Acceptance Test Driven Development with React

cucumber

puppeteer

enzyme

redux

react

axios

react-router

jest

json-server

Juntao Qiu

Acceptance Test Driven Development with React

Juntao Qiu

This book is for sale at http://leanpub.com/build-react-app-with-atdd

This version was published on 2020-01-08

Contents

About the author

Juntao Qiu is a senior Web application developer at ThoughtWorks. Over the past ten years, he has been worked on various types of projects, which help his clients to build solid, extensible and high quality Web applications. The type of projects are from traditional Web application with *jQuery + JSP(Java Server Pages)* to *SPA(Single Page Application)*s with Backbone, AngularJS and React. He knows how to handle the complexity of the real world projects by applying appropriate methodology include write clean code and effective automation tests.

He also is a technical author who has already published two books in Web application domain: *JavaScript Core Concepts and Practices(2013)* and *Lightweight Web Application Development(2015)*. And he passions on writing blogs and delivery tech speeches in community events as well. He has a real passion on *Clean Code*, *Refactoring*, *Test Driven Development*.

Additionally, he does *Muay Thai* and *Boxing* in his spare time.

Preface

I was reading a classic (old) book named *The Unix Programming Environment* when I was in college back to 2004. It was published in 1999 if I remember correctly. I was shocked by the last 3 chapters which demostrated me how to implement a high-order-calculator by using `lex` and `yacc`. Not only because the content itself - which is very impressive and informative, but also by the way it was describing. I was totally attracted, after typed all the program into my laptop and saw it compiled, the vast satisfaction intoxicated me.

That part of my memory is so profound that everything time when I want to write some tutorials or blogs, I would automatically try to use the approach used in the book. I prefer concrete and workable demos over platitude theories, it essential if you can start with something small, enhance it piece by piece, and til the end you got something complex enough to solve real world problem. The most important thing is your reader could get the whole idea of how exactly it works and how it's build up. Of course, to make each step as evident as someone who doesn't have the similar background knowledge can understand is a huge challenge for any author.

However, I've been trying to apply that approach in this book. I would like to walk you through a real React application from its very beginning in the lifecycle to somewhere that complex enough (at least for some scenarios). We will learn together about best practices from real-world projects like *unit tests*, *acceptance tests*, * componentization, **refactoring* as we go. We would start with some fundamental and simple skeleton code, then improve it gruadually, piece by piece. I promise that when you look back, you could be surprised about what you have been accomplished.

Some highlight of this book

You will be expected to learn the following topics through the book.

- Build a React application from scratch by applying Acceptance Testing Driven-Development
- Fundamental ES6 with testing (`jest`)
- Puppeteer for end-to-end testing
- Component testing by Enzyme
- State management with redux
- Design your application state shape
- Best practices from battle proven projects
- Mock backend service and make it the way you would like
- Refactor JavaScript code to make the code more concise and beautiful
- Page Object pattern to make test code itself more readable and maintainable
- Write executable document by utilize cucumber.js

A very brief history of Test Driven Development

Test Driven Development (TDD)

The definition of TDD from Wikipedia is:

> Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: Requirements are turned into very specific test cases, then the software is improved to pass the new tests, only. This is opposed to software development that allows software to be added that is not proven to meet requirements.

Actually, the IT industry has a very heated discussion around Test-driven development. People arguing about both the feasibility of applying TDD in a real-world project and the cost to do so. The conflicts are focus on particularly on one of its variant *Unit* Test Driven Development (actually when the concept of TDD was originally invented, it was tied closely to UTDD). The latest and most famous one debate of TDD came from David Heinemeier Hansson, (DHH)(author of Ruby on Rails), Kent Beck and Martin Fowler, you can find more from here[1].

Although debates around TDD have never stopped, and won't stop anytime soon from my point of view, there is one thing developers are now all agreed with: automation test itself is essential in the development process. It's well understood that tests (whether you write it before production code or after) can help developers significantly in various of ways, things like locating defects more quickly and efficiently, reduce the number of potential defects, make it easier to do refactoring while developing new features, and so on. Still, people have many arguments around when to test, the granularity of tests, and how to abstract different levels of test.

Write the test before or after the production code?

The hardest thing of applies TDD in your daily workflow is *when* you write tests before you write any production code. For most of the developers, that's not just ridiculous, but counterintuitive, and breaks their own way of work significantly. Uncle Bob has a very great article[2] talking about test first or test last approaches, if you haven't read it yet, I highly recommend you to.

[1] https://martinfowler.com/articles/is-tdd-dead/
[2] https://blog.cleancoder.com/uncle-bob/2016/11/10/TDD-Doesnt-work.html

The core of TDD is that you should build the fast feedback mechanism first, once you have it, it actually doesn't matter much you write the test first or last. By saying fast feedback I mean you can test a method or an if-else branch very lightweight and effortless. If you add tests after all the functionality have been done, you are not doing TDD by any means. Because you don't have the essential fast feedback loop, then the benefits that TDD promises might be largely absent.

By utilising fast feedback loop, TDD could make sure you always in the right track safely, and give you sufficient confidence to do the further cleanup. And proper clean up could lead a relatively better code design. Of course, the clean up does not come automatically, it needs extra efforts to be put. TDD is a great mechanism to protect you from the break the whole application when you do that.

So, what does TDD looks like anyway?

The workflow of TDD

During the TDD workflow, a developer has to answer the following two questions repeatedly:

- Is the code meet the requirement?
- Is it the simplest way to do that?

Once the code is meeting requirement in the simplest manner, we need to see if we can do the code refactoring. Code refactoring is the essential part when you are applying TDD, we'll see in detail in the next section.

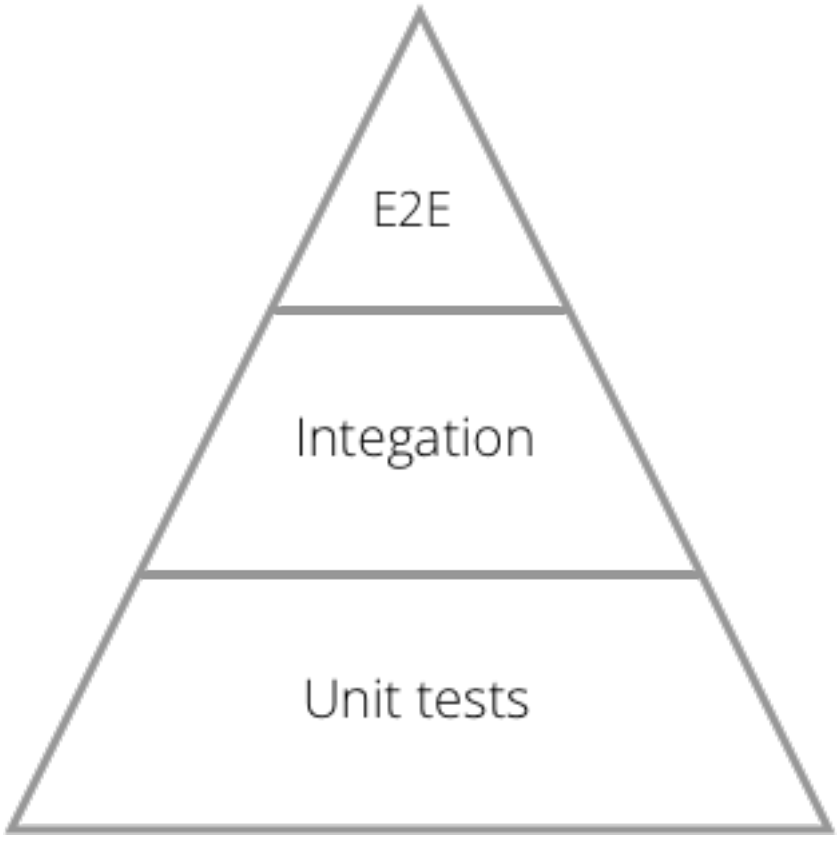
Tests for developer

Test pyramid

Mike Cohn in his famous book **Succeeding with Agile** innovated this model[3]. The essential part of this model is that in a well-designed test suite, tests should contain at least those parts:

- End to end tests
- Integration tests
- Unit tests

[3] https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid



Test Pyramid

Typically, you would have a few end-to-end tests that cover critical paths from the end users perspective. And you have more integration tests in the middle layer, those tests are making sure different components across the whole application can fit together and talk to each other correctly. Then at the bottom, you have many more unit tests that verify each building blocks could function well.

It actually has many ways to explain the pyramid. The higher the tests in the pyramid, the more expensive they are in terms of running cost, and the harder it can guide developers to locate the defects and debug.

The lower it is in the pyramid, the more should the number of tests be; because of each type of test focuses on the different perspective of the software quality, the number of tests, the time-cost, the feedback lifecycle are all different from one to another.

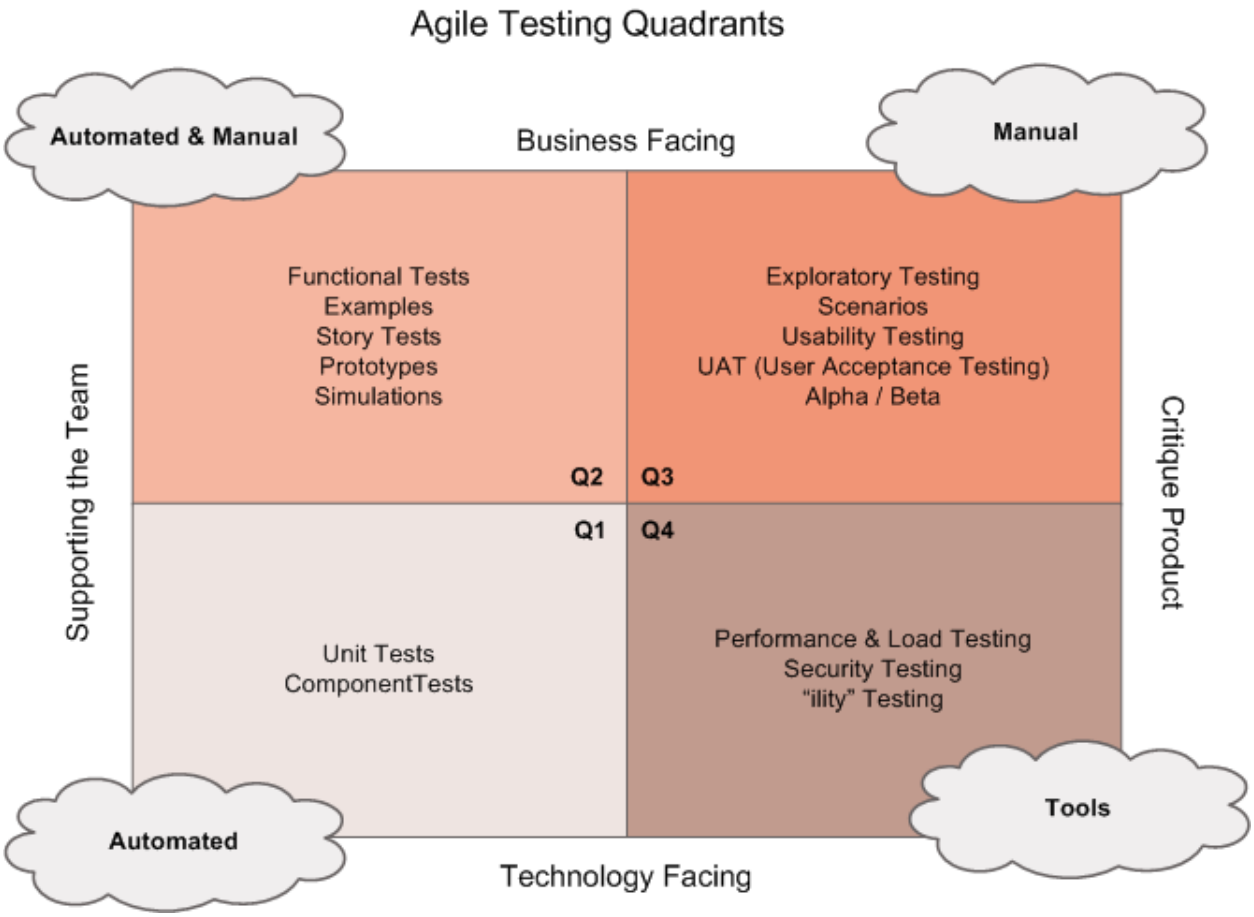
The long-running, fragile test suite does not help to support the development, or maybe even worth, it could deliver a wrong message to the team: `automation test is somehow useless`. And after some time, test suites are then abandoned, and then it could put the whole software system under significant risk.

The test pyramid is an excellent reference for us to design our test strategy. If we build everything from scratch, that's easy. We need to make sure when we need to add some test, we added it after reviewing the shape of the current test suite. In contrast, when we are working on some legacy

system, we need to refactor the whole test suite (if any) to `test pyramid` iteratively, clean up the duplicated, long-running tests from the higher level, and make sure we have enough lower level tests for supporting the development.

Agile testing quadrants

Crispin & Gregory in their book **Agile Testing: A Practical Guide for Testers and Agile Teams** are innovated the concept of Agile testing quadrants, described in the following figure:



Test Quadrants, source: https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/

In this figure, the authors divided tests into 4 quadrants. For the X axis, tests on the left-hand quadrants help the delivery team to understand what should be tested, and tests on the right-hand side help them evaluate the system from the outside perspective. For y axis, tests on the top make sure the code meets the business requirements, while tests on the bottom are related more to internal quality.

Since we're mainly focusing on `test-first` approach to understand business requirement and then driven the production code from a developer's perspective, we will only discuss tests in Q1 and Q2 in this book. In the chart, acceptance tests check if our product is meeting the business requirement,

on the other hand, unit tests and integration tests are focused on technical details. Compare to exploratory tests or performance tests on the right, all those tests are used to support the developers to write correct code.

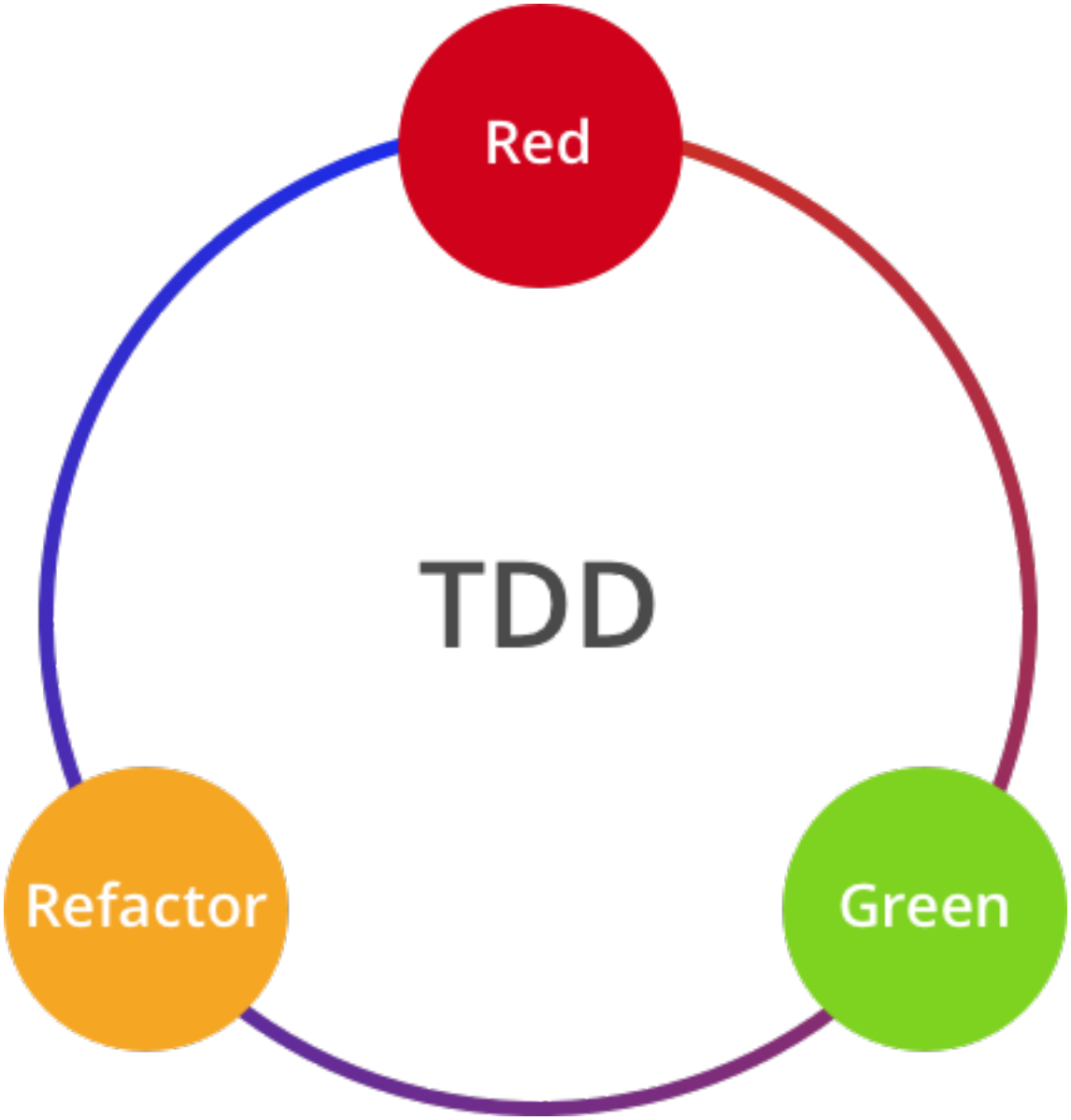
Tests for developer

Automated tests can be categorized into different types, we will only focus on those types that can directly guide developers to deliver valuable software here. `Acceptance tests` can ensure that developers are doing the right thing, while `Unit tests` make sure they are doing things right.

Tests can protect the developer from breaking existing functionality when they are working on legacy codebases (which is most likely what you are doing now). Especially when you have to work collaborate with other co-workers. On one hand, you'll be told which modules are impacted by your changes through the failed tests after a merge, on the other hand, all passing tests would provide you more confident: your changes didn't break any existing components.

Applying TDD in your next project

There is a very famous diagram that explains how to do `TDD` practically, it's named: `Red-Green-Refactor` loop:



Test Driven Development

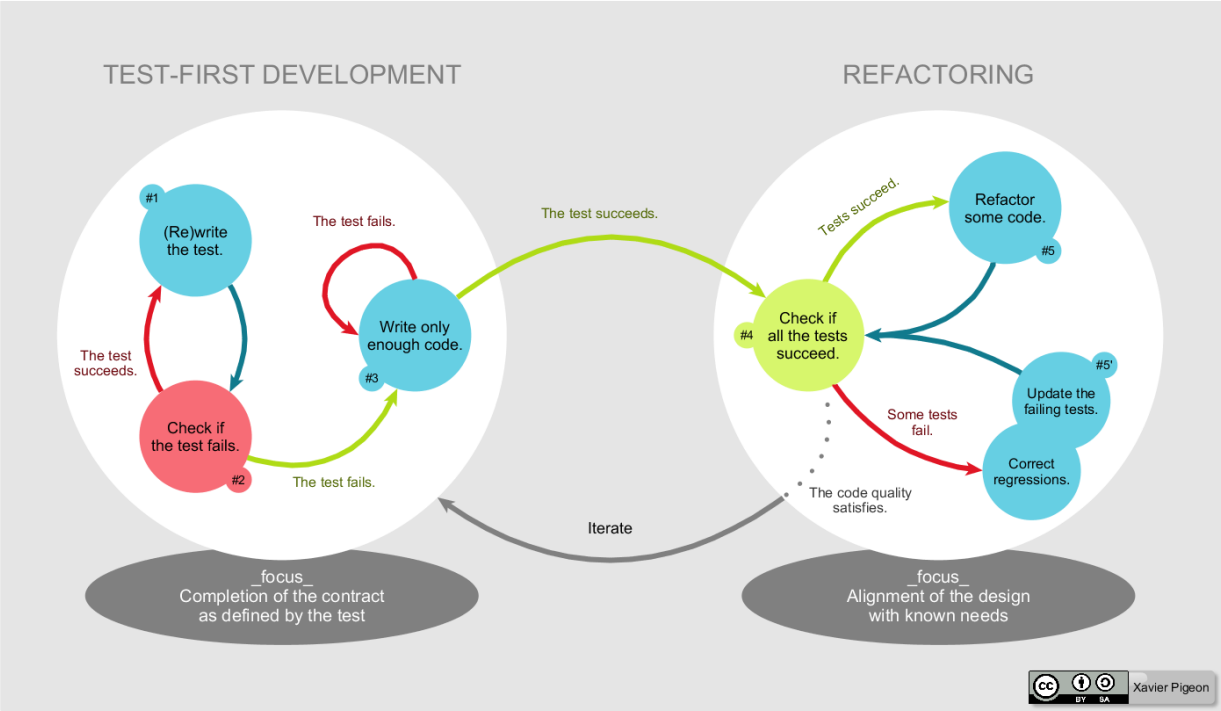
Usually, there is a short description along with this diagram, sometimes called three principles of `TDD`:

- Write a test and see it fails
- Write just *enough code* to make the test pass
- Refactor if some code smells found

At first glance, it pretty easy to follow. The problem of any principles is that they are not working for beginners. Principles are a bit `high-level`, and it's hard applying it as a guide because of the lack of details. For example, by understanding the principles could not help you to answer the following

questions: how can I write my very first test? What does that mean by `enough code`? Also, when/how to do the refactor?

This diagram should be narrowed down like this:



Test Driven Development, source https://en.wikipedia.org/wiki/Test-driven_development

`TDD` actually contains two parts: `quick implement` and `refactor`. In practice, the test for `quick implement` is not limited to the unit test. It could be `acceptance test` as well. Get started from `acceptance test` could be a better idea. From `acceptance tests` it ensures doing the right thing has a higher priority for developers, and it provides enough confidence to developers while they want to clean up and refactor the code better later on. Acceptance tests are from the end user's perspective, a passing acceptance test can make sure the code meets the business requirement. On the other hand, it can protect the developer from wasting their time.

There is a very interesting principle named `YAGNI` in extreme programming, meaning `you aren't gonna need it`. `YAGNI` can be very useful for protecting developers and avoid potential waste of time. Developers are very good at making an assumption on the potential requirement changes, and for the assumption(fake requirement) developer would then do some abstraction or optimisation to make the code more `generic` or `re-useable`. The only problem is that the assumptions rarely turn true. `YAGNI` emphasize that do not do it until you have to.

On the contrary, you can do that in the `refactor` phase. Since you already have enough test coverage, it's much safer to do the clean up like: modify the class name, extract method, or extract some class to a higher level - anything that helps to make the code more `generic` and `solid`.

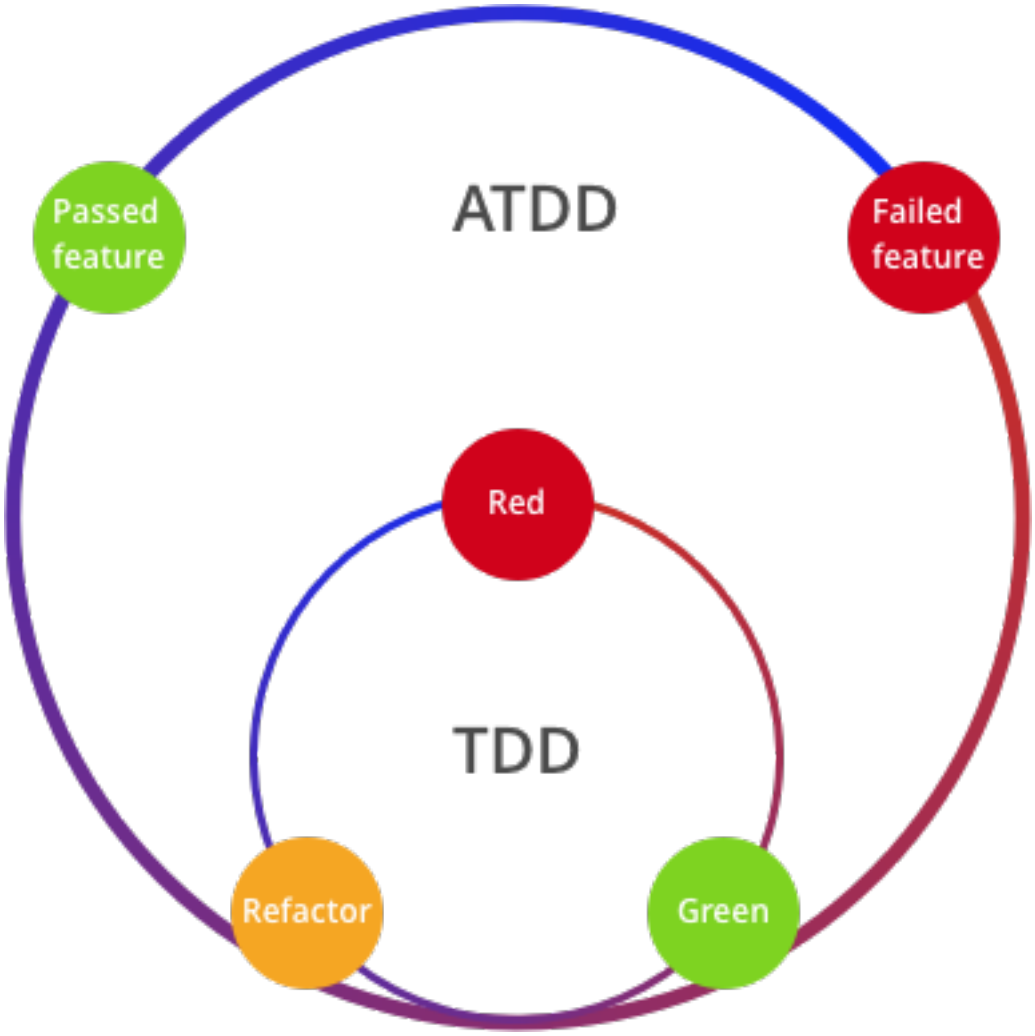
Types of TDD

TDD is a huge concept. It has so many variations like UTDD, BDD, ATDD, and so on. Traditionally TDD implying Unit Test Driven Development. In contrast, the TDD we are discussing in this book is ATDD(Acceptance Test Driven Development), it's an extending version of the traditional one, it emphasizes more on writing the Acceptance test from the business perspective and then driven out the production code.

