implement the spy function and the toBeCalledWith matcher.

Implementing the spy Function

Back in the `concise-test` project, add a new file named `src/spy.mjs` with the following content.

Listing 16-2. concise-test/src/spy.mjs

```
export const spy = (returnValue) => {
  let callHistory = [];
  const spyFunction = (...args) => {
    callHistory.push(args);
    return returnValue;
  };
  spyFunction.calls = callHistory;
  return spyFunction;
};
```

This is an interesting solution to the design shown in Figure 16-1. The function itself maintains its own call registry, by first defining the `spyFunction` variable, which is a function, and then setting a `calls` property on it. (Yes, JavaScript functions are also objects with their own set of properties.)

It's the `calls` registry that can then be used by the `toBeCalledWith` matcher.

Hidden in the middle of this function is the `returnValue` being returned by the `spyFunction` whenever it's called. This is the "stub" part of the spy function.

Implementing the toBeCalledWith Matcher

Now it's time for the matcher that will read from the `calls` array.

In the same file, first add these two imports:

```
import { ExpectationError } from "./ExpectationError.mjs";
import { equals } from "./equals.mjs";
```

Then add the following definition for `toBeCalledWith`, which simply checks for *any* call to the spy with the given arguments.

Listing 16-3. concise-test/src/spy.mjs

```
export const toBeCalledWith = (spy, ...expectedArgs) => {
  const anyMatch = spy.calls.some((callArgs) =>
    equals(callArgs, expectedArgs)
  );

  if (!anyMatch)
    throw new ExpectationError(
      "spy to be called with arguments",
      {
        actual: spy.calls,
        expected: expectedArgs,
        source: null,
      }
    );
};
```

This implementation isn't complete; in fact, the `ExpectationError` class seems like a poor fit because its failure output doesn't quite work. Here's an example of what a failure would look like:

```
api → calls POST /todos with headers
Expected spy to be called with arguments
```

in test/api.tests.mjs:

```
19 |    it("calls POST /todos with headers", () => {
20 |      saveTodo(todo);
21 |      expect(global.fetch).toBeCalledWith(
   |                           ^
22 |        "/todos",
23 |        anObjectContaining({
```

See Exercise 1 if you're interested in improving on this.

To make this available to your test framework, add the following line to src/matchers.mjs:

```
export { toBeCalledWith } from "./spy.mjs";
```

Finally, do the same for the spy function, in src/testContext.mjs:

```
export { spy } from "./spy.mjs";
```

That's everything complete; if you now run npm test test/api. tests.mjs from your todo-example project, you should see the two tests passing. Try playing around with the source to ensure the tests are actually testing what they say they are.

Exercises

1. Expand the toBeCalledWith matcher failure message to output the arguments of all the non-matching calls so that the user can easily determine if the spy was ever called and, if it was, which arguments didn't match their expected arguments.

2. Many built-in functions, like `global.fetch`, should be returned to their original state after a test has finished running. Write a pair of functions `registerMock` and `unregisterAllMocks` that can be used to ensure this happens.

3. Implement a new constraint function, `anything`, that returns `true` for any value other than the `undefined`.

4. Implement a new system for allowing modules, like the `spy` module, to register their own matchers at runtime.

Discussion Questions

1. We implemented a single matcher for our spy function, the `toBeCalledWith` matcher. What other matchers do you think would be useful with spy functions given the data that's saved?

2. A common test error is when a dependent function's signature changes but the spy is forgotten about. (For example, imagine we changed the `global.fetch` function to only take a single object parameter.) In this case, the tests will continue to pass, but at runtime everything will blow up. Discuss some solutions as to how your test runner could be modified to ensure that all spies stay aligned with the functions they are replacing.

Summary

This chapter has introduced test doubles and, in particular, spy functions. This is a highly useful tool for your testing toolbox.

The final chapter of the book, Chapter 17, extends on this by introducing *component mocks*.

CHAPTER 17

Module Mocks

The last chapter discussed why test spies are useful, and it showed one way that we can *inject* the spy into our application code–by simply overriding the value:

```
global.fetch = spy(...);
```

It turns out that spies are pretty simple to use when they are replacing global functions or when we can pass in the spy as a constructor argument.

But one occasion that it's difficult to do this is when dealing with named exports that the application code imports and loads. This is often the case with component frameworks like React.

In this chapter, you will

- Implement a *module mock* registry.