Having various tests to make sure we are always on the right track, and in the meantime, having the function correctly.

ATDD

ATDD describe the software's behaviour from the end user's point of view. We can merge the ATDD and UTDD into one diagram like this:



ATDDï¼Œource as watermark

Basically, the diagram describes the following steps

1. Write a failed acceptance test
2. Write a failed unit test
3. Write code to make the unit test pass
4. Refactor the code
5. Repeat 2-4, until acceptance test pass

However, when you look at this process closely, you would find that during the development process, the `acceptance test` will fail for quite a long time. The feedback turns to be very long, an always failed test equals to no test(protection) at all. The developers would not know whether there are defects in the implementation, or there is no implementation at all.

To resolve this problem, you have to write the point for `acceptance test` pretty small, like it just

tests a very tiny slice of the requirement. Additionally, you could use the `fake it until you make it` approach as what we are going to use in this book.

The steps remain same, only has a `fake` step:

1. Write a failed acceptance test
2. Make it pass by the most straightforward way
3. Refactor based on code smell(like hard code, magic number, etc.)
4. Add new test based on new requirement(if we need new acceptance test, back to 1, otherwise the process just like traditional TDD)

Note that in the second step, you can use `hard code` or snippet of static `HTML` just make the test pass. That at first glance looks pretty silly, but you will see the power of `fake` in the next few chapters.

The benefit of this variant is when the developer is refactoring, there is always an acceptance test protecting you from breaking the business logic. And the drawback is that if the developer doesn't have enough experience, then it's very difficult to come up with good designs - they would keep the `fake` in the same level (magic number, lack of abstractions, etc.).

Prerequisite of TDD

TDD does have a strict requirement to developers, and yet crucial: how to detect code smell and how to refactor them to good designs. For example, when you find some low qualified code(e.g. lack of abstractions) and don't have the sense to make it better, then `TDD` can help you in anyways. Even though you are forced to using the workflow of `TDD`, you would get some unmaintainable tests and poor code.

I highly recommend you read those books to build the solid foundation for approaching `TDD`, or even you decided not to do `TDD`, those books are still highly recommended.

- Clean code[4]
- Refactoring[5]

A typical workflow of `TDD` has 3 parts:

- A test case description requirement(specification)
- Some code to make the test pass
- Refactoring

A common misunderstanding is that test code is the second tier, or don't have the same importantness as production code. They are as important as production code in anyways, so while doing the refactoring, make sure the changes are made both on the production code and the tests.

[4] https://www.amazon.com/gp/product/0132350882/ref=dbs_a_def_rwt_bibl_vppi_i0

[5] https://www.amazon.com/Refactoring-Improving-Existing-Addison-Wesley-Signature-ebook/dp/B07LCM8RG2/ref=sr_1_1?crid=3VXF4S52SH9LK&keywords=refactoring&qid=1554591603&s=books&sprefix=refac%2Cstripbooks-intl-ship%2C362&sr=1-1

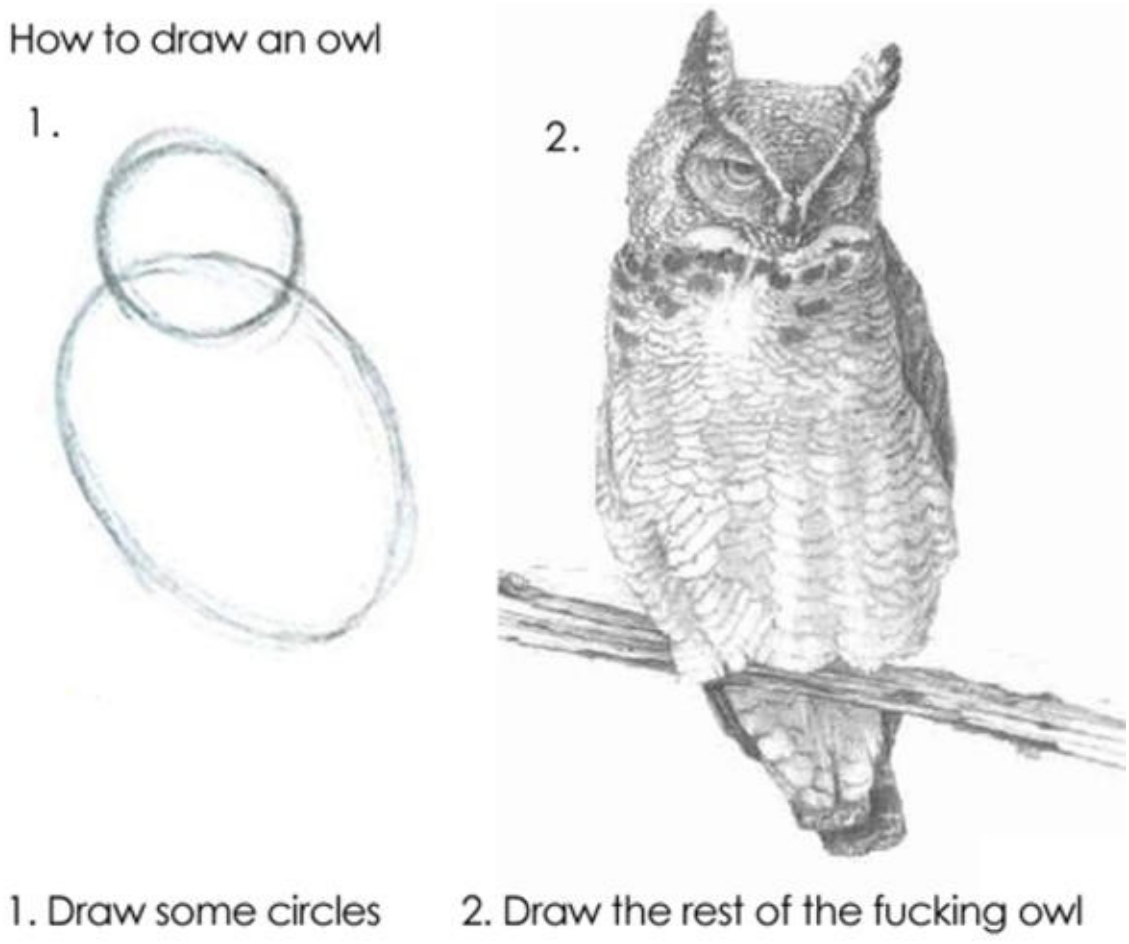
Technology

Usually, the beginner always finds that apply TDD is quite hard because of there is a dilemma around using `TDD`:

- For the simple tasks, they don't need TDD
- For the complicated tasks, set up the TDD mechanism itself is too hard

There are a lot of tutorials and articles out there to describe techniques you how to do `TDD`, and some may even contain how to split tasks. However, for the sake of simplicity, things discussed in those tutorials are significantly simplified and make it very hard to apply to real-world project directly. For example, nowadays many products are based on Web, both the interaction and a considerable partition of the logic are moved to the frontend, to the UI, And the traditional techniques of writing a unit test to driven backend logic is kind of outdated.

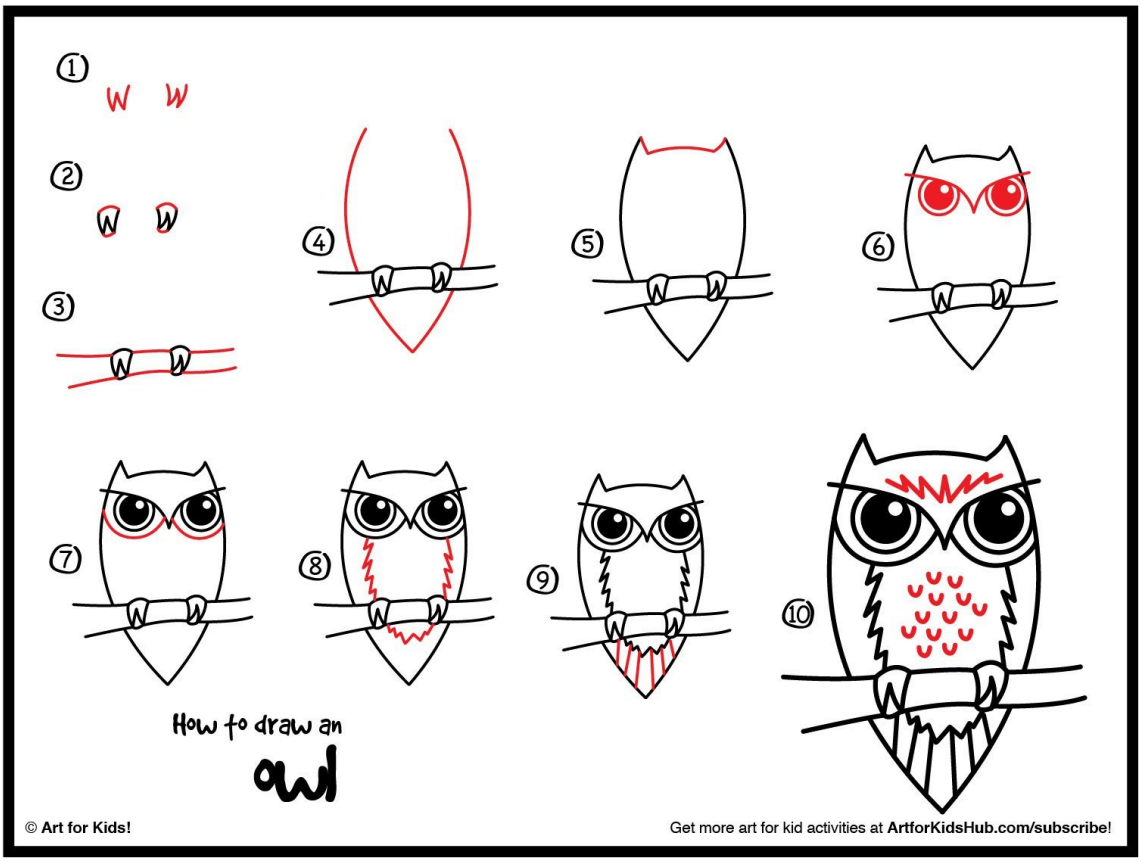
Beginners would find it difficult to use the knowledge they learnt from those articles in their daily work, just like the famous comic drawing `how to draw an owl` shows in the image below:



How to draw an owl, source: https://knowyourmeme.com/photos/572078-how-to-draw-an-owl

We are using `Acceptance Test` Driven Development to develop an end-to-end web application from scratch in this book. You can learn how to use `React` and other libraries as a whole tech stack to build a typical Web application in the real world.

I hope you can apply this method to your project much smoothly, in a step by step manner, as the image shows below.



How to draw an owl, source as watermark

Another critical skill required by applying `TDD` effectively is split the big requirement into smaller chunks - `Tasking`.

Tasking

There is a classic joke: 'How many steps to put an elephant into a fridge'? The answer is `3` steps:

- Open the fridge
- Put the elephant in

- Close it

When we have some task in hand, without using any tools and do it just think it in our brain, then it's easy to be trapped by heaps of technical details, and do not even know where to start. Our brain likes explicit and concrete things, and it hates abstraction, things invisible or implicit.

However, by using some simple tools, we can make the work much more comfortable for our brain to handle. `Tasking` is one of those tools, it can help us divide a big task into smaller ones, and tick them one after another.

For split a relatively big task into smaller pieces, one principle you can use is `INVEST`. It's actually wild used in agile story splitting.

Separation principle - INVEST

The name `INVEST` stands for:

- Independent
- Negotiable
- Valuable
- Estimate
- Small
- Testable/Timebox

Meaning for any given task, you could make it as independent as possible, can be picked up and done in parallel. Negotiable means it should not be fixed as carved on stone, the scope could change based on the trade-off of importantness and cost. Each user story must provide business value, the effort to make it should be measurable, or has an estimation. It should be relatively small, the big piece contains more unknowns and potentially would make the estimation less accurate. Finally, testable makes sure we know how the done looks like by verifying some key checkpoints.

For example, when we want to develop a searching feature for an *eCommerce* system. We can use `INVEST` guide us when we doing the analysis. For example, searching could be split as a few stories, by different dimensions:

- A user can search product by name
- A user can search product by branding
- A user can search product by name and branding

For `User can search product by the name`, we can keep using `INVEST` to split one story into a few tasks from a developer's perspective:

- The search result in memory(ArrayList + Java Stream API)

- Case sensitive supporting
- Wildcard (regular expression) supporting

We can even keep using the same principle to split each item above:

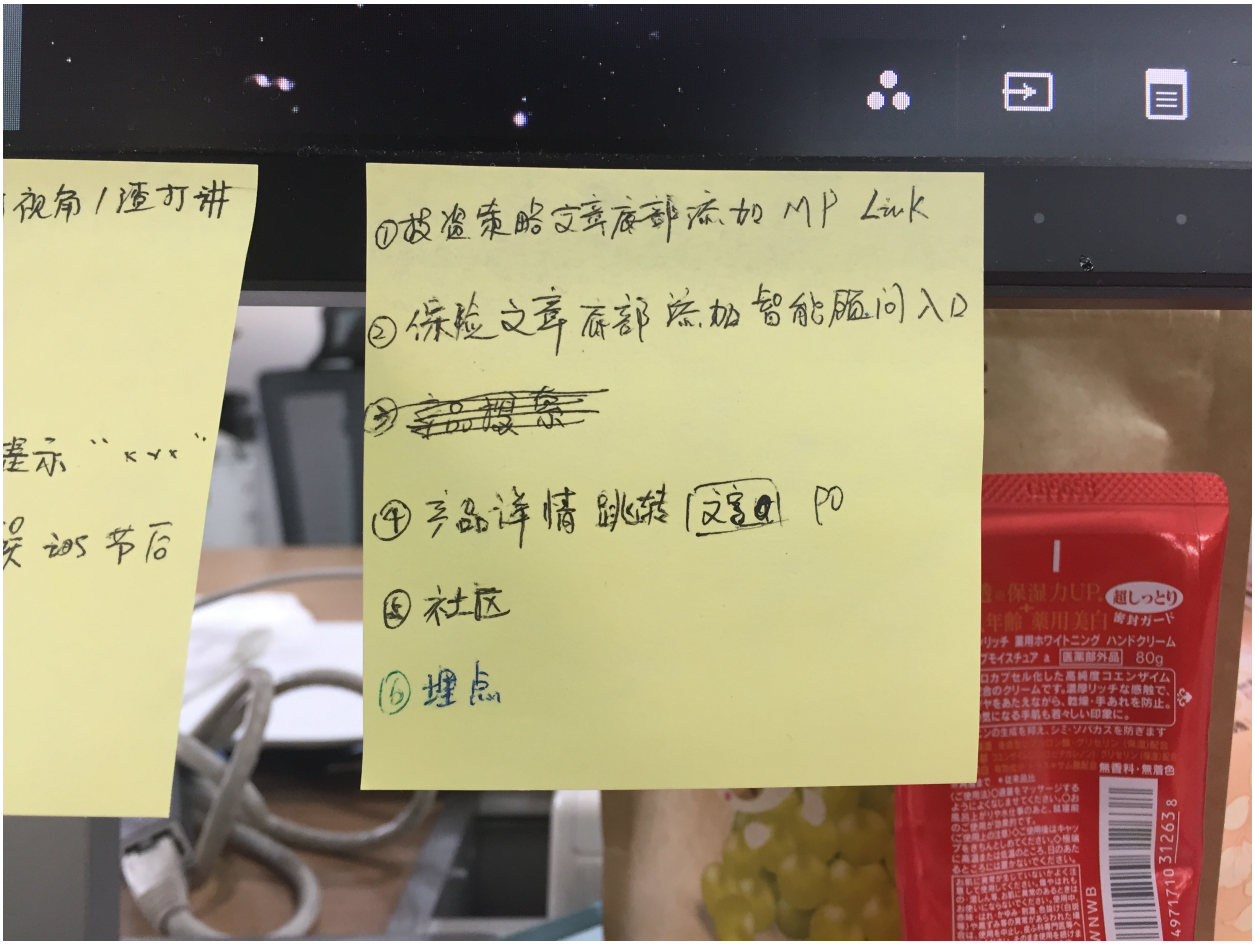
- write an acceptance test
- write code to make the test pass
- refactor
- write a unit test
- write code to make the test pass
- refactoring
- …

That would lead us a well-defined task to hand on, and allow us to verify each step clearly.

Todo in post-it

Usually, we can stop at the second round of splitting, since the `red-green-refactor` is far too detailed in terms of tasking. To make the tasks visible, we can put it down on a `post-it` note, and mark a simple tick once it's done.

By using this simple tool, it allows you to focus on what you're going to do, and make the progress more accurate when you want to update it to other team members (say, in the daily stand-up meeting). By saying 50% done, it's 50% done because you can literally read that on the tasking list you made early.



Tasking

Summary

We walked through the `test pyramid` and `agile testing quadrants`, and introduced the `Acceptance Test Driven Development` as the way we write code in this book. When doing the `ATDD`, we'll keep doing the classic `Red-Green-Refactor` loop.

Refactoring depends on the sense and experience of identifying code smell. As long as you find the smell, you can apply the refactoring technique correspondingly. And then we may archive maintainable, human readable, extendable and clean code along the way.

Additionally, `TDD` always goes with another powerful practice - `tasking` in practice, and you can use `INVEST` principle to help you split big task to smaller pieces. After the proper splitting, you can gradually improve the basic version and finish the big task iteratively.

In the next Chapter, we will introduce a concrete example to demonstrate how to apply `TDD` step by step. Alone with that example, we will also cover the fundamental knowledge needed for adopting

TDD. Such as the usage of jest testing framework, how to do the tasking with real-world examples and other techniques.

Get started with Jest

In this chapter, we're going to learn some concepts and features about `jest`, such as different types of `matchers`, the powerful and flexible`expect`, extremely useful `mock` when you are doing unit testing and so on. Additionally, we will learn how to arrange our test suite in an easy-to-maintain manner, and leverage the best practices from the real world projects.

Firstly, you would see how to set up the fundamental environment to write your first test. We use `yarn` as the package management system, and use `ES6` as the primary programming language throughout this book.

So, without further ado, let's get started.

Setup the environment

Node.js and yarn

We're going to use `node.js` as the platform in this book for almost all scenarios. I presume you have already had `node` installed on your computer. You can simply run the following command to install it if you haven't yet in `Mac` with `homebrew`:

```
1 brew install node
```

Once you have it installed locally, you can then use `npm` (Node Package Manager) to install node packages - it's a built-in binary shipped with the `node` installation. However, there is a better alternative for package management named `yarn` which runs faster than `npm` in most cases.

`yarn` is yet another package manager just like `maven` if you're familiar with `Java` ecosystem, or `pip` if you are from `python` world.

To install `yarn`, simply type:

```
1 npm install yarn
```

Jest

`Jest` from `Facebook` is a testing framework allows a developer to write reliable and fast running tests in a quite readable syntax, it can watch changes in files and re-run the necessary tests automatically, which allow you get instant feedback, and that is a crucial factor and can determine whether `TDD` works or not for you. Primarily, the faster tests can run, the more effectivity the developers will be.

Let's install it as the first step:

```
yarn add --dev jest
```

After the installation, you can run `jest --init` to specify some default settings, such as whether will `jest` find the tests file and the source code, which environment(there are a lot) should the `jest` run against (browser or node for the backend). You have to answer some questions to let `jest` understand your requirements, let's just for now accept all the default settings by just say `Yes`.

Note that if you have installed `jest` globally (with `yarn global add jest`), you can use the following command to `init` the config directly.

```
jest --init
```

Otherwise, you have to use the local installation located in `node_modules/.bin/` like this:

```
node_modules/.bin/jest --init
```

First workable test with jest

Cool, we're ready to write some tests to verify if all parts are working together now. Let's create a folder named `src` and put two files in `calc.test.js` and `calc.js`.

The file ends with `*.test.js` means it's a pattern that tells `jest` treat those files as `tests`, as defined in `jest.config.js` we generated previously:

```
  //The glob patterns Jest uses to detect test files
  testMatch: [
    "**/__tests__/**/*.js?(x)",
    "**/?(*.)+(spec|test).js?(x)"
  ],
```

Now, let's put some code in the `calc.test.js` first:

```
var add = require('./calc.js')

describe('calculator', function() {
   it('add two numbers', function() {
       expect(add(1, 2)).toEqual(3)
   })
})
```

Here is something new if you never tried to write a test in `jasmine` (a very popular testing framework before `jest`), like `describe` and `it`. `describe` is a function that can be used to describe a test suite, and you can define test cases (by using `it` function) in it. It's a proper way to put a human-readable text as the first parameter, and the executable callback function as the second one. Moreover, for `it` on the other hand, you could write the actual testing code.

The actual assertion is the statement `expect(add(1, 2)).toEqual(3)`, which describe that we expect function call `add(1, 2)` to equal to `3`.

`add` is imported from another file and is implemented like this:

```
function add(x, y) {
   return x + y;
}

module.exports = add
```

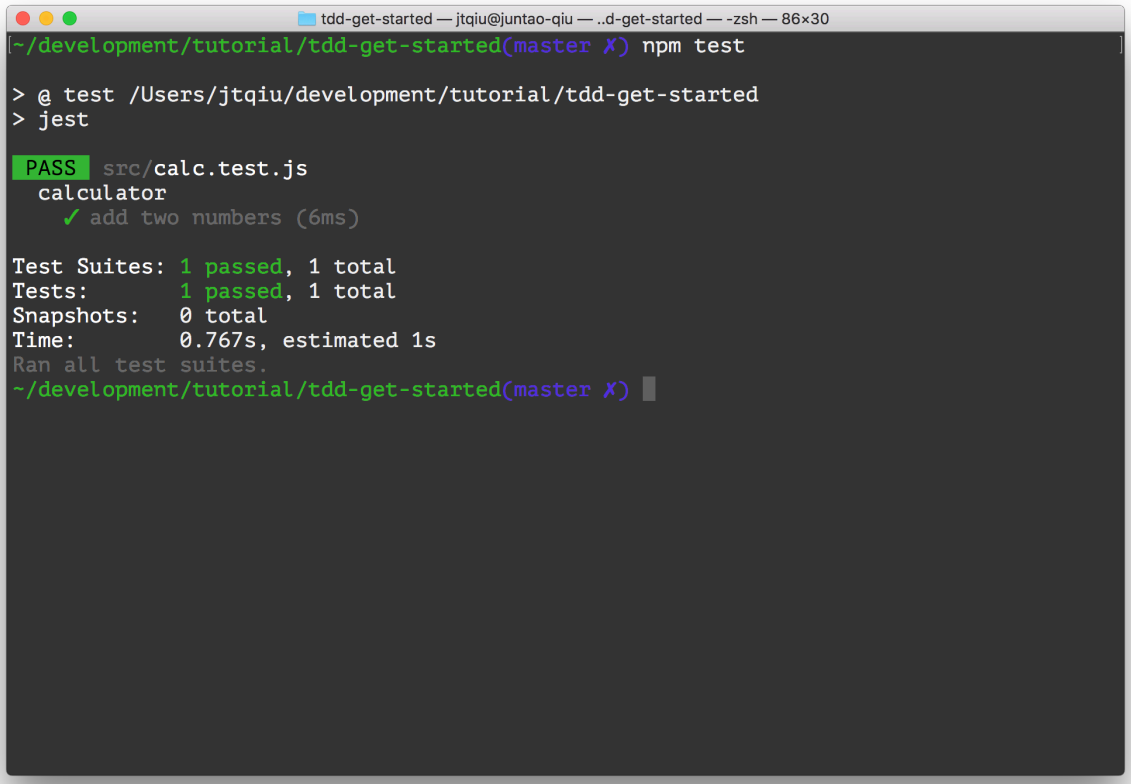
Then let's run the test and see how it goes:

```
node_modules/.bin/jest
```

alternatively, you can run the tests via

```
npm test
```

Which in turn, invokes `node_modules/.bin/jest` under the hood.



Great, we have the very first test up and running now.

Basic concepts in jest

In this section, we're about going through some basic concepts in `Jest`. We use `describe` to define a test block. Usually, we can use that mechanism to arrange different test cases and gather relevant test cases together as a group.

For example, we can put all `arithmetic` into one group:

```
describe('calculator', () => {
  it('should perform addition', () => {})
  it('should perform subtraction', () => {})
  it('should perform multiplication', () => {})
  it('should perform division', () => {})
})
```

What's more, we can still be using this approach recursively like:

```
describe('calculator', () => {
  describe('should perform addition', () => {
    it('add two positive numbers', () => {})
    it('add two negative numbers', () => {})
    it('add one positive and one negative numbers', () => {})
  })
})
```

The basic idea behind it is to making sure always group relevant tests together to make the test description make more sense for people who would maintain them. It could always benefit if you can describe the `description` (the first parameter for function `describe` and `it`) within the business context, in domain language.

For instance, when you are trying to develop a hotel reservation application, the tests should be read like:

```
describe('Hotel Sunshine', () => {
  describe('Reservation', () => {
    it('should make reservation when there are enough room available', () => {})
    it('should warn the administrator when only 5 available rooms left', () => {})
  })

  describe('Checkout', () => {
    it('should check if any appliance is broken', () => {})
    it('should refund guest when checkout earlier', () => {})
  })
})
```

There are times that you find that some duplicate code spread in test cases, for example, trying to set up the subject to be tested could be the case:

```
describe('addition', () => {
  it('add two positive numbers', () => {
    const options = {
      precision: 2
    }

    const calc = new Calculator(options)
    const result = calc.add(1.333, 3.2)
    expect(result).toEqual(4.53)
  })

  it('add two negative numbers', () => {
```

```
    const options = {
      precision: 2
    }

    const calc = new Calculator(options)
    const result = calc.add(-1.333, -3.2)
    expect(result).toEqual(-4.53)
  })

})
```

To reduce the duplication, we can use the `beforeEach` function `jest` provides to define some reusable instances of an object. It is invoked automatically before `jest` run each test case. In our case, the `calculator`, and then we can use it in all the test cases within the same `describe` block:

```
describe('addition', () => {
  let calc = null

  beforeEach(() => {
    const options = {
      precision: 2
    }
    calc = new Calculator(options)
  })

  it('add two positive numbers', () => {
    const result = calc.add(1.333, 3.2)
    expect(result).toEqual(4.53)
  })

  it('add two negative numbers', () => {
    const result = calc.add(-1.333, -3.2)
    expect(result).toEqual(-4.53)
  })
})
```

Of course, you may be wondering if there is a counterpart function named `afterEach` or something in charge of the cleanup work, there is!

```
describe('database', () => {
  let db = null;

  beforeEach(() => {
    db.connect('localhost', '9999', 'user', 'pass')
  })

  afterEach(() => {
    db.disconnect()
  })
})
```

Here we establish database connection before each test case and shut it down after. In practice, you may want to add rollback the database changes or other cleanups in `afterEach` step.

Additionally, if you want something to be set up before all the test cases one time and tear down after all of them are finished, then `beforeAll` and `afterAll` could help.

```
beforeAll(() => {
  db.connect('localhost', '9999', 'user', 'pass')
})

afterAll(() => {
  db.disconnect()
})
```

Using ES6

By default, you can only use `ES5`(a relatively old version of JavaScript) in `node.js`(interestingly, by the time(2018 March) I start to write this chapter, I cannot use most of the features in `ES6`. But I just checked(2019 April) in my terminal, most of the features are there). However, since now we're already in 2019, `ES6` should be the default programming language you choose for your frontend project. Good news is you don't have to wait till all the browsers implement the specification, you can use `babel` translate and compile the `ES6` code your write into `ES5`.

It's actually pretty easy to set up, only a few packages can make it work properly:

```
yarn add --dev babel-jest babel-core regenerator-runtime babel-preset-env
```

After the installation, define a `.babelrc` in your project root with content like:

```
{
  "presets": ["env"]
}
```

That's it! Now you should be able to write ES6 in your source and test code, and babel would do the rest:

```
import {add} from './calc'

describe('calculator', () => {
   it('add two numbers', () => {
           expect(add(1, 2)).toEqual(3)
   })
})
```

and

```
const add = (x, y) => x + y
export {add}
```

It's more neat and concise by using arrow function and one-line anonymous function (like function the `add`). Also, I like `import` and `export` much personally because of it more readable than the old `modules.export` stuff.

matchers

`Jest` provides numerous helper functions (matchers) for developers to do the assertion when writing tests. Those matchers can be used to assert a variety of data types in different scenarios.

Let's take a look at some basic usages first, and then move into some more advanced later on.

Basic usage

`toEqual` and `toBe` maybe the most common matchers you could find and probably would use in almost every test cases. Just as the name implies, there are used for assert whether values are equal to each other(the actual value and the expected value).

For example, it can be used for `string`, `number` or composed objects:

```
  it('basic usage', () => {
    expect(1+1).toEqual(2)
    expect('Juntao').toEqual('Juntao')
    expect({ name: 'Juntao' }).toEqual({ name: 'Juntao' })
  })
```

And for `toBe`

```
  it('basic usage', () => {
    expect(1+1).toBe(2)
    expect('Juntao').toBe('Juntao')
    expect({ name: 'Juntao' }).toBe({ name: 'Juntao' })
  })
```

The last test will fail. For primitives like strings, numbers, and booleans, you can use `toBe` to test the equality, and for Object, internally `jest` uses `Object.is` to check, which is strict and compares objects by memory address. So if you want to make sure all the fields match, use `toEqual`.

`Jest` also provide `.not` that you can use to assert the value oppositely:

```
  it('basic usage', () => {
    expect(1+2).not.toEqual(2)
  })
```

Sometimes, you might not want an exact match, say, you want a string to be matching some particular pattern. Then you can use `toMatch` instead:

```
  it('match regular expression', () => {
    expect('juntao').toMatch(/\w+/)
  })
```

In fact, you can write any valid regular expression:

```
  it('match numbers', () => {
    expect('185-3345-3343').toMatch(/^\d{3}-\d{4}-\d{4}$/)
    expect('1853-3345-3343').not.toMatch(/^\d{3}-\d{4}-\d{4}$/)
  })
```

Which is very convenient when you work with `strings`. Moreover, you can use comparisons with numbers:

```
  it('compare numbers', () => {
    expect(1+2).toBeGreaterThan(2)
    expect(1+2).toBeGreaterThanOrEqual(2)

    expect(1+2).toBeLessThan(4)
    expect(1+2).toBeLessThanOrEqual(4)
  })
```

Array and Object

`Jest` also provides matchers for `Array` and `Object`, for example, it's quite common to test if an element is contained in an `Array`:

```
    const users = ['Juntao', 'Abruzzi', 'Alex']

    it('match arrays', () => {
      expect(users).toContainEqual('Juntao')
      expect(users).toContain(users[0])
    })
```

Note that there is a difference between `toContain` and `toContainEqual`, basically `toContain` check if the item in a list by strictly comparing elements using `===`, on the other hand, `toContainEqual` just check the value (not the memory address).

For example, if you want to check whether an object is in a list:

```
    it('object in array', () => {
      const users = [
        { name: 'Juntao' },
        { name: 'Alex' }
      ]
      expect(users).toContainEqual({ name: 'Juntao' })
      expect(users).toContain({ name: 'Juntao' })
    })
```

The second assertion would fail since it uses a more strict comparison. Also, for an object, since its just a combination of other JavaScript primitives, so we can use `dot` notion and test the `existence` of the field or using matchers above for fields in an object.

```
    it('match object', () => {
      const user = {
        name: 'Juntao',
        address: 'Xian, Shaanxi, China'
      }

      expect(user.name).toBeDefined()
      expect(user.age).not.toBeDefined()
    })
```

The super `expect`

We have tasted the `matcher` a little bit in previous sections, let's take a look at another super weapon `Jest` provided `expect`.

There are a few useful helper functions attached in `expect` object (and it is also a function):

- expect.stringContaining
- expect.arrayContaining
- expect.objectContaining

By using those functions, you can define your own `matcher`. For example:

```
it('string contains', () => {
  const givenName = expect.stringContaining('Juntao')
  expect('Juntao Qiu').toEqual(givenName)
})
```

The variable `givenName` here is not a simple value, it's a new matcher and matches strings containing `Juntao`.

Similarly, you can use `arrayContaining` to check if a subset of an array:

```
  describe('array', () => {
    const users = ['Juntao', 'Abruzzi', 'Alex']

    it('array containing', () => {
      const userSet = expect.arrayContaining(['Juntao', 'Abruzzi'])
      expect(users).toEqual(userSet)
    })
  })
```

It looks a bit strange at first glance, but once you understand it, that pattern would help you to build more complicated matchers by yourself.

For instance, we get some data fetched from the backend API, an example of the payload looks like:

```
const user = {
  name: 'Juntao Qiu',
  address: 'Xian, Shaanxi, China',
  projects: [
    { name: 'ThoughtWorks University' },
    { name: 'ThoughtWorks Core Business Beach'}
  ]
}
```

Moreover, for some reasons, in our test, we don't care about `address` at all. But we do care if the `name` field contains `Juntao` and if the `project.name` contains `ThoughtWorks`.

So let's define a matcher by using the `stringContaining`, `arrayContaining` and `objectContaining`:

```
const matcher = expect.objectContaining({
  name: expect.stringContaining('Juntao'),
  projects: expect.arrayContaining([
    { name: expect.stringContaining('ThoughtWorks') }
  ])
})
```

This expression describes exactly what we want, and we then can use `toEqual` to do the assertion:

```
expect(user).toEqual(matcher)
```

That's pretty powerful. Basically, you can define matcher that just as you expected. It could be even used in `contract` between frontend and backend services.

Build your matchers

`Jest` also allows you to extend the `expect` object to define your own matchers. In that way, you can enhance the default matcher set or make the code more readable.

Let's see a concrete example. `jsonpath` is a library that can let developers play with `jsonpath` - similar to `xpath` in XML - with a JavaScript object.

Install it first if you haven't yet:

```
yarn add jsonpath --save
```

and then use it like:

```
import jsonpath from 'jsonpath'

const user = {
  name: 'Juntao Qiu',
  address: 'Xian, Shaanxi, China',
  projects: [
    { name: 'ThoughtWorks University' },
    { name: 'ThoughtWorks Core Business Beach'}
  ]
}

const result = jsonpath.query(user, '$.projects')
console.log(JSON.stringify(result))
```

and you will get:

```
[[{"name":"ThoughtWorks University"},{"name":"ThoughtWorks Core Business Beach"}]]
```

and query `$.projects[0].name`

```
const result = jsonpath.query(user, '$.projects[0].name')
```

would get

```
["ThoughtWorks University"]
```

the `query` would return an empty array `[]` if the path didn't match anything:

```
const result = jsonpath.query(user, '$.projects[0].address')
```

Let's define a matcher named `toMatchJsonPath` as a extension by using function `expect.extend`:

```
import jsonpath from 'jsonpath'

expect.extend({
  toMatchJsonPath(received, argument) {
    const result = jsonpath.query(received, argument)

    if (result.length > 0) {
      return {
        pass: true,
        message: () => 'matched'
      }
    } else {
      return {
        pass: false,
        message: () => `expected ${JSON.stringify(received)} to match jsonpath ${arg\
ument}`
      }
    }
  }
})
```

So internally, `Jest` would pass two parameters to the customising matcher, the first one is the actual result, the one you pass to function `expect()`. The second one, on the other hand, is the expected value you passed to the matcher, which in our case `toMatchJsonPath`.

For the return value, it's a simple JavaScript object that contains `pass`, which is a boolean value that indicates whether the test pass or not, and a `message` field to describe the reason of the pass or fail correspondingly.

After the definition, you can use it in your test just like any other built-in matchers:

```
describe('jsonpath', () => {
  it('match jsonpath', () => {
    const user = {
      name: 'Juntao'
    }
    expect(user).toMatchJsonPath('$.name')
  })

  it('mismatch jsonpath', () => {
    const user = {
      name: 'Juntao',
      address: 'ThoughtWorks'
    }
    expect(user).not.toMatchJsonPath('$.age')
  })
})
```

Pretty cool, right? It sometimes very useful when you want to make the `matcher` more readable by using some domain specific language.

For example:

```
const employee = {}
expect(employee).toHaveName('Juntao')
expect(employee).toBelongsToDepartment('Product Halo')
```

Mock

In many cases, you just don't want to make the real call to some underlying functions outside in terms of doing a unit test. So you would like to `mock` it – just pretend we're calling the real one. For example, you would not want to send an email to the client when you want to test some Email template functionality. Instead, you would like to see whether the HTML generated contains correct content, additionally, it tried to send email to some particular address. Connect to a product database to test the deletion API works would be not acceptable in most situations.

So we, as a developer, need to set up a mechanism to enable this. `Jest` provide a variety of ways to do the `mock`. The simplest one could be like the `jest.fn` for mocking a function:

```
it('create a callable function', () => {
  const mock = jest.fn()
  mock('Juntao')
  expect(mock).toHaveBeenCalled()
  expect(mock).toHaveBeenCalledWith('Juntao')
  expect(mock).toHaveBeenCalledTimes(1)
})
```

You can use `jest.fn()` to create a function that could be invoked just like other regular functions, except it provides the ability to be traced. A `mock` can track all the invoke against it, and it can record the invoke times, the parameter passed in each invokes. That could be very useful, since in many scenarios we just want to ensure the particular function has been called with specified parameters, in the correct order – we don't have to do the real invoke.

A dummy `mock` object as the previous one doesn't do anything interesting, but the following one is more meaningful:

```
it("mock implementation", () => {
  const fakeAdd = jest.fn().mockImplementation((a, b) => 5)

  expect(fakeAdd(1, 1)).toBe(5)
  expect(fakeAdd).toHaveBeenCalledWith(1, 1)
})
```

Instead of defining a static `mock`, you can define an implementation by yourself, the real implementation could be very complicated, maybe it does some calculation based on the complex formula on the given parameters.

Additionally, just imagine we have a function that sends a remote API call to fetch data:

```
export const fetchUser = (id, process) => {
  return fetch(`http://localhost:4000/users/${id}`)
}
```

In the test code, especially in a unit test, we don't want to send any remote calls, so we use `mock` instead:

```
describe('mock API call', () => {
  const user = {
    name: 'Juntao'
  }

  it('mock fetch', () => {
    // given
    global.fetch = jest.fn().mockImplementation(() => Promise.resolve({user}))
    const process = jest.fn()

    // when
    fetchUser(111).then(x => console.log(x))

    // then
    expect(global.fetch).toHaveBeenCalledWith('http://localhost:4000/users/111')
  })
})
```

We expect that the `fetch` is invoked by `http://localhost:4000/users/111`, note the `id` we are using here. And we can see that the user information is printed out on the console:

```
 PASS  src/advanced/matcher.test.js
  â—Ž Console

    console.log src/advanced/matcher.test.js:152
      { user: { name: 'Juntao' } }
```

That is something very useful. In fact, `Jest` provide other `mock` mechanism as well, but we are not going to discuss them in here. And we are not using any advanced features other than what we have already described in this chapter.

If you are interested, please check the `jest` help or homepage for more information.

Summary

We learned how to set up `ES6` and `jest` at the beginning of the Chapter, and then walked through some fundamental knowledge of `jest` test framework. Different types of `matcher` and how to use them. In the end, we even defined one `jsonpath` matcher by ourselves, it surely could simplify the matching process in the test, and make tests more readable and concise.

Test Driven Design 101

In this chapter, we will learn together how to apply TDD in your daily development by a step-to-step guide. Along with this demo, you will get the idea of how to split a big task into relative smaller ones, and accomplish each one wait a set of passing tests and a bit refactoring tricks. Just before we dive into the code, let's get some fundamental understanding of how to write a proper test.

Writing tests

So how would you write a test? Typically there are 3 steps (as always, even put an elephant into a fridge). Firstly, do some preparation work, like set up the database, initialise the object to be tested or set up some fixture data for testing. Secondly, invoke the method/function under test, you usually would assign the result to some variable. Finally, do some assertions to see whether the result is expected or not.

It's usually described as `Given`, `When` and `Then` or in the so-called `3 A` format, where `3 A` stands for `Arrange`, `Action` and `Assertion`. Either way is describing basically the same thing.

In the `Given` clause, you can describe all the preparation (set up dependencies). In step `When` you trigger the action or invoke the subject to be tested, usually a function call with prepared parameters, and finally in `Then` you examine the result to see if it matches the expected result in some way (equals to something exactly, or contains particular patterns or throws an error, and so on)

For example:

```
// given
const user = User.create({ name: 'Juntao', address: 'ThoughtWorks Software Technolog\
ies (Melbourne)' })

// when
const name = user.getName()
const address = user.getAddress()

// then
expect(name).toEqual('Juntao')
expect(address).toEqual('ThoughtWorks Software Technologies (Melbourne)')
```

Typically you can split the test cases into many and let each one has one single assertion, like:

```
it('creates user name', () => {
  // given
  const user = User.create({ name: 'Juntao', address: 'ThoughtWorks Software Technol\
ogies (Melbourne)' })

  // when
  const name = user.getName()

  // then
  expect(name).toEqual('Juntao')
});

it('creates user address', () => {
  // given
  const user = User.create({ name: 'Juntao', address: 'ThoughtWorks Software Technol\
ogies (Melbourne)' })

  // when
  const address = user.getAddress()

  // then
  expect(address).toEqual('ThoughtWorks Software Technologies (Melbourne)')
});
```

Triangulation

There are a couple of ways to writing tests and driving the implementation. Once the commonly accepted approach is `triangulation`. Let's take a close look at how to do it with some examples.

1. write one test that could drive out something

Imagine we are trying to implement a calculator with `TDD`. A test for `addition` could be the scenario to get started with.

The first test

So a specification of `addition` could be

```
describe('addition', () => {
  it('return 5 when add 2 and 3', () => {
      const a = 2, b = 3
      const result = add(a, b)
      expect(result).toEqual(5)
  })
})
```

And the simplest implementation even could be

```
const add = () => 5
```

You might argue that it makes no sense to write code like this but bear with me for now. So what the second test could be?

We can just write another example of `addition`:

```
it('return 6 when add 2 and 4', () => {
    const a = 2, b = 4
    const result = add(a, b)
    expect(result).toEqual(6)
})
```

To make the test pass, the simplest solution then turns to:

```
const add = (a, b) => 2 + b
```

The idea is to write a failed but specific test to drive the implementing code more generic, in each step. So now the implement is generic than in the first step. However, still, it has space to improve.

The third test could be something like:

```
it('return 7 when add 3 and 4', () => {
    const a = 3, b = 4
    const result = add(a, b)
    expect(result).toEqual(7)
})
```

This time there are no patterns in the test data anymore to follow, we have to write something more complicated to make it pass. The implementation turns to:

```
const add = (a, b) => a + b
```

Now the implement is more generic and could cover most of `addition` cases. In the future, our calculator might need to support `addition` for imaginary numbers, we can still do that by adding more tests to drive out the solution in the same way.

This approach of writing tests is called `Triangulation`: you write a failed test and write just *enough* code to make the test pass, then you write another test to drive the changes from another angle, that in turn, would lead you to make the implementation more generic a little bit. You keep work in this manner, step by step until the code became generic enough to support most of the cases in terms of meeting the business requirement.

At first glance, it might be too simple and slow to do that, but that's the foundation you can rely on. For a simple task, you do that, and for more complicated ones you still applying the same approach. That's actually the ability to simplify the task and split big task to smaller pieces.

Ok, let's move one step further by looking into one complicated example to learn how can we apply `TDD` in some relatively tough scenarios.

Tasking by example

In the project I'm currently working on, the team uses a very simple manner to track the efforts put on each user story (a small chunk of work that could be accomplished individually). Usually, a card could be in the following status: `analysis`, `doing`, `testing`, `done` through its lifecycle. However, it could be `block` when something it depends on is uncompleted or not ready yet.

The measure of efforts on stories we're employing is a pretty simple one. Basically, we track how many days it took when developers spent time on coding, or how many days had it been blocked. The project manager then could have the chance to know how the progress looks like and how healthy the project is, and what further actions could be taken to improve.

We put a `d` in lower case in the title of a card to indicate that it has been under `development` for half a day, and an uppercase `D` means a full day. Not surprisingly, `q` for half `qa` day, and `Q` for a whole one `qa` day. Therefore, by any given moment, you would see something like this on the title of card `[ddDQbq] Allow user login to their profile page`, `b` for blocked here of course.

Then let's build a parser that can read the tracking marks `ddDQbq` and translate it into some human-readable format like this:

```
{
  "Dev days": 2.0,
  "QA days": 1,
  "Blocked": 0.5
}
```

Looks pretty straightforward, right? Cannot wait to jump to write the code? Hold on, let's get started with a test first, and get some feeling about how to apply `TDD` in such a case.

So the first question could be: **how can we split a task like this into smaller tasks that easy to achieve and be verified?** For sure there is more than one way to do it, a reasonable split could be:

1. Write a test to make sure we can translate `d` to half dev day
2. Write a test to make sure we can translate `D` to one dev day
3. Write a test to handle more that one mark like `dD`
4. Write a test to handle `q`
5. Write a test to handle `qQ`
6. Write a test to handle `ddQ`

The split itself is essential for applying TDD practically. The small tasks could be engaging and encourage you in different ways:

1. It's fun (it has been proven that small achievement could pleasant your brain by some chemical called dopamine)
2. Make sure you get fast feedback
3. Know how the progress like of the task

The first test

OK, enough theory and let's get our hands dirty. According to the output of `tasking`, the first test could be

```
1  it('translate d to half dev day', () => {
2    expect(translate('d')).toEqual({'Dev': 0.5})
3  })
```

And pretty straightforwardly, the implement could be as simple as:

```
1  const translate = () => ({'Dev': 0.5})
```

It ignores the input and returns a dummy `{'Dev': 0.5}`, but you have to admire that it just fulfilled the requirement regarding the current sub-task. Quick and dirty, but works.

The second test

Let's check to cross the first todo from our task list, and moving on.

```
it('translate D to one dev day', () => {
  expect(translate('D')).toEqual({'Dev': 1.0})
})
```

What's the most straightforward solution come to your mind? Correct!

```
const translate = (c) => (c === 'd' ? {'Dev': 0.5}: {'Dev': 1.0})
```

I know it seems relative "stupid" to write code in this way. However, as you can see, our implementation is driven by the related tests, as long as the test passes – which means the requirements are meet – we could call it satisfied. The only reason we write code is to fulfil some business requirements anyway, right?

Since the test is now passing, you can do some refactoring if you find something could be improved, say, `magic numbers`, the method body is too long, and so on. However, for now, I suppose we're ok to continue.

So the third test could be:

```
it('translate dD to one and a half dev day', () => {
  expect(translate('dD')).toEqual({'Dev': 1.5})
})
```

Hmm, things become more complicated now, we have to parse the string of character individually and sum up the result. Code snippet following should do the trick:

```
const translate = (input) => {
  let sum = 0;
  input.split('').forEach((c) => sum += c === 'd' ? 0.5: 1.0)
  return {'Dev': sum}
}
```

Now our program can handle all the `d` or `D` combination sequence like `ddd` or `DDdDd` with no problem. Then comes the task 4:

```
  it('translate q to half qa day', () => {
    expect(translate('q')).toEqual({'QA': 0.5})
  })
```

It seems we need a `sum` function for each status, for example, `sum` in `Dev`, `sum` in `QA`. It would be more convenient if we can refactor the code a little bit to make that change easier. And the most beautiful part of TDD emerges - we don't have to worry about break the existing functionalities by accident since we have the tests to cover them.

Refactoring

Let's extract the parse out as a function itself, and use that function in `translate`. Function `translate` then could be something like this after the refactoring:

```
const parse = (c) => {
  switch(c) {
    case 'd': return {status: 'Dev', effort: 0.5}
    case 'D': return {status: 'Dev', effort: 1}
  }
}

const translate = (input) => {
  const state = {
    'Dev': 0,
    'QA': 0
  }

  input.split('').forEach((c) => {
    const {status, effort} = parse(c)
    state[status] = state[status] + effort
  })

  return state
}
```

Now it should be effortless to make the new test pass. We can add one new `case` in `parse`:

```
const parse = (c) => {
  switch(c) {
    case 'd': return {status: 'Dev', effort: 0.5}
    case 'D': return {status: 'Dev', effort: 1}
    case 'q': return {status: 'QA', effort: 0.5}
  }
}
```

Cool, I like it.

One step further

Also, the same with mark `Q`. In fact, for the task that contains different marks, there is no change required in the code at all. However, as a decent programmer, we could keep cleaning the code up to an ideal status, for example, we could extract the parse as some look-up dictionary:

```
const dict = {
  'd': {
    status: 'Dev',
    effort: 0.5
  },
  'D': {
    status: 'Dev',
    effort: 1.0
  },
  'q': {
    status: 'QA',
    effort: 0.5
  },
  'Q': {
    status: 'QA',
    effort: 1.0
  }
}
```

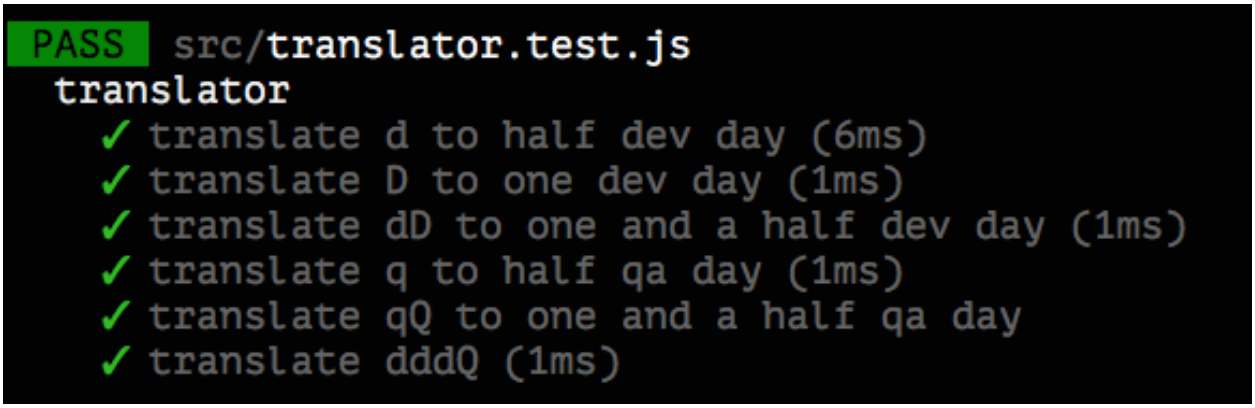
and that would simplify the `parse` to something like

```
const parse = (c) => dict[c]
```

You can even extract the `dict` as data into a separate file named `constants` and import it in `translator.js` for the sake of clearness. And for the `forEach` in function `translate`, we could use `Array.reduce` to make it even shorter:

```
const translate = (input) => {
  const items = input.split('')
  return items.reduce((accumulator, current) => {
    const { status, effort } = parse(current)
    accumulator[status] = (accumulator[status] || 0) + effort
    return accumulator
  }, {})
}
```

Nice and clean, right!



Summary

We learned the basic 3 steps for writing a proper test. And understand now how to use `Triangulation` to drive out different paths in tests. We then learned how to do a `tasking` to help us write tests and walked through a reasonable small program in `TDD` fashion step by step, and finally got something useful in a real-life scenario.

Project setup

Requirements

In this book, we are going to develop a web application from scratch. We call it `Bookish`, it's a rather simple application about `books` just as the name implied. In the application, a user could have a book list and can search by some keywords, and users are allowed to navigate to book detail page and review the `description` and `review` or `ranking` of the book. For the sake of simplicity, we will finish part of the features, via an iterative manner, and of course, by applying `ATDD` along the way.

In the application, we will develop several typical features, which are:

Feature 1 - book list

In the real world, the granularity of a feature would be much bigger than the ones we're using in this book, typically, there would be many user stories within a feature, such as Booklist, pagination, the styling of a book in the list, and so on. Let's assume there is only one story in a feature here:

- Show the book list

We can describe the user story in this form:

```
As a user, I want to see a list of books, so that I can learn something.
```

This is a very wild-used format of a user story, by describing `As a <role>`, it emphasis that who would be beneficial of this feature the most, and by saying `I want to <do something>`, you're saying how the user would interact with the system. Finally, `so that <value>` sentence would point out what's the business value precisely behind this feature.