- Implement a Node module *loader*.

- Create a new bin script for invoking Node with the new custom loader.

By the end of the chapter, you'll have a complete understanding of how module mocks work in the JavaScript runtime environment.

It should be noted that the Node API for adding loaders is still unstable and is likely to change. The code and approach detailed here is valid as of Node version 19.3.0.

How Are Module Mocks Useful?

Imagine we updated our `TodoRepository` class to call the `saveTodo` function from Chapter 16. In fact, you can make this change now in `src/TodoRepository.mjs`, by adding `saveTodo` to the `add` function.

Listing 17-1. todo-example/src/TodoRepository.mjs

```
import { saveTodo } from "./api.mjs";

export class TodoRepository {
  add(todo) {
    ...
    saveTodo(todo);
  }
}
```

But `saveTodo` makes a network request to our API, which isn't available within our unit tests. We need to instruct our test suite to mock their entire `./api.mjs` import and return our stub version of `saveTodo` instead.

The same technique can be used with any component framework too. Often you want to do that when a child component causes side effects on mount, like requesting data from an API. Figure 17-1 shows how a test double can be used by a test suite to override a child component dependency of a component under test.
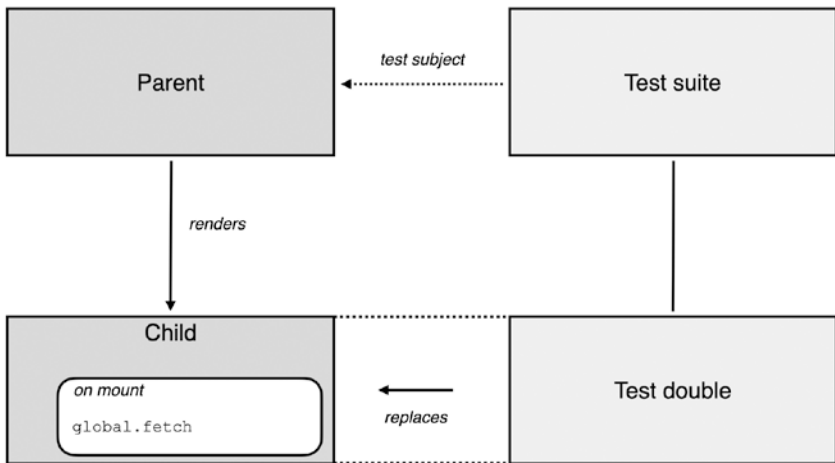
Figure 17-1. *A test suite testing a parent component can stub out a child component if the child has unwanted external effects, like making an HTTP request to fetch data*

With our new application code in place, it's time to update our test suite with our desired module mock function.

The Module Mock API

The new function we'll be using is called `registerMock,` and it takes two arguments: the path of the module to mock and the set of named exports that should be provided to any application code that imports that module.

In the `test/TodoRepository.tests.mjs` file, add the two new imports.

Listing 17-2. todo-example/test/TodoRepository.tests.mjs

```
import {
  ...,
  registerMock,
  spy,
} from "concise-test";
```

Now comes the tricky part. The `registerMock` function needs to be called *before* importing the application code that ends up importing our mocked module, which in our case is the import for `../src/todoRepository.mjs`.

In addition to that, and this bit is crucially important, the `import` statement needs to be changed to a "dynamic" `await import` function call. This is because all "non-dynamic" `import` statements are parsed and loaded before any code is executed from within the file. If we want the `registerMock` call to take effect, then we need to make sure every application code import is of the dynamic form.

Here's the updated import code.

Listing 17-3. todo-example/test/TodoRepository.tests.mjs

```
// see important note below about this next line
registerMock("./src/api.mjs", { saveTodo: spy() });

const { TodoRepository } = await import(
  "../src/todoRepository.mjs"
);
const { saveTodo } = await import("../src/api.mjs");
```

In this code snippet, there's a minor design issue with the first argument passed to the `registerMock` function. Although the solution shown works, most test runners with module mock support will expect the path to be relative to the file you're writing. In this case, you'd expect to write `../src/api.mjs` and *not* `./src/api.mjs`. The advantage of this approach is that it matches the `import` statement.

If you're interested in fixing this problem, see Exercise 1.

Finally, we can add a new test to check that `saveTodo` is called.