This format force one to think in the stakeholder's perspective, and hopefully can tell both business analyst and developer what's the most important (valuable) point of the user story they are working on.

So the acceptance criteria are:

- Given there are `10` books in the system, a user should see `10` items on the page
- In each item, the following information should be displayed: Book name, author, price, rating.

And for acceptance criteria, sometimes it would be written in a form manner like:

```
Given there are `10` books in the library
When a user visits the homepage
Then he/she would see `10` books on the page
And each book would contain at least `name`, `author`, `price` and `rating`
```

The `given` clause states the current status of the application, `when` means the user trigger some action, like, click a button or navigate to some page, and `then` is an assertion that asserts the application performs just as we expected.

Feature 2 - book detail

- Show book detail

```
As a user, I want to check the detail of a book, so that I can quickly get the brief without reading through it
```

And the acceptance criteria are:

- User clicks an item in book list and would be redirected to the detail page
- There are: Book name, author, price, description, review on the detail page

Feature 3 - Searching

- Searching book by name

```
As a user, I want to search book by its name, so that I can quickly find what I'm interested in
```

And the acceptance criteria are::

- User type `Refactoring` as keyword
- Only The books with name contain `Refactoring` shows in the book list

Feature 4 - Reviews

- Besides the necessary information of a book in detail page

```
As a user, I want to be able to add a book review to a book I read previously so that people have the same interesting could decide if it is worthwhile to read.
```

- A user can read the reviews on the detail page
- A user can post a review to a particular book
- A user can edit the review he/she posted

Create the project

Package management

Let's get started with some essential package installation and configuration at first. Make sure you have `node.js` installed locally, after that, you can use `npm` to install the tools we need to build our `Bookish` application (we have already covered that part in the previous Chapter, check it out in case you didn't settle down that):

```
npm install yarn create-react-app --global
```

create-react-app

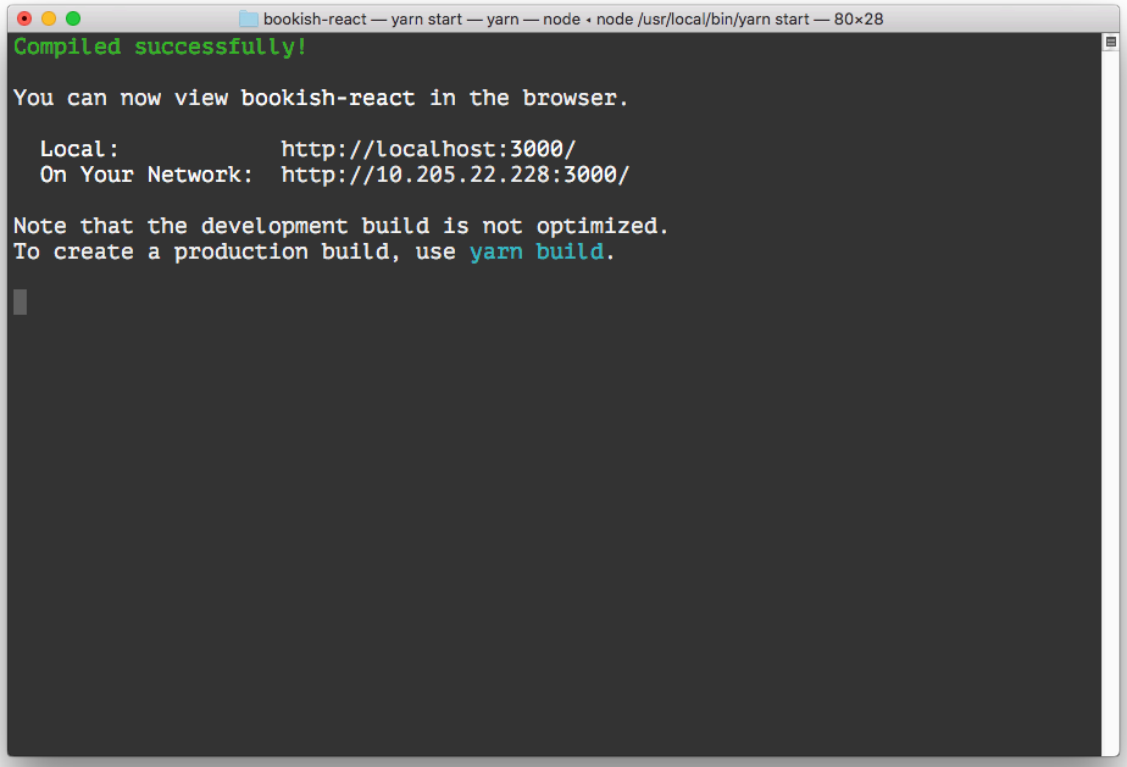
After the installation, we can use `create-react-app` provided by Facebook to create our project:

```
create-react-app bookish-react
```

`create-react-app` could help us install `react`, `react-dom` and a command line tool named `react-scripts` by default. Moreover, it will download those libraries and their dependencies automatically, such as `webpack`, `babel`, and so on. By using `create-react-app` we basically can do zero-config to make the application up and running.

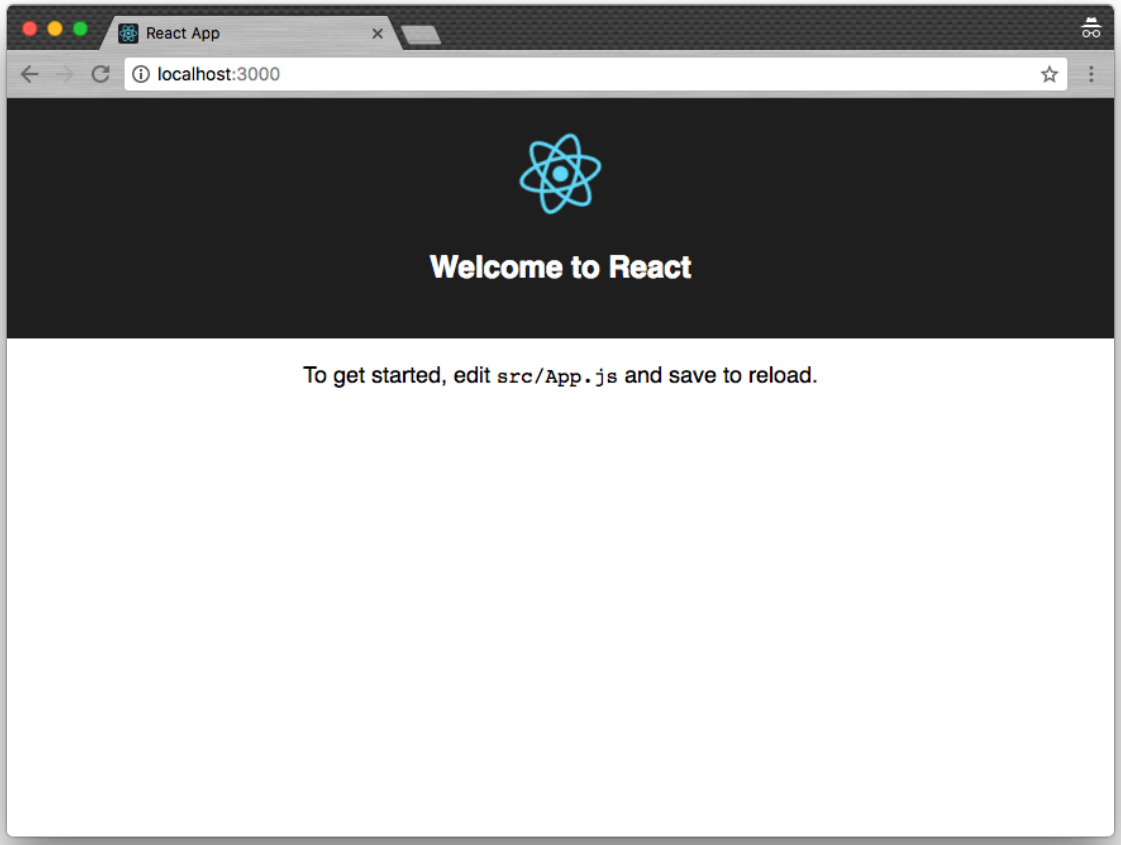
After the creation, as the console log suggested, we need to jump into `bookish-react` folder, and run `yarn start`:

```
cd bookish-react
yarn start
```



Terminal

And there would be a new browser tab opened automatically with address fulfilled `http://localhost:3000`, the UI should look like this



Yarn start

Project Structure

We don't need all of the files generated by `create-react-app`, let's do some clean up first. We can remove all the irrelative files in `src` folder like this:

```
src
├── App.css
├── App.js
├── index.css
└── index.js
```

And modify the file content as following:

```
 import React, { Component } from 'react';
-import logo from './logo.svg';
 import './App.css';

 class App extends Component {
   render() {
     return (
       <div className="App">
-        <header className="App-header">
-          <img src={logo} className="App-logo" alt="logo" />
-          <h1 className="App-title">Welcome to React</h1>
-        </header>
-        <p className="App-intro">
-          To get started, edit <code>src/App.js</code> and save to reload.
-        </p>
+        <h1>Hello world</h1>
       </div>
     );
   }
```

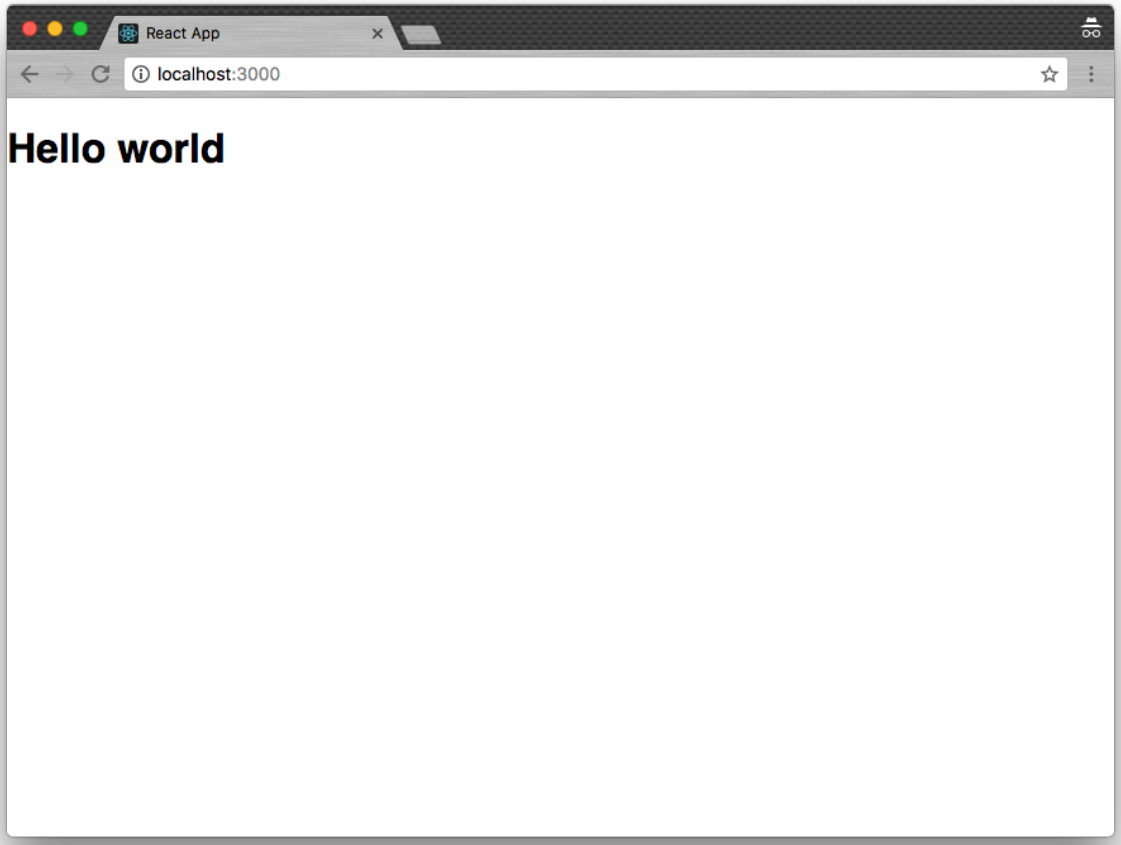
```
 import React from 'react';
 import ReactDOM from 'react-dom';
 import './index.css';
 import App from './App';
-import registerServiceWorker from './registerServiceWorker';

-ReactDOM.render(<App />, document.getElementById('root'));
-registerServiceWorker();
+ReactDOM.render(<App />, document.getElementById('root'));
```

Clean up the CSS file content as well:

```
- .App {
-   text-align: center;
- }
-
-.App-logo {
-  animation: App-logo-spin infinite 20s linear;
-  height: 80px;
-}
-
-.App-header {
```

```
11  -  background-color: #222;
12  -  height: 150px;
13  -  padding: 20px;
14  -  color: white;
15  -}
16  -
17  -.App-title {
18  -  font-size: 1.5em;
19  -}
20  -
21  -.App-intro {
22  -  font-size: large;
23  -}
24  -
25  -@keyframes App-logo-spin {
26  -  from { transform: rotate(0deg); }
27  -  to { transform: rotate(360deg); }
28  -}
```

Then we got the UI like this:



Puppeteer

After that, we need to set up the `acceptance tests` environment first. In this book, we'll use `Puppeteer` to do the UI tests. `Puppeteer` from Google is built on top of Headless Chrome, and it provides a lot of APIs to do the DOM manipulation, JavaScript evaluation, which can be used for run the `UI` tests.

First, we need to install it locally:

```
1 yarn add puppeteer --dev
```

with `--dev` option means we just add it as a dependency for the development stage, we don't want to include it in production code.

Our first end 2 end test

The most challenging thing about TDD might be where to start, how we write the very first test?

Our first test could be:

- Make sure there is a `Heading` element on the page, the content is `Bookish`

This test looks useless at first glance, but actually, it can make sure that:

- Frontend code can compile and translate
- The browser can render our page correctly(without any script errors)

So, first we create a file `e2e.test.js` with the following content:

```
1  import puppeteer from 'puppeteer'
2
3  const appUrlBase = 'http://localhost:3000'
4
5  let browser
6  let page
7
8  beforeAll(async () => {
9    browser = await puppeteer.launch({})
10   page = await browser.newPage()
11 })
12
13 describe('Bookish', () => {
14   test('Heading', async () => {
15     await page.goto(`${appUrlBase}/`)
16     await page.waitForSelector('h1')
17     const result = await page.evaluate(() => {
18       return document.querySelector('h1').innerText
19     })
20
21     expect(result).toEqual('Bookish')
22   })
23 })
24
25 afterAll(() => {
26   browser.close()
27 })
```

I know it looks a little bit scary, but it simple. Let's review it block by block.

```
import puppeteer from 'puppeteer'
```

Firstly, we import `puppeteer`, and then specify the `devServer` address, so `Puppeteer` knows whether to access the application.

```
const appUrlBase = 'http://localhost:3000'
```

in `beforeAll` hook, we start the Chrome in `headless` mode:

```
beforeAll(async () => {
  browser = await puppeteer.launch({})
  page = await browser.newPage()
})
```

and then stop it in `afterAll` hook.

```
afterAll(() => {
  browser.close()
})
```

async & await

Note here we're marking the anonymous function in `beforeAll` as `async` and putting an `await` keyword before `puppeteer.launch()`.

The `async` and `await` are the ES6 syntax that can make the asynchorize programming much easier in JavaScript. Let's take a look at a simple example:

```
function fetchContactById(id) {
  return fetch(`http://localhost:8080/contacts/${id}`).then((response) => {
    return response.json()
  })
}
```

In function `fetchContactById`, we use `fetch` to send http request and return the json response back (as a `Promise` object). Then we can consume this `Promise` object just by:

```
function main() {
  fetchContactById(1).then((contact) => {
    console.log(contact.name)
  })
}
```

However, by using `async/await`, we can simply define a `variable` to wait for the return of the function call – the process is blocked – and once the underlying `Promise` is resolved, the contact has the correct value, and the control process is back and `console.log` is evaluated.

```
async function main() {
  const contact = await fetchContactById(1)
  console.log(contact.name)
}
```

There are no much differences you may say, but if we enhance the example above a little bit, say, by adding another API call `fetchUserById` and we need the returned value of that API to send the `fetchContactById`:

```
function fetchUserById(id) {
  return fetch(`http://localhost:8080/users/${id}`).then((response) => {
    return response.json()
  })
}
```

Then the code becomes like:

```
function main() {
  fetchUserById('juntao').then((user) => {
    fetchContactById(user.id).then((contact) => {
      console.log(contact.name)
    })
  })
}
```

the indention could keep growing and growing if there are requests send one by one, but by using the `await/async` pair, the code would be much concise like:

```
async function amain() {
  const user = await fetchUserById('juntao')
  const contact = await fetchContactById(user.id)
  console.log(contact.name)
}
```

That's much more compact and neat!

Then let's take a look at the main block:

```
describe('Bookish', () => {
  test('Heading', async () => {
    await page.goto(`${appUrlBase}/`)
    await page.waitForSelector('h1')
    const result = await page.evaluate(() => {
      return document.querySelector('h1').innerText
    })

    expect(result).toEqual('Bookish')
  })
})
```

In the test case, we access port `3000` by using `puppeteer`, and wait for `h1` to show up, and then invoke `evaluate` to call the native DOM script:

```
document.querySelector('h1').innerText
```

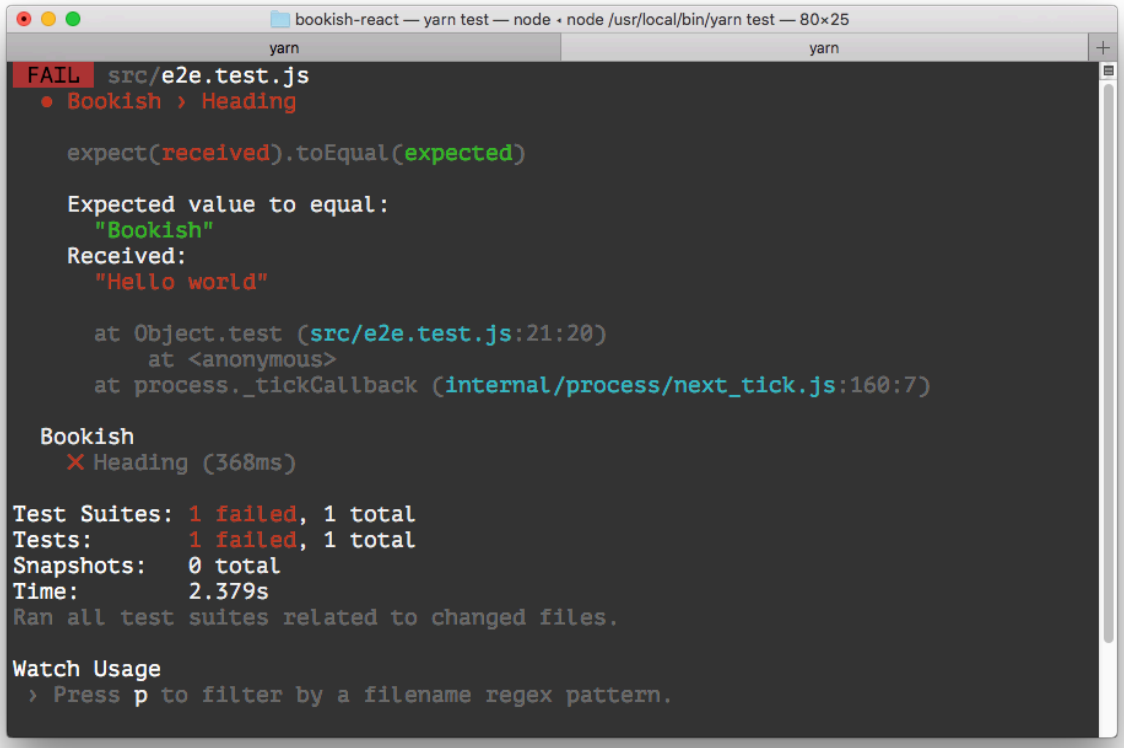
once we get the content of `h1`, we can do a assertion:

```
expect(result).toEqual('Bookish')
```

Since `create-react-app` has already built `jest` testing framework in, we can simply run the following command to run the tests:

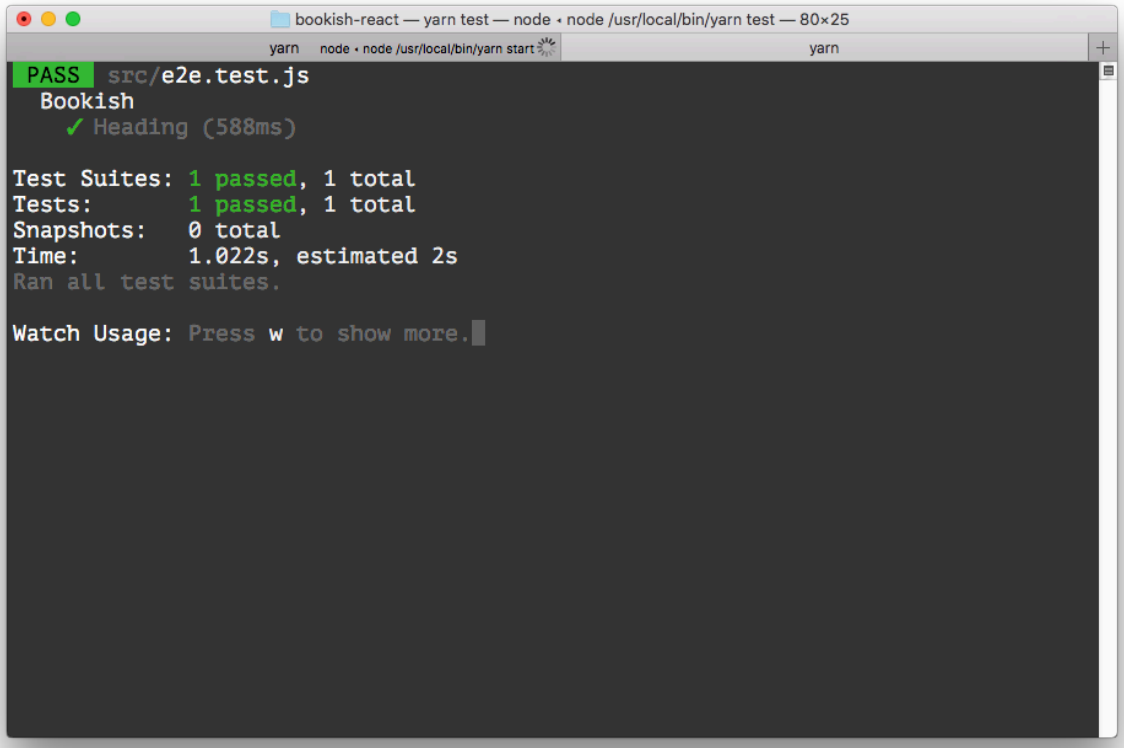
```
yarn test
```

Of course, we don't have any implementation yet, the test failed:



Still, remember the `red-green-refactor` cycle? Since the test now failed (in red), we should make it pass first. We can modify the content in `h1` to `Bookish`:

```
1  class App extends Component {
2    render() {
3      return (
4        <div className="App">
5          <h1>Bookish</h1>
6        </div>
7      );
8    }
9  }
```



Great, the test now passes. Go on the cycle, Hnn, refactor. But for now, it seems ok so we can skip this.

Commit code to VCS (Version Control System)

OK, we now have an acceptance test and its' implementation, we can put the code to version control in case we need to look back in the future. I'm going to use `git` in this book since it's the most popular one and you could find it installed in almost every developer's computer.

Run the following command could `init` the current folder as a `git` repository:

```
1 git init
```

Then commit it locally, of course, you may want to push it to some remote like in Github or GitLab to share with other colleagues:

```
1 git add .
2 git commit -m "make the first e2e test pass"
```

If you have something don't want to be published or shared with others, current a `.gitignore` text file, and put the filename you don't want to share in it, like:

```
*.log
.idea/
```

That's it!

Now look at what we've got here:

- A running acceptance test suite
- A page that can render `Bookish` as the `heading`

Great achievement, isn't? We can now commence on the business requirement implementation.

Implement the Book List

Our first requirement is to develop a book list. From acceptance test's perspective, all we have to do is make sure that the page contains a list of books, we don't need to worry about what technology that will be used to implement the page, it doesn't matter whether the page is dynamically generated or just a static HTML, as long as the test passes at this stage.