Listing 17-4. todo-example/test/TodoRepository.tests.mjs

```
it("calls saveTodo with the correct arguments", () => {
  repository.add(newTodo);
  expect(saveTodo).toBeCalledWith(newTodo);
});
```

That's it for the new test; now let's get on with implementing `registerMock`.

Implementing the Mock Registry

Create a new file in the `concise-test` project, named `src/moduleMocks.mjs`. It has two exports, the `registerMock` function and the `moduleModuleExport` function, which we'll make use of in the next section.

Listing 17-5. concise-test/src/moduleMocks.mjs

```
import url from "url";
import path from "path";

global.mockRegistry = {};

export const registerMock = (
  relativeFilePath,
  mockedModuleExportsFor
) => {
  const absoluteFilePath = path.resolve(
    process.cwd(),
    relativeFilePath
  );
  const fileUrl = url.pathToFileURL(absoluteFilePath);
  global.mockRegistry[fileUrl] = mockedModuleExports;
};
```

```
export const mockedModuleExportsFor = (path) =>
  global.mockRegistry[path];
```

This code makes use of a new global variable named `mockRegistry`. Whenever `registerMock` is called, it simply inserts a new entry into this object, with the file path as the key and the new set of exports as the value.

The `registerMock` function is the one that our tests will call, so export that now in `src/testContext.mjs`:

```
export { registerMock } from "./moduleMocks.mjs";
```

So how do we make an `import` statement read this mock registry rather than returning the real implementation? With a custom loader.

Implementing the Node Loader

Node has a Loader API[1] that we can use for this. It's important to note that this is an experimental API and is likely to change. But at its heart is a function named `load` that is given a URL to an import, which we'll assume is always a file.

Given a `url`, all we have to do is look that up in our registry (using the `mockedModuleExportsFor` function from the last section) and return that set of imports if it's found.

There's a complication, however. In its current form, the `load` function wants a JavaScript *source string* as output: we can't give it a plain ol' object like the one we have in our registry. The reason for this is that the loader is called for *all* types of file, not just JavaScript source files.

Let's write the implementation for the loader first, which uses a `toSourceString` function to do this transformation. We'll look at that afterward.

Add the following to a new file named `src/loader.mjs`.

---

[1] https://nodejs.org/api/esm.html#loaders

Listing 17-6. concise-test/src/loader.mjs

```
import { mockedModuleExportsFor } from "./moduleMocks.mjs";

export async function load(url, context, nextLoad) {
  const mockedExports = mockedModuleExportsFor(url);
  if (mockedExports) {
    return {
      format: "module",
      shortCircuit: true,
      source: toSourceString(
        url,
        Object.keys(mockedExports)
      ),
    };
  }
  return nextLoad(url);
}
```

Okay, so what about `toSourceString`? Since each of our exports exists in a global variable named `mockRegistry`, a simple approach is to take each key in our module mock and write out an `export` statement string for it that points into this global variable. So, for example, we might end up with

```
export const saveTodo =
  global.mockRegistry["<path to api.mjs>"].saveTodo;
```

As a simplification, we can pull out the path to the current mock as an internal variable:

```
const mockedExports =
  global.mockRegistry["<path to api.mjs>"];
export const saveTodo = mockedExports.saveTodo;
```

Let's implement that now. Still in `src/loader.mjs`, add the following import statement at the top of the file:

```
import { EOL } from "os";
```

Then, add the following definitions above the `load` definition.

Listing 17-7. concise-test/src/loader.mjs

```
const toConstString = (name, value) =>
  `const ${name} = ${value};`;

const toExportedConstString = (name, value) =>
  `export const ${name} = ${value};`;

const toSourceString = (path, exportedKeys) =>
  [
    toConstString(
      "mockedExports",
      `global.mockRegistry["${path}"]`
    ),
    ...exportedKeys.map((exportedKey) =>
      toExportedConstString(
        exportedKey,
        `mockedExports.${exportedKey}`
      )
    ),
  ].join(EOL);
```

That's the loader complete. Next, it's time to ensure the loader is called when Node is started.

Adding the New Bin Script

Remember our `src/cli.mjs` bin script that we call when the `npm test` command is run? Well, it's no longer sufficient, because the loader needs to be passed to Node as a command-line argument, using the `--loader` flag.

Create a new file in the `concise-test` project, named `bin/concise-test.sh`, with the following content.

Listing 17-8. concise-test/bin/concise-test.sh

```
#!/usr/bin/env bash

dir="$(dirname "$(realpath "$0")")"
node --loader $dir/../src/loader.mjs $dir/../src/cli.mjs $@
```

The `dir` variable here is used to get the path of the script file–in other words, the `bin` directory. The `loader.mjs` and `runner.mjs` files are then found in relation to this location.

You will need to run the command `chmod a+x bin/concise-test.sh`.

Update `package.json` to now use this as the bin script:

```
"bin": "./bin/concise-test.sh",
```

The implementation is now *almost* complete. You can test it by running the command `npm test test/TodoRepository.tests.mjs`.

Unfortunately, this solution only works when running a single test suite. The next section looks at why this is and how we can solve it.

Bypassing the Module Cache

If you run `npm test`, you'll see that things don't go quite to plan. It turns out that we have a problem of cache invalidation:

```
TodoRepository → add → calls saveTodo with the correct
arguments
Cannot read properties of undefined (reading 'some')

  in test/TodoRepository.tests.mjs:

  59 |        it("calls saveTodo with the correct
             arguments", () => {
  60 |          repository.add(newTodo);
  61 |          expect(saveTodo).toBeCalledWith(newTodo);
     |                                 ^
  62 |        });
  63 |      });
```

As it turns out, Node will only call `load` *once* per URL. After a particular URL has been loaded, it is cached by the V8 runtime. Any subsequent import will use the cached version. That's why this test suite broke: it expected the *mocked* version of the API, but it got the *real* version because the API test suite ran first and loaded the real version before the cached version could be registered.

What we need is for each test suite to get its own copy of the imports.

One way around this is to use Node's *worker threads*. The idea is that we construct a new `Worker` object for each test script file. Each of these is run in its own thread, which gets its own copy of the V8 runtime and therefore its own cache.

> This solution also allows to parallelize our test suites, which could help reduce the total execution time of a test run if you have multiple cores available.

In our new implementation, we will create a new file named `src/worker.mjs` that acts as the new entrypoint for each of these worker threads. This will then import a single test script, call `runParsedBlocks`, and report its results back to the parent thread.

The most difficult part of this new implementation is the cross-thread communication. Because these threads won't share any memory, they instead rely on passing messages from the child worker back to the parent. Figure 17-2 shows how this interaction occurs.
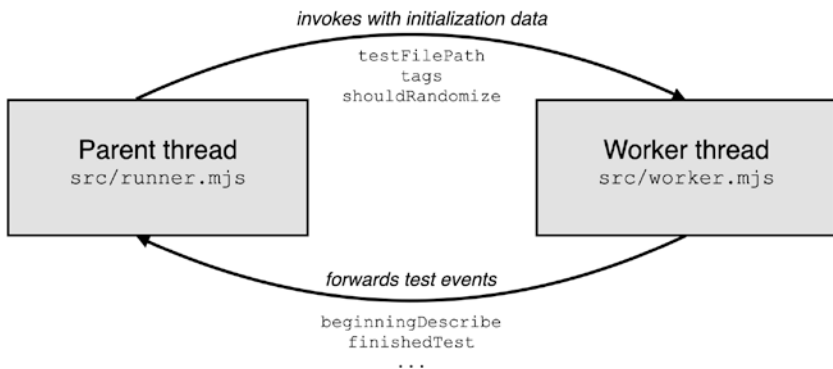
Figure 17-2. *The interaction between the parent thread and the worker thread*

To kick things off, let's write the new test reporter for our worker threads.

Creating a Forwarding Reporter

Add the following file, `src/reporters/workerForwarder.mjs`. It defines a new `forwardEvent` function that acts as a mechanism for taking test events (like `beginningDescribe`, `finishedTest`, etc.) and forwarding them to the parent thread.

Listing 17-9. concise-test/src/reporters/workerForwarder.mjs

```
import { parentPort } from "node:worker_threads";
import { listen } from "../eventDispatcher.mjs";

const forwardEvent = (event) =>
  listen(event, (...args) => {
    parentPort.postMessage([event, args]);
  });

export const installReporter = () => {
  forwardEvent("beginningDescribe");
  forwardEvent("finishedTest");
  forwardEvent("skippingDescribe");
  forwardEvent("skippingTest");
};
```

The `forwardEvent` function works by calling `listen` on all our test events—except for the `finishedTestRun` event that is left for the parent thread.