A list of books

Firstly, we add a test case in `e2e.test.js` like the following:

```
      expect(result).toEqual('Bookish')
    })
+
+  test('Book List', async () => {
+    await page.goto(`${appUrlBase}/`)
+    await page.waitForSelector('.books')
+    const books = await page.evaluate(() => {
+      return [...document.querySelectorAll('.book .title')].map(el => el.innerText)
+    })
+
+    expect(books.length).toEqual(2)
+  })
 })

 afterAll(() => {
```

And we expect that there is a `.books` container, which contains serval `.book` element, and `.book` contains a `.title` element. Now if we run the test, it will fail miserably. According to the principle of `TDD`, we can make a quick implementation, modify the `App.js` like should do the work:

```
      return (
        <div className="App">
          <h1>Bookish</h1>
+         <div className="books">
+           <div className="book">
+             <h2 className="title"></h2>
+           </div>
+           <div className="book">
+             <h2 className="title"></h2>
+           </div>
+         </div>
        </div>
      );
    }
```

Book name

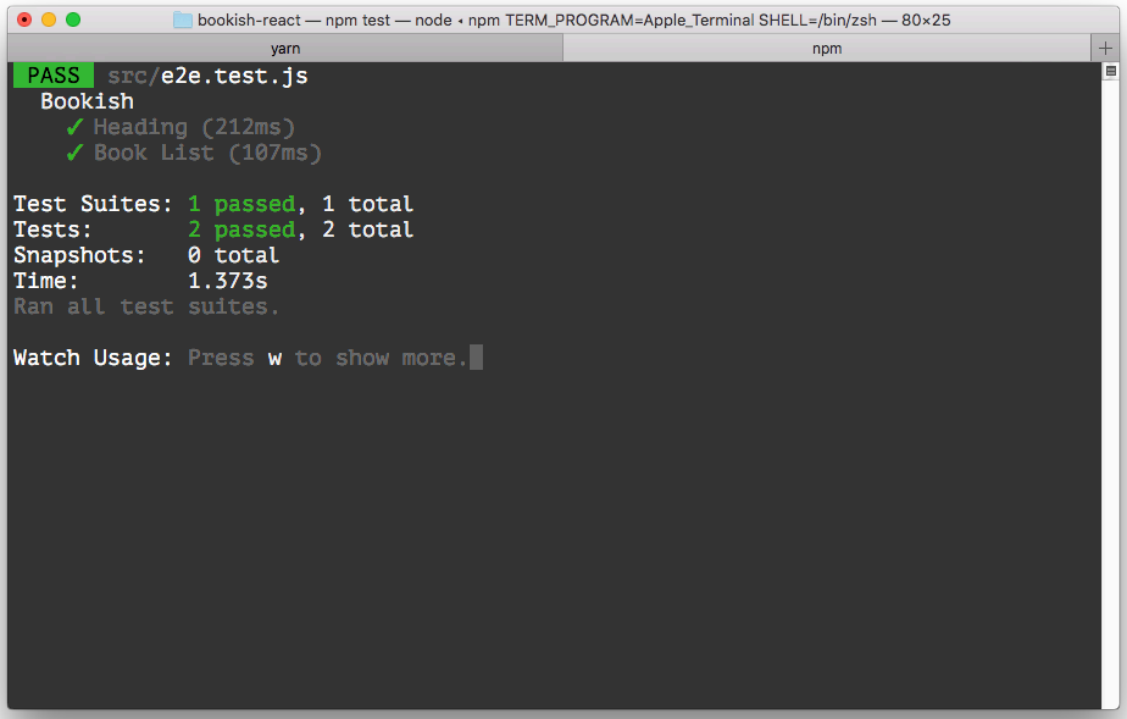
Great, the test is now passing. So just as you can see, we have *driven* the HTML structure by test, now let's add another expectation in the test:

```
      })

      expect(books.length).toEqual(2)
+     expect(books[0]).toEqual('Refactoring')
+     expect(books[1]).toEqual('Domain-driven design')
    })
  })
```

To make the test pass, the simplest way would be, not surprisingly, using hard code:

```
          <div className="books">
            <div className="book">
-             <h2 className="title"></h2>
+             <h2 className="title">Refactoring</h2>
            </div>
            <div className="book">
-             <h2 className="title"></h2>
+             <h2 className="title">Domain-driven design</h2>
            </div>
          </div>
        </div>
```

Awesome! Our tests pass again.



booklist

Now it's time to take a look at the code to see if there are any code smells, and do the corresponding refactoring if possible. Firstly, we find that put all the `.book` in method `render` might not be a good idea, we can use a `for-loop` to generate this HTML.

Refactoring

We can define a new function named `renderBooks` and then use `map` to generate the list instead.

```
  renderBooks(books) {
    return (<div className="books">
      {
        books.map(book => {
          return (<div className="book">
            <h2 className="title">{book.name}</h2>
          </div>)
        })
      }
    </div>)
  }
```

When invoke it, we can pass an array of book object like:

```
  render() {
    const books = [{ name: 'Refactoring' }, { name: 'Domain-driven design' }]
    return (
      <div className="App">
        <h1>Bookish</h1>
        {this.renderBooks(books)}
      </div>
    );
  }
```

Our tests are still passing. That's to say, we make our internal implementation better, but without modifying the external behavior, this is just one of the benefits TDD can provide: easier and safer for refactoring.

Refactoring again

Now the code is clean and compact than before, but it could be better. By applying `Componentization`, the granularity of abstraction should be based on `component` instead of `function`. For example, now we are using function `renderBooks` to render a passed array to a booklist, we should abstract a component named `BookList`, and create a file `BookList.js` like this:

```
import React from 'react'

function BookList({books}) {
  return (<div className="books">
    {
      books.map(book => {
        return (<div className="book">
          <h2 className="title">{book.name}</h2>
        </div>)
      })
    }
  </div>)
}

export default BookList
```

Now we can use this customized component just as `React` built-in components like `div` or `h1`:

```
  render() {
    const books = [{ name: 'Refactoring' }, { name: 'Domain-driven design' }]
    return (
      <div className="App">
        <h1>Bookish</h1>
        <BookList books={books} />
      </div>
    );
  }
```

By doing this refactoring, our code becomes much more declarative and easier to understand. Additionally, our tests remain `green`.

Network

Generally speaking, the data of book list don't hardcoded in the code. In real life, those data are stored somewhere remotely and need to be fetched when application launching. To make our application works in that way, we should do those tasks at least:

- Config a mock server to provides book data we need
- Using client-side network library `axios` to fetch data from the mock server
- Using the data fetched to render our component

mock server

A mock server is wild using during the development process, and here we'll introduce `json-server` as our mock server. It's a very lightweight and easy-to-get-started package.

First, we need to install it globally:

```
npm install json-server --global
```

And create an empty folder named `mock-server`

```
mkdir -p mock-server
cd mock-server
```

And then create a `db.json` file with the following content:

```
{
  "books": [{ "name": "Refactoring" }, { "name": "Domain-driven design" }]
}
```

This file defines the `route` and data for that `route`, now we can launch the server by:

```
json-server --watch db.json --port 8080
```

If you open the browser to access `http://localhost:8080/books`, you should see something like this:

```
[
  {
    "name": "Refactoring"
  },
  {
    "name": "Domain-driven design"
  }
]
```

Of course, you can use `curl` to fetch it from the command line:

```
$ curl http://localhost:8080/books

[
  {
    "name": "Refactoring"
  },
  {
    "name": "Domain-driven design"
  }
]
```

Now our mock server is up and running, and the application now can fetch those data via HTTP.

Async request

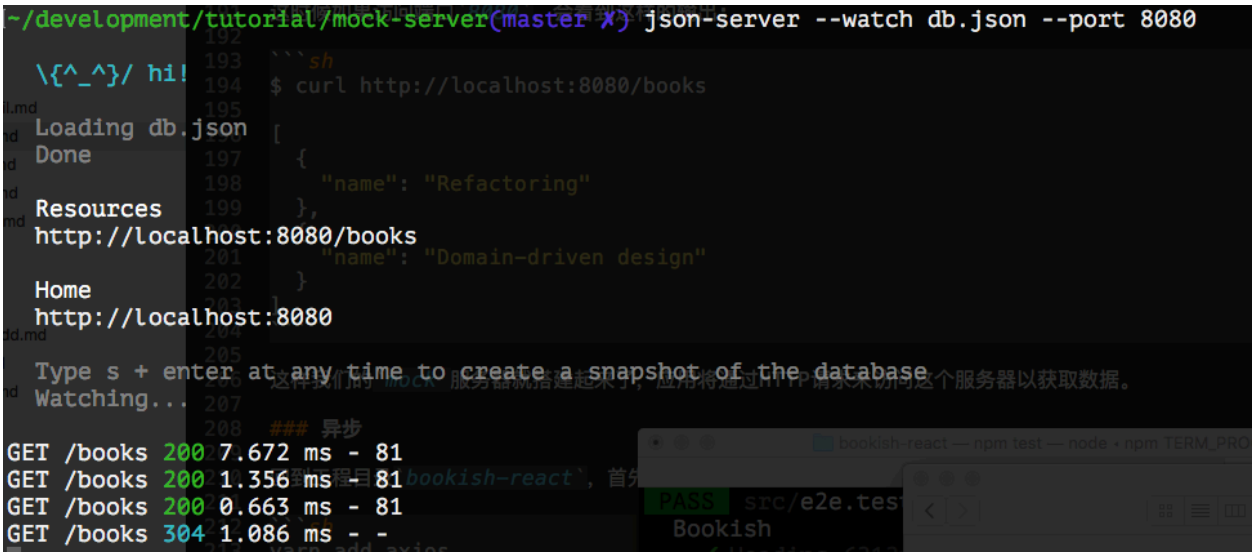
Back to the application folder `bookish-react`, to send the request and get the data, we need to install `axios` in our project:

```
yarn add axios
```

and then we can use it to fetch data in `App.js`:

```
 class App extends Component {
+  constructor(props) {
+    super(props)
+    this.state = {
+      books: []
+    }
+  }
+
+  componentDidMount() {
+    axios.get('http://localhost:8080/books').then(res => {
+      this.setState({
+        books: res.data
+      })
+    })
+  }
+
   render() {
-    const books = [{ name: 'Refactoring' }, { name: 'Domain-driven design' }]
+    const {books} = this.state
     return (
       <div className="App">
```

You can see the console log from the mock server, the `books` API is reached:



mock server

Setup & Teardown

Now let's take a look at the code and tests, there is an implicit assumption: tests know that the implementation would return 2 books. The only problem is this assumption makes the tests look not that direct: why we are expecting `expect(books.length).toEqual(2)`, Instead of 3? Why those two books are `Refactoring` and `Domain-driven design`? That kind of assumption should be avoided or should be clearly defined somewhere in the tests.

Before each test runs, we should create some fixture data and clear them up after each test.

Actually `json-server` provides a programmatic way to launch. Programmatically, we can define the behaviors of the mock server by some code.

Let's do that. In folder `mock-server`, create a file named `server.js`, and add some `middlewares` in it:

```
server.use((req, res, next) => {
  if (req.method === 'DELETE' && req.query['_cleanup']) {
    const db = router.db
    db.set(req.entity, []).write()
    res.sendStatus(204)
  } else {
    next()
  }
})
```

This function can do some action based on the HTTP request method and query strings. If the request is a `DELETE` method and there is a `_cleanup` parameter in the query string, then we will clean the entity the request is requesting. For example. When you using `DELETE` to access `http://localhost:8080/books?_cleanup=true`, then this function will set the `books` to empty.

After that, you can launch the server by the command follows:

```
node server.js
```

The complete version of the mock-server code is hosted here[6]. Once we have this middleware, we can use it in our tests setup and teardown hooks:

```
  afterEach(() => {
    return axios.delete('http://localhost:8080/books?_cleanup=true').catch(err => er\
r)
  })

  beforeEach(() => {
    const books = [
      {"name": "Refactoring", "id": 1},
      {"name": "Domain-driven design", "id": 2}
    ]

    return books.map(item => axios.post('http://localhost:8080/books', item, {header\
s: { 'Content-Type': 'application/json' }}))
  })
```

Before each test, we insert 2 books into the mock-server by `POST` them to URL: `http://localhost:8080/books`, and after each test, we clean them up by send `DELETE` request to endpoint `'http://localhost:8080/books?_cleanup=true'`.

```
  beforeEach(() => {
    const books = [
      {"name": "Refactoring", "id": 1},
      {"name": "Domain-driven design", "id": 2},
      {"name": "Building Micro-service", "id": 3}
    ]

    return books.map(item => axios.post('http://localhost:8080/books', item, {header\
s: { 'Content-Type': 'application/json' }}))
  })
```

Now we can modify the data in setup whatever we want, say, add another book `Building Micro-service`, and expecting got `3` books in the test:

[6] https://github.com/abruzzi/react-tdd-mock-server

```
test('Book List', async () => {
  await page.goto(`${appUrlBase}/`)
  await page.waitForSelector('.books')
  const books = await page.evaluate(() => {
    return [...document.querySelectorAll('.book .title')].map(el => el.innerText)
  })

  expect(books.length).toEqual(3)
  expect(books[0]).toEqual('Refactoring')
  expect(books[1]).toEqual('Domain-driven design')
  expect(books[2]).toEqual('Building Micro-service')
})
```

Loading indicator

Our application is fetching data remotely, and there is no guarantee that the data could return immediately. If our user is using this application in some great legacy environment, we would like there is some indicator shows to make the user experience better. Additionally, when there is no network connection at all (or a timeout), we need to show some error message.

Before we jump to add this to code, let's imagine that how can we simulate those two scenarios:

- Slow request
- Fail request

Unfortunately, neither of those two scenarios is easy to simulate, and even we can do that, we have to couple our test with the code tightly. Let's re-think what we want to do carefully: there are 3 status of the component(loading, error, success), so if we can test the behaviors under that 3 status isolated, then we can make sure our component is functional.

Refactor first

To make the test easy to write, we need to refactor a little bit first. Take a look at `App.js`:

```
import BookList from './BookList'

class App extends Component {
  constructor(props) {
    super(props)
    this.state = {
      books: []
    }
  }

  componentDidMount() {
    axios.get('http://localhost:8080/books').then(res => {
      this.setState({
        books: res.data
      })
    })
  }

  render() {
    const {books} = this.state
    return (
      <div className="App">
        <h1>Bookish</h1>
        <BookList books={books} />
      </div>
    );
  }
}
```

What if we do some method extraction like this:

```
   constructor(props) {
     super(props)
     this.state = {
-      books: []
+      books: [],
+      loading: true,
+      error: null
     }
   }

   componentDidMount() {
     axios.get('http://localhost:8080/books').then(res => {
```

```
13          this.setState({
14  -         books: res.data
15  +         books: res.data,
16  +         loading: false
17  +       })
18  +     }).catch(err => {
19  +       this.setState({
20  +         loading: false,
21  +         error: err
22      componentDidMount() {
23        axios.get('http://localhost:8080/books').then(res => {
24          this.setState({
25  -         books: res.data
26  +         books: res.data,
27  +         loading: false
28  +       })
29  +     }).catch(err => {
30  +       this.setState({
31  +         loading: false,
32  +         error: err
33          })
34        })
35      }
36
37      render() {
38  -     const {books} = this.state
39  +     const {loading, error, books} = this.state
40  +
41  +     if (loading) {
42  +       return <div className="loading" />
43  +     }
44  +
45  +     if (error) {
46  +       return <div className="error" />
47  +     }
48  +
49        return (
50          <div className="App">
51            <h1>Bookish</h1>
```

Emm, it's workable, but the disadvantage is that we coupled the network requesting with rendering together, that make the unit test very hard. So let's separate the network and render:

Container Component

We can extract the network related content into a separate file named `BookListContainer`, and there is no render logic at all:

```
import React, { Component } from 'react';

import axios from 'axios'
import BookList from './BookList'

class BookListContainer extends Component {
  constructor(props) {
    super(props)
    this.state = {
      books: []
    }
  }

  componentDidMount() {
    axios.get('http://localhost:8080/books').then(res => {
      this.setState({
        books: res.data
      })
    })
  }

  render() {
    return <BookList {...this.state} />
  }
}

export default BookListContainer
```

The `App.js` is simplified correspondingly like this:

```
import React, {Component} from "react";
import "./App.css";
import BookListContainer from "./BookListContainer";

function App() {
  return (
    <div className="App">
      <h1>Bookish</h1>
      <BookListContainer />
    </div>
  )
}

export default App;
```

Then we are good to do the unit test of `BookList` component for the 3 status we mentioned above. Let's start with unit tests to driven the `BookList` component:

Unit test

Before we add any unit tests, we need to add some packages:

```
yarn add enzyme enzyme-adapter-react-16 --dev
```

Create a `setupTests.js` under `src` folder, and configure the `enzyme` in this file with the content:

```
import Enzyme from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';

Enzyme.configure({ adapter: new Adapter() });
```

Those allow us to do the unit tests without worry about configuring the enzyme stuff. To handle the `loading` and `error` cases, we can either cover them in the unit test or higher-level tests. Consider the test pyramid theory, and we should put them in unit tests:

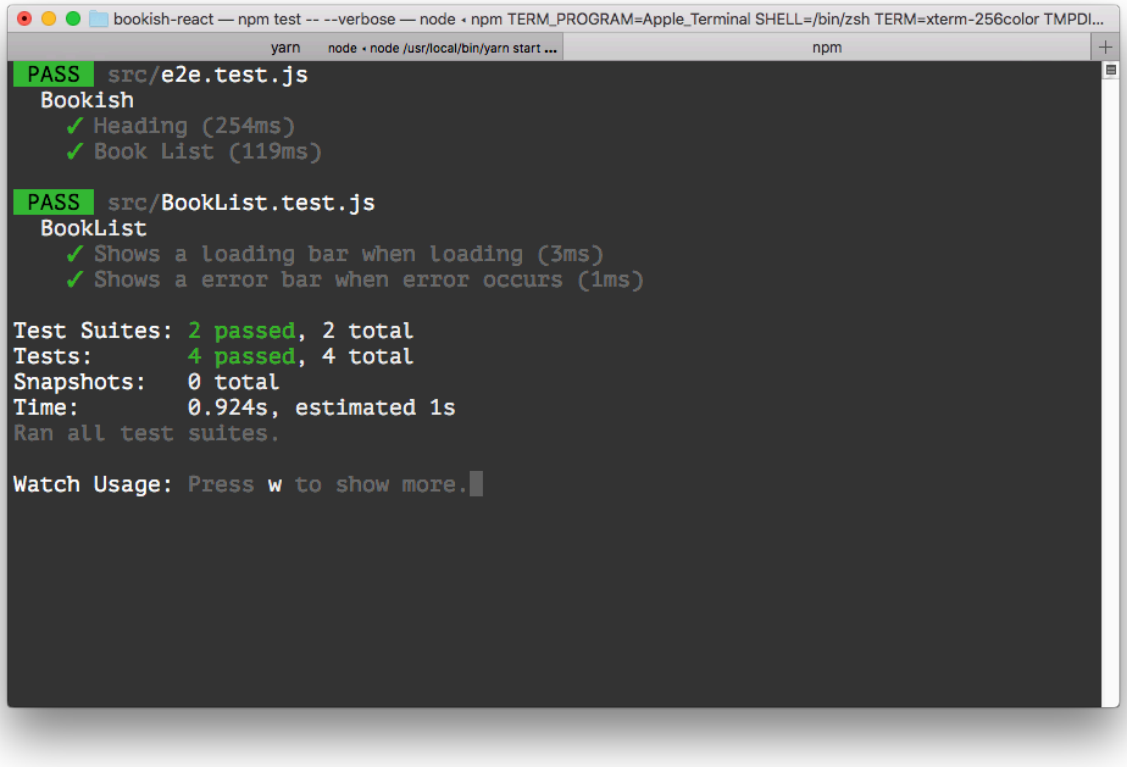
```
import React from 'react'
import {shallow} from 'enzyme'
import BookList from './BookList'

describe('BookList', () => {
  it('Shows a loading bar when loading', () => {
    const props = {
      loading: true
    }
    const wrapper = shallow(<BookList {...props}/>)
    expect(wrapper.find('.loading').length).toEqual(1)
  })
})
```

We are using `shallow` provided by `enzyme` to render `BookList`. Since we don't have it yet, the test would fail, we can make a quick implementation like:

```
function BookList({loading, books}) {
  if (loading) {
    return <div className="loading" />
  }

  return (<div className="books">
    {
      books.map(book => {
        return (<div className="book">
          <h2 className="title">{book.name}</h2>
        </div>)
      })
    }
  </div>)
}
```

Moreover, for the network error case:

```
1   it('Shows a error bar when error occurs', () => {
2     const props = {
3       error: {
4         "message": "Something went wrong"
5       }
6     }
7     const wrapper = shallow(<BookList {...props}/>)
8     expect(wrapper.find('.error').length).toEqual(1)
9   })
```



unit tests

Finally, we can add a `happy-path` to make sure our component renders in the success scenario:

```
  it('Shows a list of books', () => {
    const props = {
      books: [
        {name: "Refactoring"},
        {name: "Building Micro-service"}
      ]
    }
    const wrapper = shallow(<BookList {...props}/>)
    expect(wrapper.find('.book .title').length).toEqual(2)
  })
```

You may be wondering we have already tested this case in `Acceptance test`, is this a duplication? Well, *Yes* and *No*, the cases in the unit test can be used as documentation, it specifies that what arguments the component requires, field names and types. For example, in the `props`, we explicitly show that `BookList` requires an object with `books` field, which is an array.

After running the tests, we can see a warning in the console:

```
Warning: Each child in an array or iterator should have a unique "key" prop.

    Check the top-level render call using <div>. See https://fb.me/react-warning-key\
s for more information.
        in div (at BookList.js:15)
```

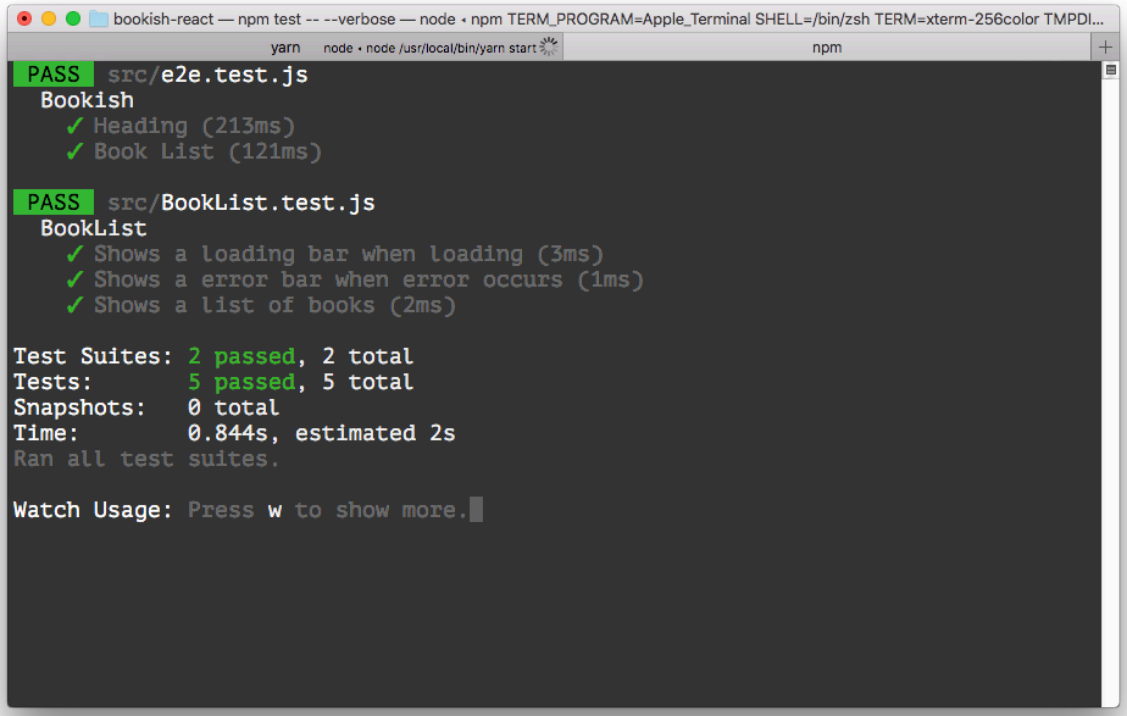
That means when rendering a list, `React` require a unique `key` for each of the items, like `id`. We can quickly fix it by:

```
  it('Shows a list of books', () => {
    const props = {
      books: [
        {name: "Refactoring", id: 1},
        {name: "Building Micro-service", id: 2}
      ]
    }
    const wrapper = shallow(<BookList {...props}/>)
    expect(wrapper.find('.book .title').length).toEqual(2)
  })
```

Now, our **final** version of `BookList` is:

```
import React from 'react'

function BookList({loading, error, books}) {
  if (loading) {
    return <div className="loading" />
  }

  if (error) {
    return <div className="error" />
  }

  return (<div className="books">
    {
      books.map(book => {
        return (<div className="book" key={book.id}>
          <h2 className="title">{book.name}</h2>
        </div>)
      })
    }
  </div>)
}

export default BookList
```

All unit tests are passing, cool!



unit tests

Now let's modify the usage of `BookList`, add the `loading` and `error` attributes:

```
class BookListContainer extends Component {
  constructor(props) {
    super(props)
    this.state = {
      books: [],
      loading: true,
      error: null
    }
  }

  componentDidMount() {
    axios.get('http://localhost:8080/books').then(res => {
      this.setState({
        books: res.data,
        loading: false
      })
    }).catch(err => {
```

```
18        this.setState({
19          loading: false,
20          error: err
21        })
22      })
23    }
24
25    render() {
26      return <BookList {...this.state} />
27    }
28  }
```

Note

Sometimes, we may find that it's complicated to write tests to some code: there are a lot of external dependencies. Then we need to do refactoring first, extract the dependencies out, and then add tests.

Another thing is that if a test terminated unexpectedly, the tests after it would fail. That's because we inserted some data in `beforeEach`, but since the test terminated, the cleanup stage is not executed, so you may need to clean up it manually by:

```
1  curl http://localhost:8080/books?_cleanup=true -X DELETE
```

After the `books` is reset, then rerun the tests should be ok.

Implement the Book Detail

For each book in the book list, we want it has a hyperlink, and when a user clicks this link the browser will redirect to the detail page. Detail page contains more detail content like book title, cover image, description, reviews and so on.

Acceptance test

We can describe this requirement as an acceptance test like this:

```
  test('Goto book detail page', async () => {
    await page.goto(`${appUrlBase}/`)
    await page.waitForSelector('a.view-detail')

    const links = await page.evaluate(() => {
      return [...document.querySelectorAll('a.view-detail')].map(el => el.getAttribu\
te('href'))
    })

    await Promise.all([
      page.waitForNavigation({waitUntil: 'networkidle2'}),
      page.goto(`${appUrlBase}${links[0]}`)
    ])

    const url = await page.evaluate('location.href')
    expect(url).toEqual(`${appUrlBase}/books/1`)
  })
```

Run the test, and it failed. We don't have `/books` route yet, and we don't have the link. To make the test pass, we add a hyperlink in `BookList` component:

```
    books.map(book => {
      return (<div className="book" key={book.id}>
        <h2 className="title">{book.name}</h2>
+           <a href={`/books/${book.id}`} className="view-detail">View Detail</a>
      </div>)
    })
  }
```

Also, to make sure the page shows expected content after the navigation, we need to add those lines in `e2e.test.js`:

```
      const url = await page.evaluate('location.href')
      expect(url).toEqual(`${appUrlBase}/books/1`)
+
+     await page.waitForSelector('.description')
+     const result = await page.evaluate(() => {
+       return document.querySelector('.description').innerText
+     })
+     expect(result).toEqual('Refactoring')
```

That makes sure the page has `.description` section and its content is `Refactoring`. The test fails again, let's fix it by adding client-side routing to our application.