The `parentPort` object is the special Node object that sends our message to the parent thread. Any time an event is received, it is immediately forwarded to the `parentPort` by calling `parentPort.postMessage`.

When we come to update `src/runner.mjs`, we'll write code to receive these messages in the parent thread and send them on to the existing reporter.

Defining the Worker Thread Entrypoint

Next, let's create the new worker entrypoint, the `src/worker.mjs` file. Its listing is given in the following. Its key job is to import a single test file, which will trigger all of our existing test runner logic for parsing `describe` and `it` function calls. Then, it calls `runParsedBlocks` on that, which invokes all of the test cases and reports on results.

The `workerData` import in this file is the initialization data that will be passed by the parent thread. It contains both the test suite file to be invoked (the `testFilePath` property) and also the command-line arguments that will be needed (the `tags` and `shouldRandomize` properties). These arguments are passed into the `runParsedBlocks` function.

Listing 17-10. concise-test/src/worker.mjs

```
import { workerData } from "node:worker_threads";
import { runParsedBlocks } from "./testContext.mjs";
import { installReporter } from "./reporters/
workerForwarder.mjs";

const { testFilePath, tags, shouldRandomize } =
  workerData;

const exitCodes = {
  ok: 0,
  failures: 1,
  parseError: 3,
};

installReporter();

try {
  await import(testFilePath);
} catch (e) {
  console.error(e.message);
```

```
  console.error(e.stack);
  process.exit(exitCodes.parseError);
}

const failed = await runParsedBlocks({
  tags,
  shouldRandomize,
});

process.exit(failed ? exitCodes.failures : exitCodes.ok);
```

You will notice that the worker thread uses `process.exit` to report its
success or failure result. We'll make use of this in the parent thread.

Updating the Test Runner to Use Worker Threads

Let's continue by updating our `src/runner.mjs` file to invoke the new
Worker objects.

The first step is to create an absolute path to the new `src/worker.mjs`
file. Node requires the absolute path–relative paths won't do. We'll save
this in a constant named `workerFilePath` that will be used later on.

Add the following code to the top of the `src/runner.mjs` file.

Listing 17-11. concise-test/src/runner.mjs

```
import { fileURLToPath } from "url";
import { Worker } from "node:worker_threads";

const __dirname = path.dirname(
  fileURLToPath(import.meta.url)
);
const workerFilePath = path.join(
  __dirname,
  "./worker.mjs"
);
```

254

Just below that, create a new function, createWorker, which takes a testFilePath and processOptions and starts a new worker thread, entering at the workerFilePath and passing the arguments as the workerData value.

Listing 17-12. concise-test/src/runner.mjs

```
const createWorker = (testFilePath, processOptions) => {
  const worker = new Worker(workerFilePath, {
    workerData: {
      ...processOptions,
      testFilePath,
    },
  });
  worker.on("message", ([event, args]) =>
    dispatch(event, ...args)
  );
  return worker;
};
```

You can see in the preceding snippet how any message received from the worker thread is immediately sent on to the dispatch function, ready for reporting.

Next, we need a waitForWorker function, which converts the exit message from the worker thread to a Promise object. We need this so that we can wait for each worker thread to complete before we print summary information at the end of a test run.

Listing 17-13. concise-test/src/runner.mjs

```
const waitForWorker = (worker) =>
  new Promise((resolve) => {
    worker.on("exit", (code) => {
```

```
    resolve(code);
  });
});
```

The waitForWorker function will receive the worker thread's exit code. Next, we need a function anyFailures that takes these exit codes and returns true or false depending on if any of the worker threads had a non-zero exit code:

```
const anyFailures = (exitCodes) =>
  exitCodes.filter((exitCode) => exitCode !== 0).count >
  0;
```

Now it's time to use these three functions. Replace the existing run function definition with this version.

Listing 17-14. concise-test/src/runner.mjs

```
export const run = async () => {
  installReporter();
  const testFilePaths = await chooseTestFiles();
  const processOptions = {
    tags: readTags(),
    shouldRandomize: readRandomFlag(),
  };
  const exitCodes = await Promise.all(
    testFilePaths.map((testFilePath) =>
      waitForWorker(
        createWorker(testFilePath, processOptions)
      )
    )
  );
```

```
  dispatch("finishedTestRun");
  process.exit(
    anyFailures(exitCodes)
      ? exitCodes.failures
      : exitCodes.ok
  );
};
```

Almost done. The final test is to delete the `import` statement for `runParsedBlocks`, because it's no longer used in this file.