Routing

Firstly, we need to add `react-router` and `react-router-dom` as dependencies, and they provide the client-side routing mechanism for us.

```
yarn add react-router react-router-dom
```

So in `index.js`, we import `BrowserRouter` and wrap the `<App />` by it. Then the whole application can share the global `Router` configurations.

```
-ReactDOM.render(<App />, document.getElementById('root'));
+import {BrowserRouter as Router} from 'react-router-dom'
+
+ReactDOM.render(<Router>
+  <App />
+</Router>, document.getElementById('root'));
```

We then define two routes in `App.js`:

```
+import BookDetailContainer from './BookDetailContainer'
+
+import {Route} from 'react-router-dom'

 function App() {
   return (
     <div className="App">
       <h1>Bookish</h1>
-      <BookListContainer />
+      <main>
+        <Route exact path="/" component={BookListContainer} />
+        <Route path="/books/:id" component={BookDetailContainer} />
+      </main>
     </div>
   )
 }
```

Finally, we need to create a new file `BookDetailContainer.js`, it's pretty similar to `BookListContainer.js`, except that the `id` of the book will be passed through `react-router` in `match.params.id`. Once we have the id, we can send HTTP request to load book detail, and put the fetching code in `componentDidMount` hook:

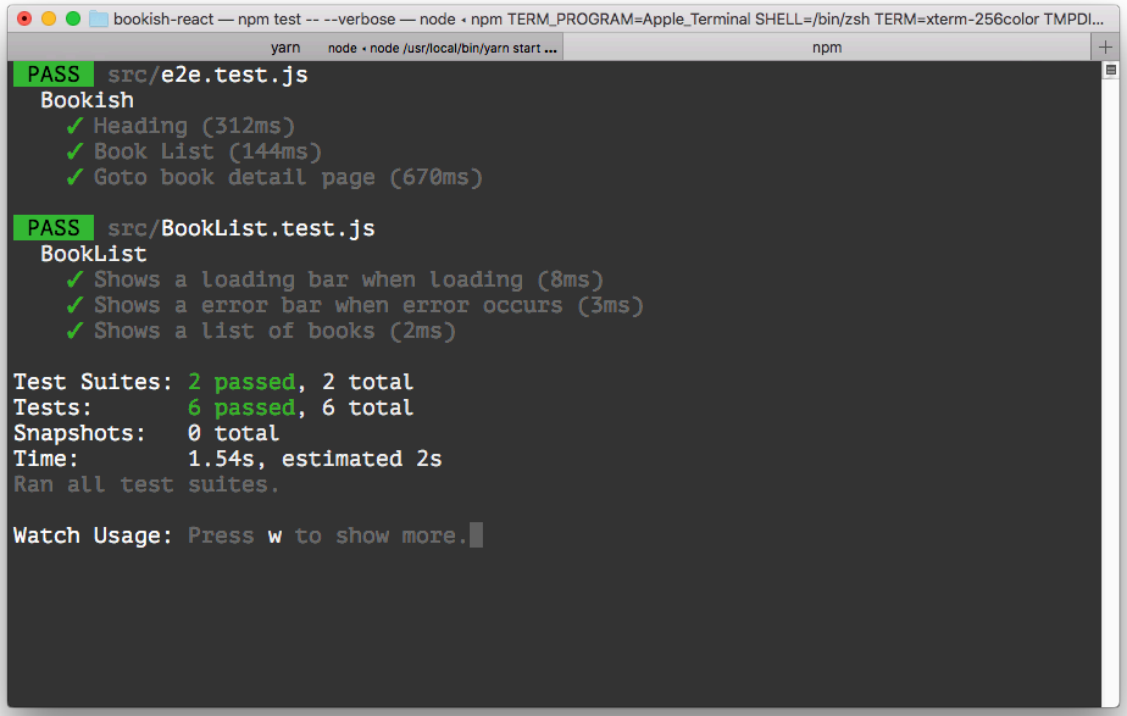
```
import React, {Component} from 'react'
import axios from 'axios'

class BookDetailContainer extends Component {
  constructor(props) {
    super(props)
    this.state = {
      book: {}
    }
  }

  componentDidMount() {
    const id = this.props.match.params.id
    axios.get(`http://localhost:8080/books/${id}`).then((res) => {
      this.setState({
        book: res.data
      })
    })
  }

  render() {
```

```
22      const {book} = this.state
23      return (<div className="detail">
24        <div className="description">{book.description}</div>
25      </div>)
26    }
27  }
28
29  export default BookDetailContainer
```

One more thing is that we need to add `description` field for the fixture in `beforeEach` block of the end-to-end test:

```
1     beforeEach(() => {
2       const books = [
3         {"name": "Refactoring", "id": 1, "description": "Refactoring"},
4         {"name": "Domain-driven design", "id": 2, "description": "Domain-driven design\
5  "},
6         {"name": "Building Micro-service", "id": 3, "description": "Building Micro-ser\
7  vice"}
8       ]
9
10      return books.map(item => axios.post('http://localhost:8080/books', item, {header\
11 s: { 'Content-Type': 'application/json' }}))
12    })
```

Great, our tests are now passing!



detail

Unit tests

In the end-to-end test, we just make sure there is a `.description` section in the page. If we want to verify all the fields are correctly rendered on the page, we can add them to lower level test – unit test. Unit tests run fast and check more specific than end-to-end tests.

Before we add the unit test for `BookDetailContainer`, let's do a small refactor first because that the `BookDetailContainer` is coupled with `axios.get` stuff, and we don't want to involve this in unit level tests.

Let's extract a `BookDetail` component from `BookDetailContainer` first:

```
import React from 'react'

function BookDetail({book}) {
  return (<div className="detail">
    <div className="description">{book.description}</div>
  </div>)
}

export default BookDetail
```

And the `render` method of `BookDetailContainer` then be simplified as:

```
  render() {
    return <BookDetail {...this.state}/>
  }
```

Now we can quickly add unit tests in file `BookDetail.test.js`, and try to drive the implementation:

```
  it('Shows book name', () => {
    const props = {
      book: {
        name: "Refactoring",
        description: "The book about how to do refactoring"
      }
    }
    const wrapper = shallow(<BookDetail {...props}/>)
    expect(wrapper.find('.name').text()).toEqual("Refactoring")
  })
```

The implementation could be straightforward:

```
import React from 'react'

function BookDetail({book}) {
  return (<div className="detail">
    <h2 className="name">{book.name}</h2>
    <div className="description">{book.description}</div>
  </div>)
}

export default BookDetail
```

File structure

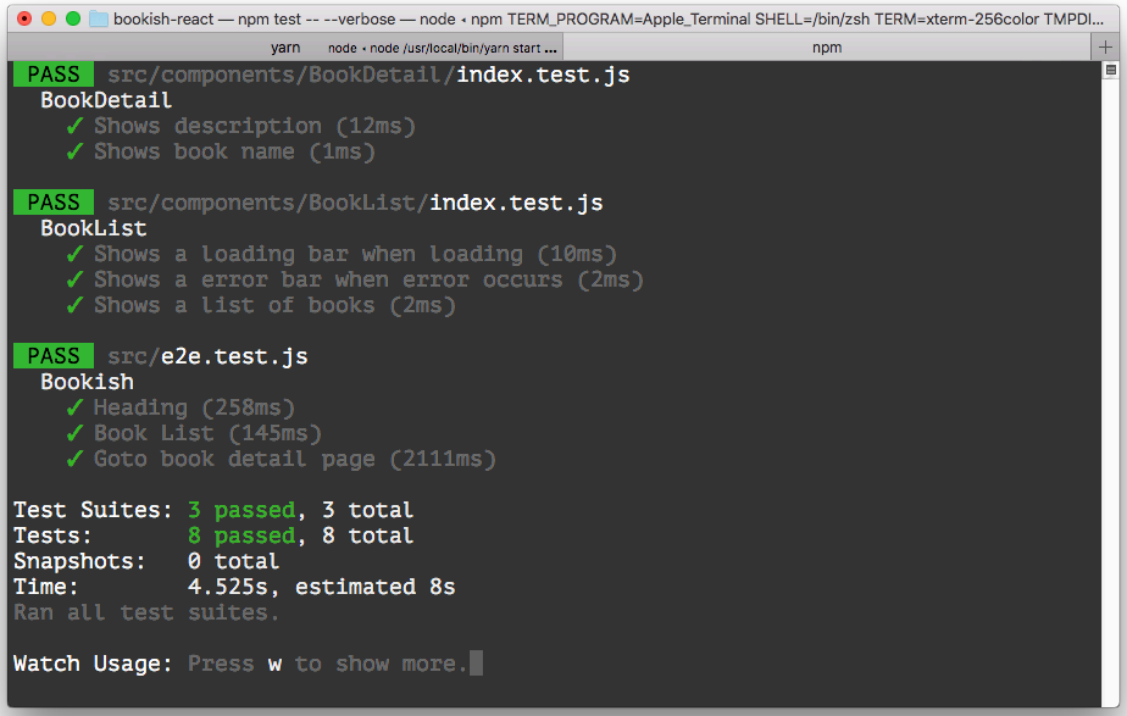
Currently, the file structure is pretty flattened, and there is no hierarchy at all, all the files are just there in one folder. That is some sort of bad smell, it makes us very difficult to find what we're looking for, let's re-structure them.

```
src
├── App.css
├── App.js
├── BookDetail.js
├── BookDetail.test.js
├── BookDetailContainer.js
├── BookList.js
├── BookList.test.js
├── BookListContainer.js
├── e2e.test.js
├── index.css
├── index.js
└── setupTests.js
```

In `React` community, a prevalent pattern to classify component is split them into `container` component and `presentational` component. While `container` component responds to fetch and transform data, and `presentational` component renders data whatever passed from outside.

We put all `containers` in to `containers` folder, and for `presentational` component we simply put them into a folder named `components`:

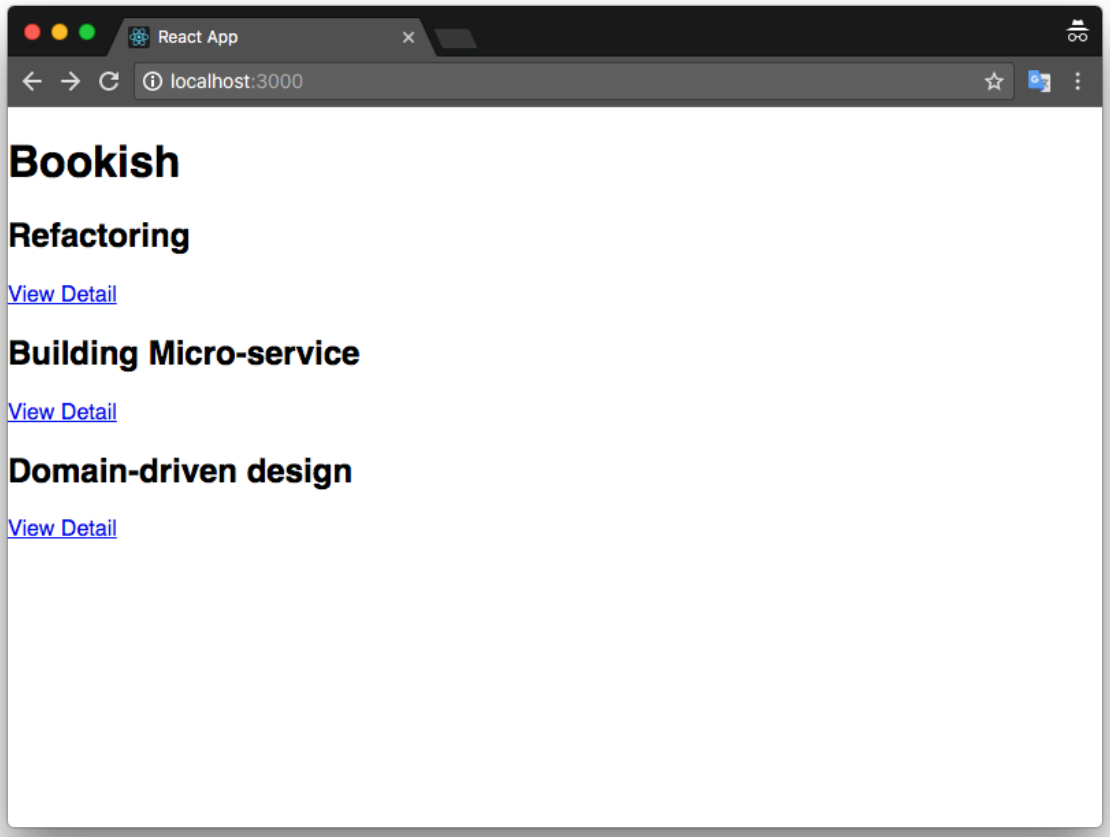
```
src
├── App.css
├── App.js
├── components
│   ├── BookDetail
│   │   ├── index.js
│   │   └── index.test.js
│   └── BookList
│       ├── index.js
│       └── index.test.js
├── containers
│   ├── BookDetailContainer.js
│   └── BookListContainer.js
├── e2e.test.js
├── index.css
├── index.js
└── setupTests.js
```



restructured

User Interface

We've already finished two exciting and challenging features by now. However, the user interface is too simple and ugly, let's add some style for that.



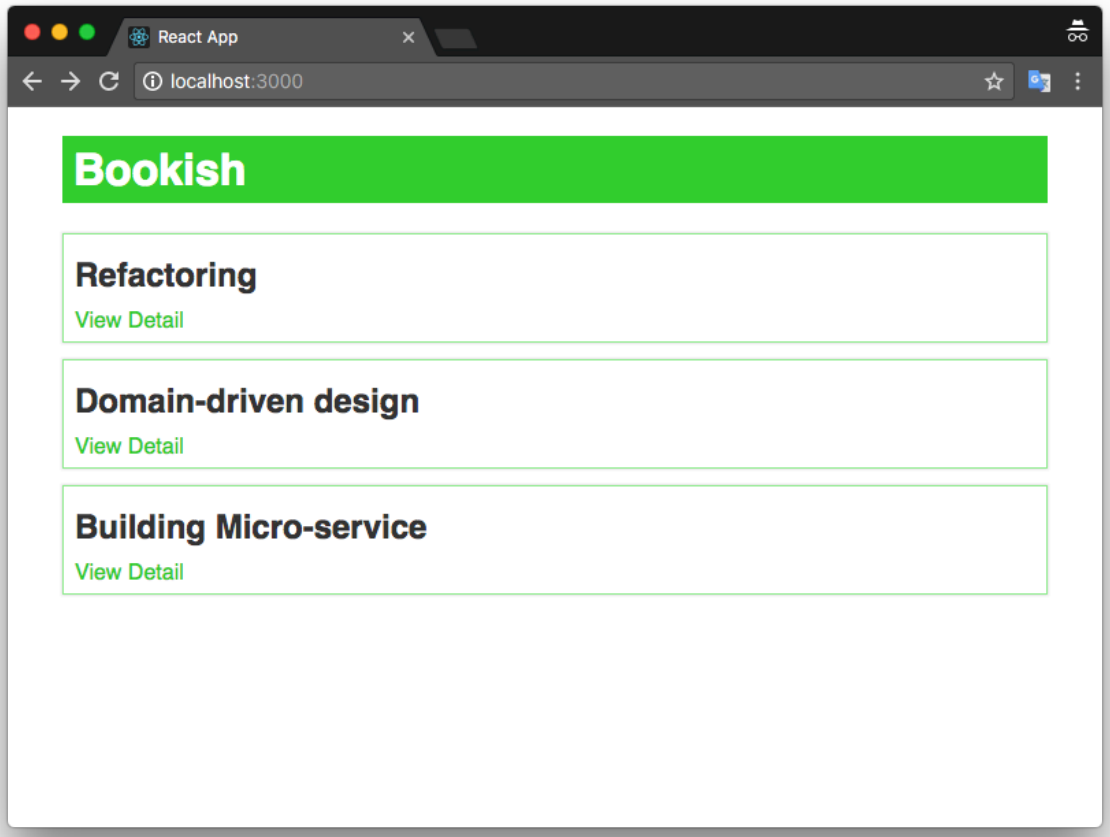
ugly

Let's define some layout related stuff in `App.css`:

```
.container {
    max-width: 720px;
    margin: 20px auto;
}

h1 {
    background: limegreen;
    color: white;
    padding: 6px 8px;
}
```

And then the book list related style in `BookList/index.css`

```
.book {
    border: 1px solid lightgreen;
    margin-bottom: 12px;
    padding: 6px 8px;
    box-shadow: 0 0 2px #ccc;
}

h2 {
    margin: 10px 0;
    color: #333;
}

.view-detail {
    color: limegreen;
    text-decoration: none;
}
```



better

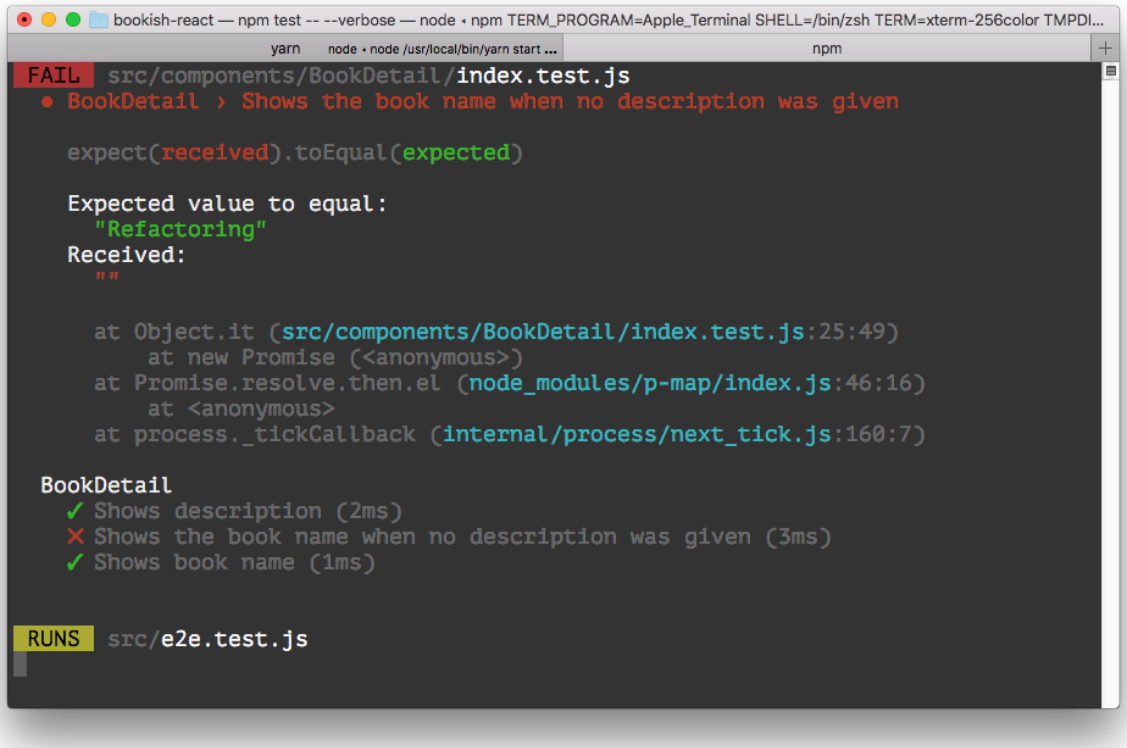
Handling default value

Well, there is a requirement adjustment: since the data provided by the backend service may contain some unexpected null values in some of the fields. For example, there is no guarantee that the `description` field is always presented (it may be just an empty string or null value), when in this case, we need to use the book name as the description as a fallback.

We can add a test to describe this case, note the props object doesn't contain `description` field at all:

```
1   it('Shows the book name when no description was given', () => {
2     const props = {
3       book: {
4         name: "Refactoring"
5       }
6     }
7     const wrapper = shallow(<BookDetail {...props}/>)
8     expect(wrapper.find('.description').text()).toEqual("Refactoring")
9   })
```

Then our test failed again:



We can fix that by a conditional operator:

```
function BookDetail({book}) {
  return (<div className="detail">
    <h2 className="name">{book.name}</h2>
    <div className="description">{book.description ? book.description : book.name}</\
div>
  </div>)
}
```

Yet another change

Soon there is a new change comes: if the length of `description` is greater than 300, we need to truncate the content to 300 characters and show a `show more...` link. When a user clicks the link, the full content will display.

We can add a new test for this case:

```
  it('Shows *more* link when description is too long', () => {
    const props = {
      book: {
        name: "Refactoring",
        description: "The book about how to do refactoring ...."
      }
    }
    const wrapper = shallow(<BookDetail {...props}/>)
    expect(wrapper.find('a.show-more').length).toEqual(1)
    expect(wrapper.find('.description').text()).toEqual("The book about how to do re\
factoring ....")
  })
```

Moreover, this drives us to write/modify the code to meet the requirement, and once all the tests pass, we can do the refactoring: extract new method, create new files, move method/class around, rename variables or change folder structures, and so on.

It is kind of an endless process, and there is always some space for us to improve and when we got enough time we can keep this process over and over again until we reach somewhere the code is clean and self-explaining.

Searching by keyword

Our third feature is that a user can search books by its name. It is handy when the book list become very long (it's hard for a user to find what he is looking for when content is more than one screen or one page).

Acceptance test

Similarly, we start by writing an `acceptance test`:

```
  test('Show books which name contains keyword', async () => {
    await page.goto(`${appUrlBase}/`)

    const input = await page.waitForSelector('input.search')
    page.type('input.search', 'design')

    // await page.screenshot({path: 'search-for-design.png'});
    await page.waitForSelector('.book .title')
    const books = await page.evaluate(() => {
      return [...document.querySelectorAll('.book .title')].map(el => el.innerText)
    })

    expect(books.length).toEqual(1)
    expect(books[0]).toEqual('Domain-driven design')
  })
```

We try to type keyword `design` into `.search` input box, and expect that only `Domain-driven design` shows up in the book list.

The simplest way to implement is just modifying the `BookListContainer` and add an `input` to it:

```
  render() {
    return (
      <div>
        <input type="text" className="search" placeholder="Type to search" />
        <BookList {...this.state}/>
      </div>
    )
  }
```

And then define a handling method for the `change` event for the `input` component:

```
  filterBook(e) {
    this.setState({
      term: e.target.value
    })

    axios.get(`http://localhost:8080/books?q=${e.target.value}`).then(res => {
      this.setState({
        books: res.data,
        loading: false
      })
    }).catch(err => {
      this.setState({
        loading: false,
        error: err
      })
    })
  }
```

and bind it on `input` component:

```
<input type="text" className="search" placeholder="Type to search" onChange={this.fi\
lterBook}
               value={this.state.term}/>
```

Note that we are using `books?q=${e.target.value}` as the URL to fetching data, that's a full-text searching API provided by `json-server`, you just need to send `books?q=domain` to the backend and it will return all the content that contains `domain`.

You can try it on the command line like this:

```
curl http://localhost:8080/books?q=domain
```

Now our tests green again. Let's jump to the next step of the `Red-Green-Refactoring`.

Refactoring

Obviously, the `filterBook` is almost the same as the code in `componentDidMount`, we can extract a function `fetchBooks` to remove the duplication:

```
  componentDidMount() {
    this.fetchBooks()
  }

  fetchBooks() {
    const {term} = this.state
    axios.get(`http://localhost:8080/books?q=${term}`).then(res => {
      this.setState({
        books: res.data,
        loading: false
      })
    }).catch(err => {
      this.setState({
        loading: false,
        error: err
      })
    })
  }

  filterBook(e) {
    this.setState({
      term: e.target.value
    }, this.fetchBooks)
  }
```

Emm, better than before. Since `fetchBooks` are coupling network request and `state` changing together, we can split them into define 2 functions:

```
  updateBooks(res) {
    this.setState({
      books: res.data,
      loading: false
    })
  }

  updateError(err) {
    this.setState({
      loading: false,
      error: err
    })
  }

  fetchBooks() {
```

```
16      const {term} = this.state
17      axios.get(`http://localhost:8080/books?q=${term}`).then(this.updateBooks).catch(\
18  this.updateError)
19    }
20
21    filterBook(e) {
22      this.setState({
23        term: e.target.value
24      }, this.fetchBooks)
25    }
```

Now the code turns much clean and comfortable to read.

One step further

Let's say, someone else may want to use the search box we just finished on his own page, how can we reuse it? It's tough because currently, the search box is highly tightly with the rest code in `BookListContainer`, we need to extract into another component `SearchBox`:

```
1  import React from 'react'
2
3  function SearchBox({term, onChange}) {
4    return (<input type="text" className="search" placeholder="Type to search" onChang\
5  e={onChange}
6                  value={term}/>)
7  }
8
9  export default SearchBox
```

After that extraction, the `render` method of `BookListContainer` turns:

```
1    render() {
2      return (
3        <div>
4          <SearchBox term={this.state.term} onChange={this.filterBook} />
5          <BookList {...this.state}/>
6        </div>
7      )
8    }
```

And for unit tests we can test it this way:

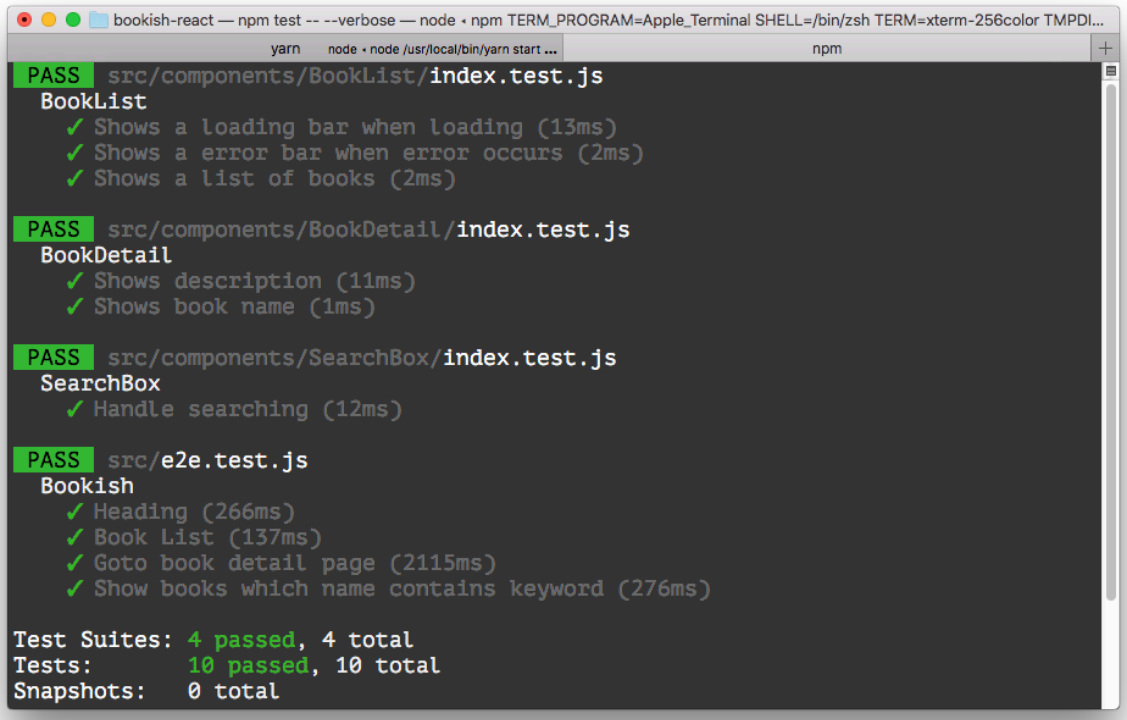
```
import React from 'react'
import {shallow} from 'enzyme'
import SearchBox from './SearchBox'

describe('SearchBox', () => {
  it('Handle searching', () => {
    const onChange = jest.fn()
    const props = {
      term: '',
      onChange
    }

    const wrapper = shallow(<SearchBox {...props}/>)
    expect(wrapper.find('input').length).toEqual(1)

    wrapper.simulate('change', 'domain')

    expect(onChange).toHaveBeenCalled()
    expect(onChange).toHaveBeenCalledWith('domain')
  })
})
```

Note that we are using `jest.fn()` to create a `spy` object that can record the trace of invocations. We use `simulate` API provided by `enzyme` to simulate a `change` event with `domain` as it's payload. We can then expect that `onChange` method has been called with data `domain`.

Now we noticed that `SearchBox` is just a presentational component, we can move it to `components` folder:

```
src
├── App.css
├── App.js
├── components
|   ├── BookDetail
|   |   ├── index.js
|   |   └── index.test.js
|   ├── BookList
|   |   ├── index.css
|   |   ├── index.js
|   |   └── index.test.js
|   └── SearchBox
|       ├── index.js
|       └── index.test.js
```

```
├── containers
│   ├── BookDetailContainer.js
│   └── BookListContainer.js
├── e2e.test.js
├── index.css
├── index.js
└── setupTests.js
```

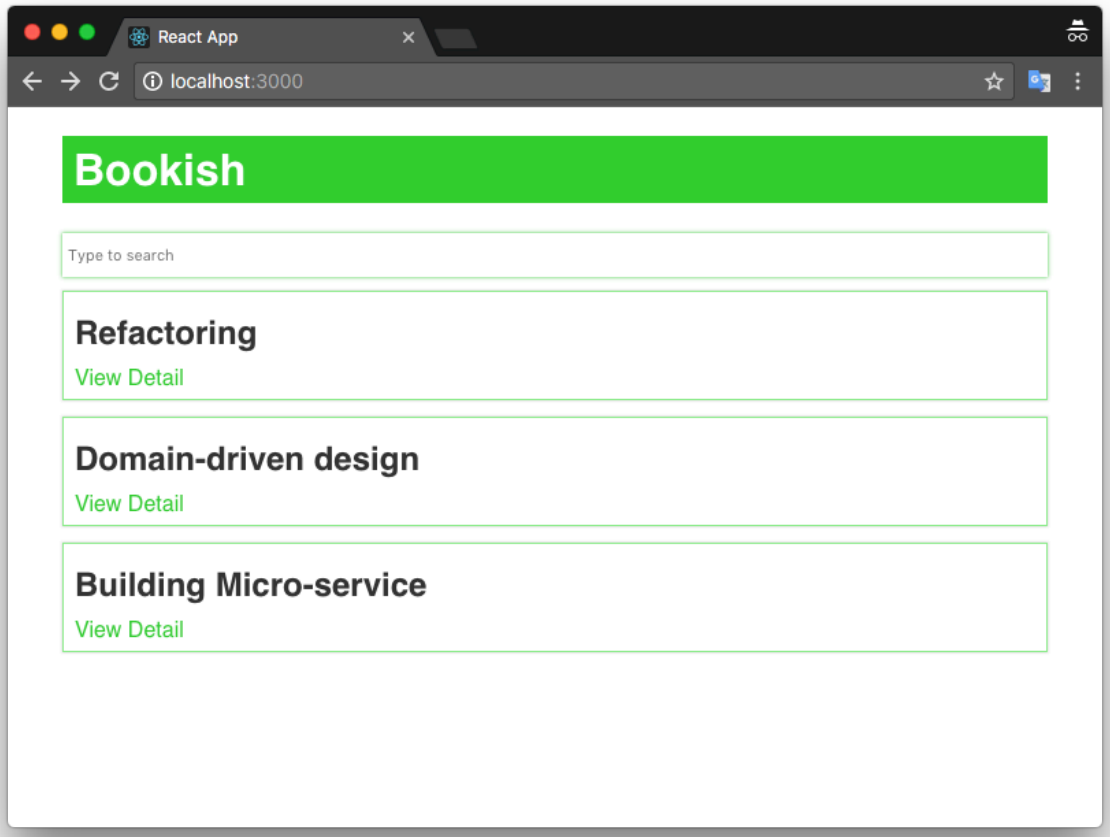


searching

Some style updates

```
.search {
    box-sizing: border-box;
    width: 100%;
    padding: 2px 4px;
    height: 32px;
}
```

Now our user interface looks quite like a real application:



Furthermore, let's restructure the `container` folder to make it consistency with the `component` folder:

```
src
├── App.css
├── App.js
├── components
|   ├── BookDetail
|   |   ├── index.js
|   |   └── index.test.js
|   ├── BookList
|   |   ├── index.css
|   |   ├── index.js
|   |   └── index.test.js
|   └── SearchBox
|       ├── index.css
|       ├── index.js
|       └── index.test.js
```

```
├── containers
│   ├── BookDetailContainer
│   │   └── index.js
│   └── BookListContainer
│       └── index.js
├── e2e.test.js
├── index.css
├── index.js
└── setupTests.js
```

We defined an `index.js` in each folder, and then you can simply import it by the folder name just like

```
import BookListContainer from "./containers/BookListContainer/"
```

without that, you may see some duplication in the path like this:

```
import BookListContainer from "./containers/BookListContainer/BookListContainer"
```

What we've done?

Great, we have finished all the 3 features! Let's have a quick review of what we've got here:

- 3 presentational components (BookDetail, BookList, SearchBox) and their unit tests
- 2 container components (BookDetailContainer, BookListContainer)
- 3 acceptance tests to cover the most valuable path (list, detail, and searching)

Moving forward

Maybe you have already noticed some code smell in our `end-to-end` tests, we mixed the DOM element selecting and content extracting code with the test assertions:

```
test('Heading', async () => {
  await page.goto(`${appUrlBase}/`)
  await page.waitForSelector('h1')
  const result = await page.evaluate(() => {
    return document.querySelector('h1').innerText
  })

  expect(result).toEqual('Bookish')
})
```

We can definitely improve this by introducing the `Page Object` pattern. In this pattern, we encapsulate the implementation details into a separate Object and expose some interfaces to access the element on the page, that would significantly improve the readability.

Page Object

Let's first create a new file named `BookListPage` with content as follows:

```
export default class BookListPage {

    constructor(page) {
        this.page = page;
    }

    async getHeading() {
      await this.page.waitForSelector('h1')
      const result = await this.page.evaluate(() => {
        return document.querySelector('h1').innerText
      })
      return result;
    }
}
```

In this class, we defined an `async` method named `getHeading`, and moved the code extracting `heading` text out of the test case. And in the test case, we can invoke this method like this:

```
  test('Heading', async () => {
    await page.goto(`${appUrlBase}/`)
    const lp = new BookListPage(page)
    const heading = await lp.getHeading()
    expect(heading).toEqual('Bookish');
  })
```

By this measure, we make the test code itself much more concise and easy to be understood.

Similarly, we can move the get book list parts into the `BookListPage` class like this:

```
async getBooks() {
    await this.page.waitForSelector('.books')
    const books = await this.page.evaluate(() => {
      return [...document.querySelectorAll('.book .title')].map(el => el.innerText)
    })
    return books;
}
```

The test case then would be simplified as:

```
  test('Book List', async () => {
    await page.goto(`${appUrlBase}/`)

    const listPage = new ListPage(page)
    const books = await listPage.getBooks();

    expect(books.length).toEqual(3)
    expect(books[0]).toEqual('Refactoring')
    expect(books[1]).toEqual('Domain-driven design')
    expect(books[2]).toEqual('Building Micro-service')
  })
```

We have hide the implementation details and separate the DOM-related logic and the test assertions. That makes the code easier to read and less disturbing the reader's (maybe ourselves) attention.

Restructure the folder for end-to-end tests

For the sake of clarity, we can create a new folder for `end-2-end` tests and `Page Objects`:

```
├── e2e
│   ├── e2e.test.js
│   └── pages
│       └── ListPage.js
```

Summary

In the above 3 chapters, we have developed 3 features of the application `Bookish`, we have learned how to apply ATDD in the real project. We have learned how to set up the `react` environment quickly, and how to use `mock-server` to launch the mock service.

We introduced `Puppeteer` to write `acceptance tests`, once we have the test, we write simple code to pass it, and refactor when there is code smell found in the code. During the whole process, we keep using the classic `Red-Green-Refactor` cycle. And we do the refactoring, we split the code based on their responsibility, and extract methods, rename classes, restructure the folders to make the code much more compact and easy to read and maintain.

Additionally, we have added some extension for `json-server`, and that enable us to prepare some data before running the test cases, and clean up after the test finished. That makes the test itself much more readable and independent.

Finally, we introduced the `Page Object` pattern to enhance the readability of our `end-to-end` tests.

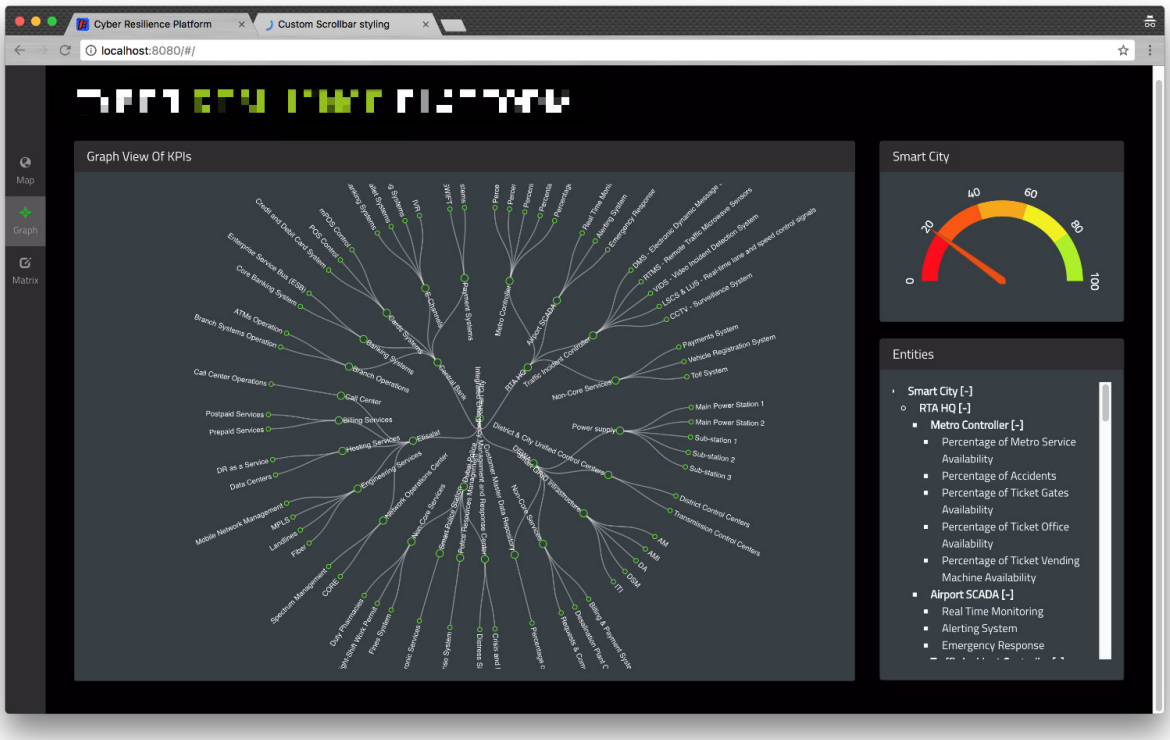
Introduce the state management

State management

During quite an extended period, frontend development is dealing with the state synchronization of different components(if any, in jQuery age, there is no real component actually, just DOM fragments). Keywords fulfilled in two search boxes on the page(one on the top and the other on the bottom), the active status of tabs, routing and the hash in URL, `show more...` link, and so on. All of those status management are making frontend developer crazy, even when MVVM library like `Backbone`, or two-way data binding `Angular` was invented, things still quite hard if you have to manage status among different components.

However, for now, web development is entirely different. In a typical web page, interaction and data translation become very complicated.

Let's take a look at this simple page:



state management

There is a tree component on the right-hand side and a graph component in the middle. Now when you click a node on the right-hand tree, then the node should be collapse/expand based on its previous status, and the status change should be applied to the graph as well.

If you don't use any external library, use DOM customize event may cause a `dead-loop`. So a more reasonable way is to extract the underlying data out, and use pub-sub pattern: tree and graph are all listening to the changes of the data, once the data changes, component should re-render itself.

Things like this are prevalent, you can find it on almost every web page. And if you implement you own pub-sub, it may be trivial and hard to maintain, fortunately, we are not alone.

Redux

`Redux` is a very popular JavaScript state management container, by using it, your application is straightforward to test, track the status, and debug. It's not bound with any libraries or frameworks, so you don't have to use it with `React`.

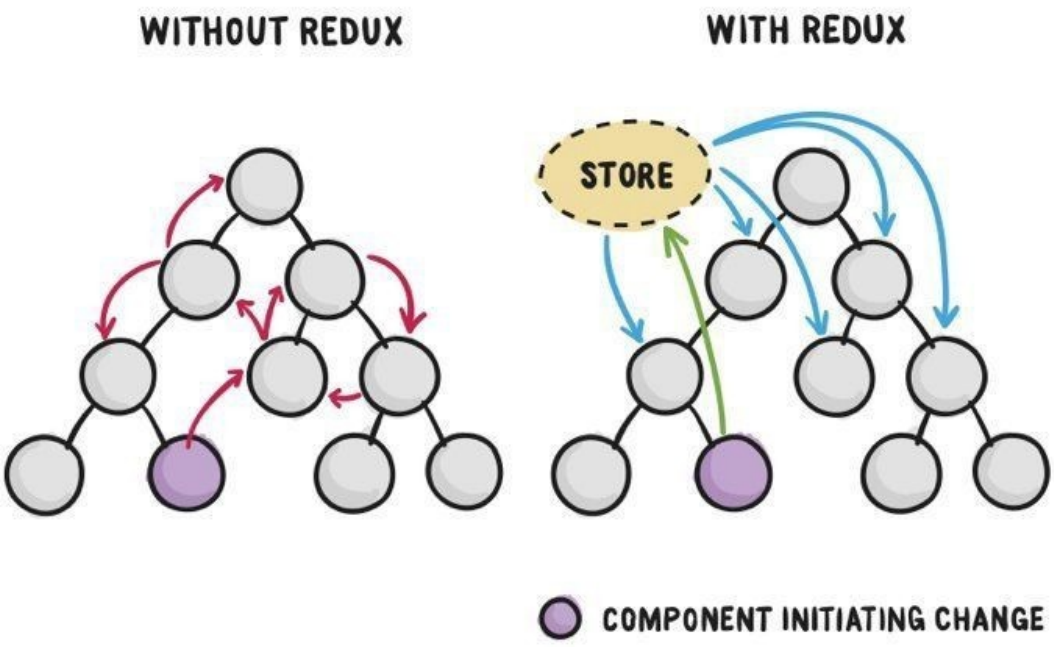
The `redux` documentation says that:

> Redux is a predictable state container for JavaScript apps.

Three Principles of Redux

- Single source of truth
- State is read-only
- Changes are made with pure functions

In the `Redux` world, all the state comes from a global data source, at any given time, this data source can be mapped to the view. When changes occur: user clicked a button, a timeout, backend async message arrived there will be an `action` created, and the `action` will go through some pure function called `reducer`, `reducer` will change the state, and that may trigger another re-render of the views.



with / without redux, source: https://kuanhsuh.github.io/2017/09/28/What-s-Redux-and-how-to-use-it/

Thanks for the excellent `virtual dom` mechanism provided by `React`, the UI will re-render in minimum effort.

Decouple

If you take a close look at our `container` code, you will find that it's actual response to two things:

```
class BookDetailContainer extends Component {
  constructor(props) {
    super(props)
    this.state = {
      book: {}
    }
  }

  componentDidMount() {
    const id = this.props.match.params.id
    axios.get(`http://localhost:8080/books/${id}`).then((res) => {
      this.setState({
        book: res.data
```

```
      })
    })
  }

  render() {
    return <BookDetail {...this.state}/>
  }
}
```

On the one hand, it has code to do the network related stuff. On the other hand, it manages the state that sub-component needs. For example, in `constructor`, `this.state.book` is empty, and when the network request is successful, `this.state.book` is set as the response. This actually couple our `container` tightly with network details.

Ideally, we can write `BookDetailContainer` like this:

```
class BookDetailContainer extends Component {
  componentDidMount() {
    const id = this.props.match.params.id
    fetchBookById(id)
  }

  render() {
    return <BookDetail {...this.props}/>
  }
}
```

and use it like this:

```
<BookDetailContainer {book: {}, fetchBookById: () => {}} />
```

`fetchBookById` could be either a synchronize function call or a synchronize remote call, but for `BookDetailContainer`, it doesn't matter.

That's why the state management container can help us, and the container can handle the details for us like listen to the changes, dispatch actions, reducer the state and broadcast changes.

view = f(state)

There is a very famous formula in `React` community: `view = f(state)`, means that `view` is just a map of `state`. `state` means a snapshot of our application state. Say, when a user opens `Bookish` home page, the snapshot at that time could be:

```
const state = {
    books: [
      {"name": "Refactoring", "id": 1, "description": "Refactoring"},
      {"name": "Domain-driven design", "id": 2, "description": "Domain-driven design\
"},
      {"name": "Building Micro-service", "id": 3, "description": "Building Micro-ser\
vice"}
    ],
    term: ''
}
```

And when user types `Domain` in the search box, then the snapshot becomes to:

```
const state = {
    books: [
      {"name": "Domain-driven design", "id": 2, "description": "Domain-driven design\
"}
    ],
    term: 'Domain'
}
```

That two piece of data (state) can at some point present the whole application. and since `view = f(state)`, for any given `state`, the `view` is always predictable. So the only thing the application developer cares about is how to manipulate the data, the `UI` will render automatically.

I know the idea may sound pretty straightforward, but until recently it comes true in the real world productions(the first release of `redux` is at Jun 2015, which is just less than 3 years ago)

References

- Redux motivation[7]
- Redux ☒ An Introduction[8]

State management

Environment set up

Firstly, we need to add some packages to use `redux`:

[7] https://cn.redux.js.org/docs/introduction/Motivation.html
[8] https://www.smashingmagazine.com/2016/06/an-introduction-to-redux/

```
yarn add redux redux-thunk history react-router-redux
```

Start from `action`

`action` is an excellent entry point, and it makes you think in the way that how the components interact with each other, and how the component interacts with the outside world.

Take `BookListContainer` for example, and we expect that it has the ability to set up the keyword for searching:

```
import {fetchBooks} from './actions'

describe('BookListContainer related actions', () => {
  it('Set search keyword', () => {
    const term = ''
    const expected = {
      type: 'SET_SEARCH_TERM',
      term
    }
    const action = setSearchTerm(term)
    expect(action).toEqual(expected)
  })
})
```

Of course, the action `fetchBooks` doesn't exist yet. Fortunately, it's effortless to implement one by using `redux`:

```
export const setSearchTerm = (term) => {
  return {type: 'FETCH_BOOKS', term}
}
```

That's just a piece of cake.

Async actions

However, things get a little trick for async actions. To make that work, we need to config `redux-thunk` and create a mock `store`.

Let's add the `redux-mock-store` to our dependencies first:

```
yarn add redux-mock-store
```

Before we write the tests, we need to config a `mockStore` in `action.test.js` like this:

```
import configureMockStore from 'redux-mock-store'
import thunk from 'redux-thunk'

const middlewares = [thunk]
const mockStore = configureMockStore(middlewares)
```

Then let's define the happy path that network is ok and we can get the data we are fetching:

```
  it('Fetch data successfully', () => {
    const books = [
      {id: 1, name: 'Refactoring'},
      {id: 2, name: 'Domain-driven design'}
    ]
    axios.get = jest.fn().mockImplementation(() => Promise.resolve({data: books}))

    const expectedActions = [
      { type: 'FETCH_BOOKS_PENDING'},
      { type: 'FETCH_BOOKS_SUCCESS', payload: books }
    ]
    const store = mockStore({ books: [] })

    return store.dispatch(fetchBooks('')).then(() => {
      expect(store.getActions()).toEqual(expectedActions)
    })
  })
```

Here we expect `fetchBooks` can create two `actions`: one to indicate that the request is sending, another to indicate the response comes. Since the request is using `axios` underline, we can simply use `jest.fn().mockImplementation()` to stub it.

It would interrupt the `axios.get` and call the `axios.get` we defined instead.

```
axios.get = jest.fn().mockImplementation(() => Promise.resolve({data: books}))
```

So we don't send a real HTTP request. Here is the implementation:

```
import axios from 'axios'

export const fetchBooks = () => {
  return (dispatch) => {
    dispatch({type: 'FETCH_BOOKS_PENDING'})
    return axios.get(`http://localhost:8080/books`).then((res) => {
      dispatch({type: 'FETCH_BOOKS_SUCCESS', payload: res.data})
    })
  }
}
```

Firstly, we `dispatch` a `FETCH_BOOKS_PENDING` action, and call `axios.get`, when the promise is resolved, we can `dispatch` the `FETCH_BOOKS_SUCCESS` action and with the response as payload.

The failure scenario

For the network failure(e.g. timeout), we need to cover that in unit test as well. By using `jest.fn().mockImplementation()`, we can simply say that:

```
axios.get = jest.fn().mockImplementation(() => Promise.reject({message: 'Something w\
ent wrong'}))
```

Moreover, we verify the failed action is dispatched correspondingly:

```
  it('Fetch data with error', () => {
    axios.get = jest.fn().mockImplementation(() => Promise.reject({message: 'Somethi\
ng went wrong'}))

    const expectedActions = [
      { type: 'FETCH_BOOKS_PENDING'},
      { type: 'FETCH_BOOKS_FAILED', err: 'Something went wrong' }
    ]
    const store = mockStore({ books: [] })

    return store.dispatch(fetchBooks('')).then(() => {
      expect(store.getActions()).toEqual(expectedActions)
    })
  })
```

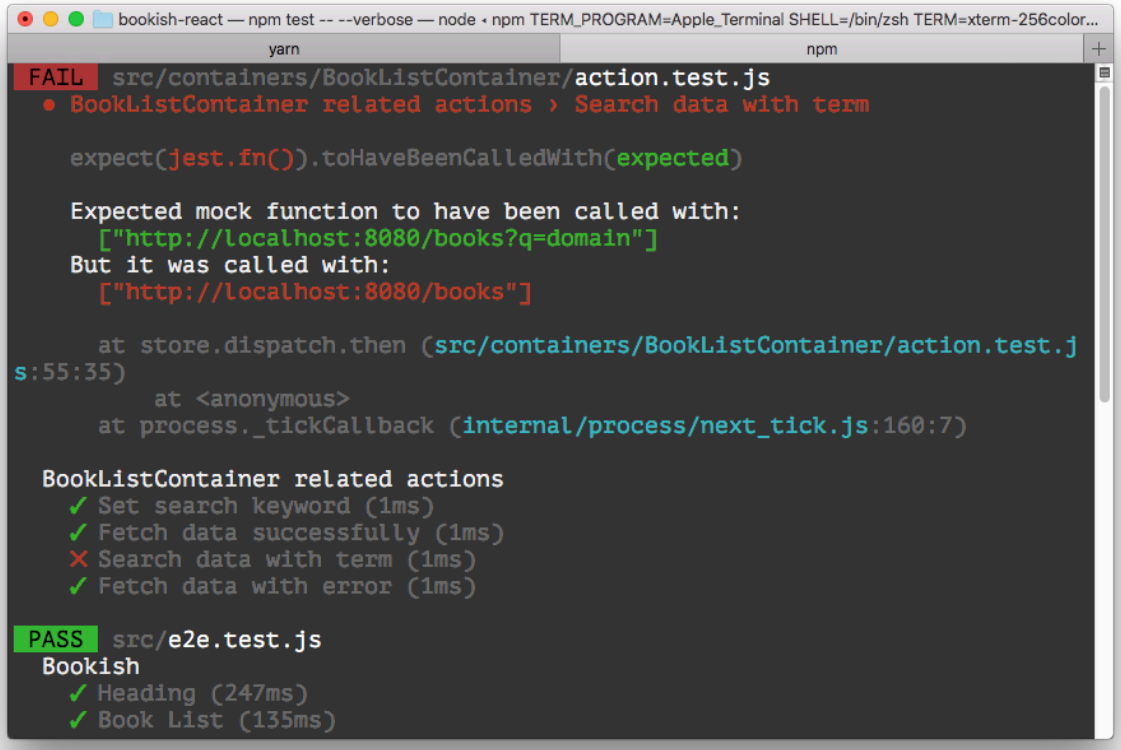
We can add the `catch` case in promise `rejected` branch to make our test green:

```
export const fetchBooks = (term) => {
  return (dispatch) => {
    dispatch({type: 'FETCH_BOOKS_PENDING'})
    return axios.get(`http://localhost:8080/books?q=${term}`).then((res) => {
      dispatch({type: 'FETCH_BOOKS_SUCCESS', payload: res.data})
    }).catch((err) => {
      dispatch({type: 'FETCH_BOOKS_FAILED', err: err.message})
    })
  }
}
```

Searching action

We are expecting that the action `fetchBooks` can use `term` value in `store` as the keyword when sending the request, which enable the filter functionality. Note that we are setting the `term` to `domain` in the `mockStore`:

```
  it('Search data with term in state', () => {
    const books = [
      {id: 1, name: 'Refactoring'},
      {id: 2, name: 'Domain-driven design'}
    ]
    axios.get = jest.fn().mockImplementation(() => Promise.resolve({data: books}))

    const expectedActions = [
      { type: 'FETCH_BOOKS_PENDING'},
      { type: 'FETCH_BOOKS_SUCCESS', payload: books }
    ]
    const store = mockStore({ books: [], term: 'domain' })

    return store.dispatch(fetchBooks('')).then(() => {
      expect(axios.get).toHaveBeenCalledWith('http://localhost:8080/books?q=domain')
    })
  })
```



search

Actually, we can read `term` from `store` in code, and attach it to `query string` when sending the HTTP request.

```
 export const fetchBooks = () => {
-  return (dispatch) => {
+  return (dispatch, getState) => {
     dispatch({type: 'FETCH_BOOKS_PENDING'})
-    return axios.get(`http://localhost:8080/books`).then((res) => {
+    const state = getState()
+    return axios.get(`http://localhost:8080/books?q=${state.term}`).then((res) => {
       dispatch({type: 'FETCH_BOOKS_SUCCESS', payload: res.data})
     }).catch((err) => {
       dispatch({type: 'FETCH_BOOKS_FAILED', err: err.message})
```

This code looks a little bit tricky, the arguments `dispatch` and `getState` actually set by `redux-thunk` middleware when the `fetchBooks` is invoked.