If you run `npm test` now, you will not see any tests reported–and no errors too. There's one final issue we need to address: serialization of the test objects we send from the worker threads back to the parent.

Changing Reported Test Objects to Be Serializable

Our existing test reporter needs to send test information across when certain events occur: the `describeStack` object (used to determine indenting levels), `describe` object, and `test` object. When we built the test reporter, we did the simplest thing and just sent across the exact same objects that the runner used to execute tests.

But there's a problem: these objects contain *function* values. These values can't be serialized. Thankfully, they aren't needed for reporting purposes, so we can remove them from the objects before dispatching them.

In the file `src/testContext.mjs`, start by adding these three new formatting functions.

Listing 17-15. concise-test/src/testContext.mjs

```
const formatDescribeStack = (describeStack) =>
  describeStack.map((describe) => describe.name);

const formatDescribe = ({ name, sharedContextFn }) => ({
  name,
  sharedContextFn: sharedContextFn !== undefined,
});

const formatTest = ({ name, describeStack, errors }) => ({
  name,
  errors,
  describeStack: formatDescribeStack(describeStack),
});
```

Then update each of the four `dispatch` calls in the `src/testContext.`
`mjs` file, as shown in the following.

Listing 17-16. concise-test/src/testContext.mjs

```
dispatch(
  "skippingDescribe",
  formatDescribeStack(describeStack),
  formatDescribe(describe)
);

dispatch(
  "beginningDescribe",
  formatDescribeStack(describeStack),
  formatDescribe(describe)
);

dispatch("skippingTest", formatTest(test));

dispatch("finishedTest", formatTest(currentTest));
```

Now you can run tests with `npm test` and they should pass. The test runner has created a worker thread for each test file, each with its own module cache and each with its own set of module mocks.

Exercises

1. Update `registerMock` so that it takes the file string relative to the test file. It will need to take this path and convert it to be an absolute file path. To do that, you'll need to get access to the call frame. See Chapter 6 for ideas on how to do that.

2. With the solution to Exercise 1 in place, update `registerMock` so that it throws an exception if the file path is not valid.

3. The worker threads implementation introduces the possibility that multiple test suites will run concurrently, meaning that their reported test output will appear jumbled up. Fix this problem, so that regardless of any concurrency involved in running specific test suites, the test runs are still reported in the correct order.

Discussion Question

1. Our solution for formatting test objects so that they can be serialized feels a little… *hacky*. Compare this approach with doing the "formatting" inside the `forwardEvent` function itself and leaving the original reporter code alone. What other approaches could

be used to solve the problem? For example, what
if the reporter kept track of its *own* describe stack,
rather than needing the `describeStack` argument?

Summary

We have ended the book with an implementation of the most complicated
feature of a JavaScript test framework: module mocks. You have seen
not just how they can be implemented but also how they can be used
effectively to test an application codebase.

Index

A

Act phase, 74, 79, 91

addDuringExecutionModifier, 159

add function, 32, 54, 242

addModifier function, 158, 194, 195

afterEach, definition, 75

afterEach function, 66–67, 78, 135

anObjectContaining function, 222, 226–228, 233

anObjectMatching function, 229, 230

ansiColors, 36

anyFailures function, 37, 39, 256

API calls, 188, 221

Application code, 5, 7, 43, 117, 130, 150, 186, 204, 241

Arrange phase, 63, 66, 74, 78, 79, 211, 232

Arrange, Act, Assert (AAA) pattern, 62–65

Array.slice function, 112

arraysWithSameLength function, 224, 225

AssertionError, 89

Asynchronous behavior, 147, 148

Asynchronous requests, 149

Asynchronous tests, 99, 147–159

Async method, 147

Automated test enlightenment
first stage, 5, 6
second stage, 6
third stage, 8–10

B

Babel plugin, 158

Bash-like runtime environment, 5

Battle-hardened implementation, 173

beforeEach blocks
application, 72–74
currentDescribe, 75
definition, 70–72
describe block's, 68, 74
describeStack, 69
order, 67, 68
updateDescribe, 75

beforeEach function
AAA pattern, 62–65
Arrange statements, 66
multiple, 67
setup code, 61

beginningDescribe event, 164, 183

Behavior-Driven Development (BDD), 24

Boolean flag, 37