Refactor

There are a lot of hardcode and magic strings in the `action` test and implementation, and we can extract them to some commonplace so they can be referenced from there. Let's create a file `types.js`:

```
export const SET_SEARCH_TERM = 'SET_SEARCH_TERM'
export const FETCH_BOOKS_PENDING = 'FETCH_BOOKS_PENDING'
export const FETCH_BOOKS_SUCCESS = 'FETCH_BOOKS_SUCCESS'
export const FETCH_BOOKS_FAILED = 'FETCH_BOOKS_FAILED'
```

And import it as a variable `types` when we want to use it, say, in unit tests:

```
import * as types from './types'
```

Then we can use `types.FETCH_BOOKS_PENDING` to reference it:

```
const expectedActions = [
  { type: types.FETCH_BOOKS_PENDING},
  { type: types.FETCH_BOOKS_SUCCESS, payload: books }
]
```

Reducer

`reducer` in `redux` is just a pure function. If the input is certain, the output always predictable. So that's pretty easy to test the reducer:

```
import reducer from './reducer'
import * as types from './types'

describe('Reducer', () => {
  it('Set the search keyword', () => {
    const initState = { term: '' }
    const action = {type: types.SET_SEARCH_TERM, term: 'domain'}

    const state = reducer(initState, action)

    expect(state.term).toEqual('domain')
  })
})
```

As you can see here, we expect the `reducer` that when receiving an action, it returns a new `state`, and the `term` of the new state should be just the value in the `action`.

```
import * as types from './types'

const initialState = {
  term: ''
}

export default (state = initialState, action) => {
  switch (action.type) {
    case types.SET_SEARCH_TERM:
      return {
        ...state,
        term: action.term
      }
    default:
      return state
  }
}
```

And it's pretty straightforward to implement it. Similarly, `FETCH_BOOKS_PENDING` and `FETCH_BOOK_SUCCESS` can be tested like this:

```
  it('Show loading when request is sent', () => {
    const initState = { loading: false }

    const action = {type: types.FETCH_BOOKS_PENDING}
    const state = reducer(initState, action)

    expect(state.loading).toBeTruthy()
  })
```

Testing action-creator just like Value-Object testing in Java/.Net, and testing `reducer` is like testing the static `util` classes. It's pretty easy and straightforward. Actually, in the `React` community, people tend to test `action+reducer+store` all together instead.

We'll talk about it in the next section.

Integration test for store

```
import axios from 'axios'

import * as actions from './containers/actions'
import store from './store'

describe('Store', () => {
  const books = [
    {id: 1, name: 'Refactoring'}
  ]

  it('Loading books from remote', () => {
    axios.get = jest.fn().mockImplementation(() => Promise.resolve({data: books}))

    return store.dispatch(actions.fetchBooks()).then(() => {
      const state = store.getState()
      expect(state.list.books.length).toEqual(1)
      expect(state.list.books).toEqual(books)
    })
  })
})
```

We imported the `actions` defined before, and then using a real `store` to do the `dispatch`, and expect it returns the `correct` response. We import the real `reducers` and create a store by using `createStore` provided by `redux`:

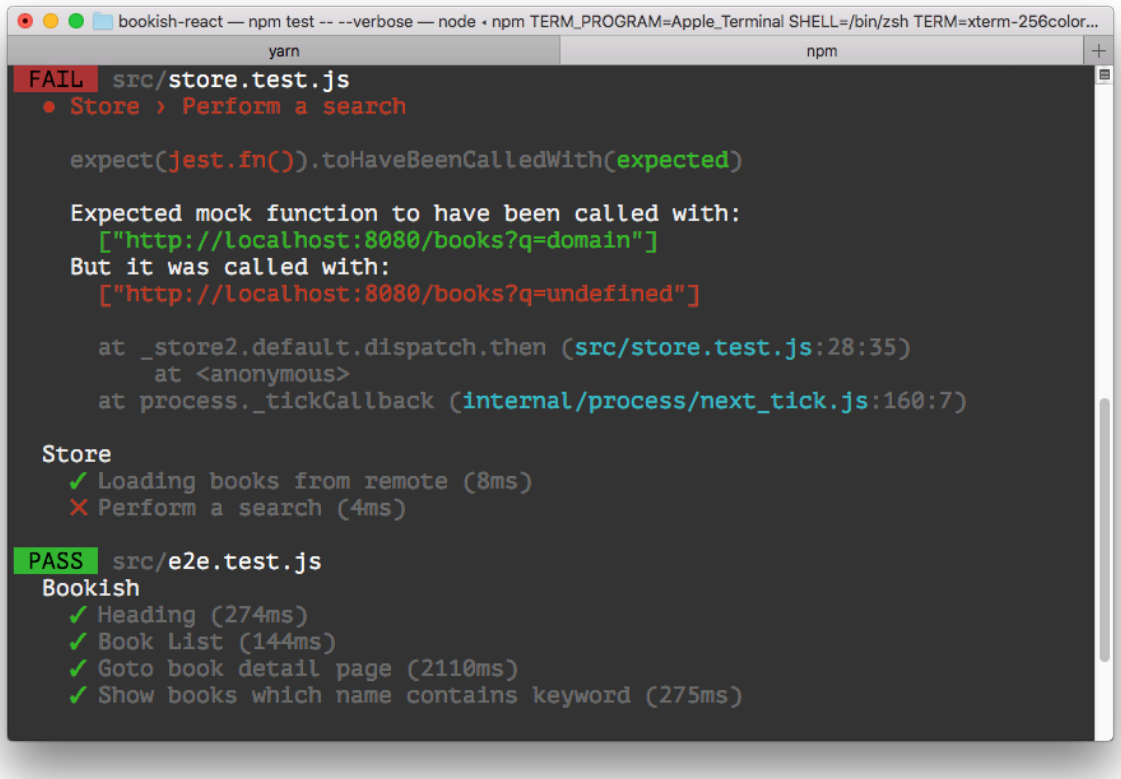
```
import list from './containers/reducer'

const rootReducer = combineReducers({
  list
})

// ...
const store = createStore(
  rootReducer,
  initialState,
  composedEnhancers
)

export default store
```

That's an integration test, it connects `action + reducer + store` altogether. This kind of test is a little heavy than the unit test, but it provides its unique value: things can work together perfectly.

Here is another example of `searching`

```
1   it('Perform a search', () => {
2     axios.get = jest.fn().mockImplementation(() => Promise.resolve({data: books}))
3
4     store.dispatch(actions.setSearchTerm('domain'))
5     return store.dispatch(actions.fetchBooks()).then(() => {
6       const state = store.getState()
7       expect(state.list.term).toEqual('domain')
8       expect(axios.get).toHaveBeenCalledWith('http://localhost:8080/books?q=domain')
9     })
10  })
```

In the scenario, we send two `actions` to `store`: `setSearchTerm` and `fetchBooks`. Since we are exporting `reducer` as a `list` in `store.js`, so the test fails:



We need to fix that in `reducer.js` like this:

```
    return (dispatch, getState) => {
      dispatch({type: types.FETCH_BOOKS_PENDING})
      const state = getState()
-     return axios.get(`http://localhost:8080/books?q=${state.term}`).then((res) => {
+     return axios.get(`http://localhost:8080/books?q=${state.list.term}`).then((res)\
 => {
        dispatch({type: types.FETCH_BOOKS_SUCCESS, payload: res.data})
      }).catch((err) => {
        dispatch({type: types.FETCH_BOOKS_FAILED, err: err.message})
```

Then we will got a lot of unit test failure, that's because the `mockStore` we were using in unit test are without `list` prefix, let's fix them all by adding a `list` wrapper:

```
-       const store = mockStore({ books: [], term: 'domain' })
+       const store = mockStore({list: { books: [], term: 'domain' }})
```

Migration

Our next step is to migrate our application to `redux`. Since we have already got enough acceptance tests there, we don't need to worry about break the functionality.

Firstly, we need to add `react-redux` as a dependency:

```
yarn add react-redux
```

And we pass the `store` to a `Provider` component in `index.js`, and then the whole component tree can share this `store` at any time:

```
+import { Provider } from 'react-redux'
+import store from './store'

-ReactDOM.render(<Router>
-  <App />
-</Router>, document.getElementById('root'));
+const root = <Provider store={store}>
+  <Router>
+    <App />
+  </Router>
+</Provider>
+
+ReactDOM.render(root, document.getElementById('root'));
```

Since the `presentational` components are just state-less, our migration will only impact the `container` component. For `BookListContainer`, there will be many changes, and data fetching is delegated to `actions`:

```
   filterBook(e) {
-    this.setState({
-      term: e.target.value
-    }, this.fetchBooks)
+    this.props.setSearchTerm(e.target.value)
+    this.props.fetchBooks()
   }
```

By using `react-redux` we can map the `state` in global `store` to component's props:

```
-export default BookListContainer

+const mapStateToProps = state => ({
+  loading: state.list.loading,
+  books: state.list.books,
+  error: state.list.error,
+  term: state.list.term
+})
+
+const mapDispatchToProps = dispatch => bindActionCreators({
+  setSearchTerm,
+  fetchBooks
+}, dispatch)
+
+export default connect(mapStateToProps, mapDispatchToProps)(BookListContainer)
```

So the final version of `BookListContainer` looks like this:

```
class BookListContainer extends Component {
  constructor(props) {
    super(props)
    this.filterBook = this.filterBook.bind(this)
  }

  componentDidMount() {
    this.props.fetchBooks()
  }

```

```
  filterBook(e) {
    this.props.setSearchTerm(e.target.value)
    this.props.fetchBooks()
  }

  render() {
    return (
      <div>
        <SearchBox term={this.props.term} onChange={this.filterBook} />
        <BookList {...this.props}/>
      </div>
    )
  }
}
```

Test the `container`

To test the `BookListContainer`, we need to export it as a component. We don't care about the default exported class `connect(mapStateToProps, mapDispatchToProps)(BookListContainer)`, as long as the `BookListContainer` works properly, then the connected version will be just fine:

```
-class BookListContainer extends Component {
+export class BookListContainer extends Component {
```

Testing `BookListContainer` is pretty easy, just set up the `props`, and verify the structure:

```
describe('BookListContainer', () => {
  it('render', () => {
    const props = {
      loading: false,
      books: [],
      fetchBooks: jest.fn()
    }

    const wrapper = shallow(<BookListContainer {...props} />)
    expect(wrapper.find('SearchBox').length).toEqual(1)
    expect(wrapper.find('BookList').length).toEqual(1)
  })
})
```

To test event handling, you can use `jest.fn()` and then verify the behaviors:

```
  it('invoke correct actions', () => {
    const props = {
      loading: false,
      books: [],
      fetchBooks: jest.fn(),
      setSearchTerm: jest.fn()
    }

    const wrapper = shallow(<BookListContainer {...props} />)

    wrapper.find('SearchBox').simulate('change', {target: {value: 'domain'}})
    expect(props.setSearchTerm).toHaveBeenCalledWith('domain')
    expect(props.fetchBooks).toHaveBeenCalled()
  })
```

We've already verified that function in `actions.js` works well, here we can make sure the correct action is invoked.

Fetch Book Detail

```
it('Fetch book by id', () => {
  const book = {id: 1, name: 'Refactoring'}
  axios.get = jest.fn().mockImplementation(() => Promise.resolve({data: book}))

  const store = mockStore({list: { books: [], term: '' }})

  return store.dispatch(fetchABook(1)).then(() => {
    expect(axios.get).toHaveBeenCalledWith('http://localhost:8080/books/1')
  })
})
```

We can copy `fetchBooks` and modify a little bit into `fetchABook`, it needs an `id` parameter to send the request:

```
export const fetchABook = (id) => {
  return (dispatch) => {
    dispatch({type: types.FETCH_BOOK_PENDING})
    return axios.get(`http://localhost:8080/books/${id}`).then((res) => {
      dispatch({type: types.FETCH_BOOK_SUCCESS, payload: res.data})
    }).catch((err) => {
      dispatch({type: types.FETCH_BOOK_FAILED, err: err.message})
    })
  }
}
```

So the integration test in `store` is similar correspondingly:

```
  it('Fetch a book from remote', () => {
    axios.get = jest.fn().mockImplementation(() => Promise.resolve({data: books[0]}))

    return store.dispatch(actions.fetchABook(1)).then(() => {
      const state = store.getState()
      expect(state.list.current).toEqual(books[0])
    })
  })
```

We can use `state.list.current` for the content of `detail page`, and we add that field to `initialState` in `reducer.js`:

```
    books: [],
-   error: ''
+   error: '',
+   current: {}
 }

 export default (state = initialState, action) => {
@@ -31,6 +32,11 @@ export default (state = initialState, action) => {
         loading: false,
         error: action.err
       }
+    case types.FETCH_BOOK_SUCCESS:
+      return {
+        ...state,
+        current: action.payload
+      }
```

And `BookDetailContainer` could be simplified as:

```
class BookDetailContainer extends Component {
  componentDidMount() {
    const id = this.props.match.params.id
    this.props.fetchABook(id)
  }

  render() {
    return <BookDetail {...this.props}/>
  }
}
```

We will use `state.list.current` as `book` prop in `mapStateToProps`, so the global store can then be connected it.

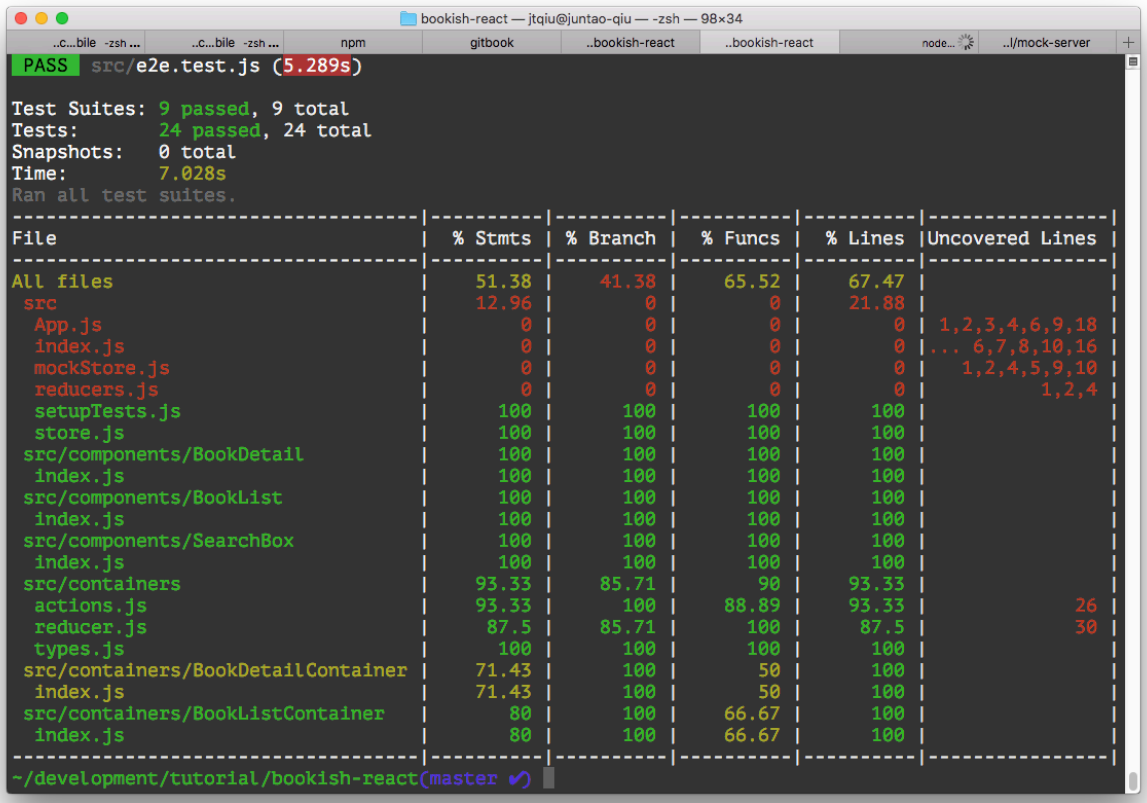
```
const mapStateToProps = state => ({
  book: state.list.current
})

const mapDispatchToProps = dispatch => bindActionCreators({
  fetchABook
}, dispatch)

export default connect(mapStateToProps, mapDispatchToProps)(BookDetailContainer)
```

Since this part of the code is verified by `redux`, we don't have to test them at all. Our job is to make sure `BookDetailContainer` works properly:

```
describe('BookDetailContainer', () => {
  it('render', () => {
    const bookId = 1
    const props = {
      match: {
        params: {
          id: bookId
        }
      },
      fetchABook: jest.fn()
    }

    const wrapper = shallow(<BookDetailContainer {...props} />)
    expect(wrapper.find('BookDetail').length).toEqual(1)
    expect(props.fetchABook).toHaveBeenCalledWith(bookId)
```

```
16    })
17  })
```

And we have successfully migrated to `redux`, and the test coverages look like this:



coverage

Summary

Take a look at what we've done here:

- Unit tests for actions, reducers
- Integration tests for `action + reducer + store`
- Acceptance tests remain green

That's a really great achievement for us.

Summary

We introduced `redux` as the state management mechanism in this chapter. By unit testing the `action`, `reducer` we've driven out the necessary `redux` components for our application. After some refactoring, we've migrated our `container` code to `redux`.

After the migration, we found that `container` is very easy to test, so we `added` some unit test for the `container` component.

This `test-last` seems a little bit weird, but if you treat it as part of `refactoring` then that's just fine. Actually, when dealing with the legacy code we always face the similar problem: sometimes the code is just too hard to test, you may need to do many changes to write test. In this case, we can just write a high-level test(acceptance test) to make sure the business requirements are always meet, then we can use `refactoring` to split the current implementation, and after that, we add proper unit tests.

Those unit tests not only verify the functionality but also act as documentation, and other team members can easily understand how to use your component by taking a look at your tests.

The Reviews of a book

In any real-world projects, you are most likely have to deal with some type of resource management. An advertising management system manages `schedule` or `campaign` by create, modify, or delete it under some business restriction. While an HR assistant system would help HR to manage `employee` by creating (when the company has new hires), modify (someone who standout being promoted), delete (retire). If you look at the problem those systems are trying to resolve, you could find a similar pattern: they're all applying `CURD`(Creation, Retrieve, Update, Deletion) operations on some resources.

However, not all the system have to involve all those 4 types of operations, for some critic system, no data deleted, the programmer just set some flag in the record to mark them as deleted - the records are still there, but the user cannot retrieve them from the GUI anymore.

In this chapter, we'll learn how to implement a classic set of `CURD` operations on resource `review` by extending our application `bookish`, with `ATDD` applying of course.

Business requirements

In the book detail page, there is some basic information about the book like title, description, cover image, and so on. However, we want something shows that can help the end user to know better about the book, like `review` s. Generally speaking, a book can have more than one `reviews`, those reviews are all from the reader who has a strong opinion about the book. It could be just an expression of stratification, in both positive and negative ways. Sometimes, there is also a rating with the review, for the sake of simplicity, we take some necessary information here.

Empty list first

```
describe('ReviewList', () => {

  it('Empty list', () => {
    const props = {
      reviews: []
    }
    const wrapper = shallow(<ReviewList {...props}/>)
    expect(wrapper.find('.reviews-container').length).toBe(1);
  })

})
```

Let's start with the simplest scenario when there is no data provided at all, we need to render an empty container `reviews-container`. Make the test pass should be straightforward:

```
import React, { Component } from 'react';

class ReviewList extends  Component {
  render () {
    return (<div className="reviews-container">
    </div>)
  }
}

export default ReviewList
```

Alternatively, we can simplify the component as a pure presentational component:

```
import React from 'react';

export const ReviewList = () => {
  return (<div className="reviews-container" />)
}
```

With a static list

Our second test case should involve some mock data:

```
  it('Render List', () => {
    const props = {
      reviews: [
        { name: 'Juntao', date: '2018/06/21', content: 'Excellent work, really impre\
ssive on the efforts you put'},
        { name: 'Abruzzi', date: '2018/06/22', content: 'What a great book'}
      ]
    }

    const wrapper = shallow(<ReviewList {...props}/>)
    expect(wrapper.find('.reviews-container').length).toBe(1);
    expect(wrapper.find('.review').length).toBe(2)
  })
```

Note that we demonstrate that how to use the component from outside (pass a reviews array, with each, has particular fields like `name`, `date` and `content`). It would be easy for other programmers to reuse our component without a look into our implementation.

So a `map` should do the work for us:

```
class ReviewList extends  Component {
  render () {
    const { reviews } = this.props
    return (<div className="reviews-container">
      {reviews.map(review => <div className="review"></div>)}
    </div>)
  }
}
```

And we need to make sure the content is rendered correctly:

```
+
+    const firstReview = wrapper.find('.review').at(0);
+    expect(firstReview.text()).toEqual('Excellent work, really impressive on the ef\
forts you put');
```

Since the map requires a unique identity for the `key` attribute, let's simply combine the `name` and `date` as a key, we should have done that with some `id` in the coming section when we integrate with the backend API.

```
-      {reviews.map(review => <div className="review"></div>)}
+      {reviews.map(review => <div className="review" key={`${review.name}-${review.\
date}`}>{review.content}</div>)}
```

Use the Review component

Now, let's put the `ReviewList` in `BookDetail` for our first integration. I bet you know the workflow! Right, test first. So we can add a new test case in `BookDetail.test.js` since we want to verify if the BookDetail has a `ReviewList` on it.

```
  it('Shows ReviewList', () => {
    const props = {
      book: {
        name: "Refactoring",
        description: "The book about how to do refactoring",
        reviews: []
      }
    }
    const wrapper = shallow(<BookDetail {...props}/>)
    expect(wrapper.find(ReviewList).length).toEqual(1)
  })
```

Note the `props` here which contains a `reviews` attribute. Great, and for the implementation, we introduce the `ReviewList` component and thanks to the componentization that's it:

```
import ReviewList from "./ReviewList/index";
```

```
function BookDetail({book}) {
  return (<div className="detail">
    <h2 className="name">{book.name}</h2>
    <div className="description">{book.description ? book.description : book.name}</\
div>
    {book.reviews && <ReviewList reviews={book.reviews} />}
  </div>)
}
```

Fulfill a book review form

⌧⌧⌧

Although we can make some static data to display on the BookDetail, it would be better if we can show some real data from the end user. So we need a simple form for the user to fulfill what's their point of view about the book. To put it simply, for now, we can provide 2 input box and a submit button. The first input is used for the user's name (or email address), and the second one (a textarea) is used for the review content.

We can add a new test case in `BookDetail` component:

```
  it('Shows Review Form', () => {
    const props = {
      book: {
        name: "Refactoring"
      }
    }
    const wrapper = shallow(<BookDetail {...props}/>)
    expect(wrapper.find('form').length).toEqual(1)
    expect(wrapper.find('form input[name="name"]').length).toEqual(1)
    expect(wrapper.find('form textarea[name="content"]').length).toEqual(1)
    expect(wrapper.find('form button[name="submit"]').length).toEqual(1)
  })
```

So make sure it shows under the description section and above `reviews`:

```
      <div className="description">{book.description ? book.description : book.name}<\
/div>
+     <form>
+       <input type="text" name="name"/>
+       <textarea name="content" cols="30" rows="10"></textarea>
+       <button name="submit">Submit</button>
+     </form>
      {book.reviews && <ReviewList reviews={book.reviews} />}
```

However, the form element is uneditable, and we have to connect them with `state`:

```
class BookDetail extends Component {
  constructor(props) {
    super(props)
    this.state = {
      name: '',
      content: ''
    }
  }

  updateName = (e) => {
    this.setState({name: e.target.value})
  }

  updateContent = (e) => {
    this.setState({content: e.target.value})
  }
```

```
17
18    saveReview = (e) => {
19      e.preventDefault()
20    }
21
22    render() {
23      const {book} = this.props
24
25      return (<div className="detail">
26        <h2 className="name">{book.name}</h2>
27        <div className="description">{book.description ? book.description : book.name}\
28  </div>
29        <form>
30          <input type="text" name="name" value={this.state.name} onChange={this.update\
31  Name} />
32          <textarea name="content" cols="30" rows="10" value={this.state.content} onCh\
33  ange={this.updateContent} />
34          <button name="submit" onClick={this.saveReview}>Submit</button>
35        </form>
36        {book.reviews && <ReviewList reviews={book.reviews} />}
37      </div>)
38    }
39  }
```

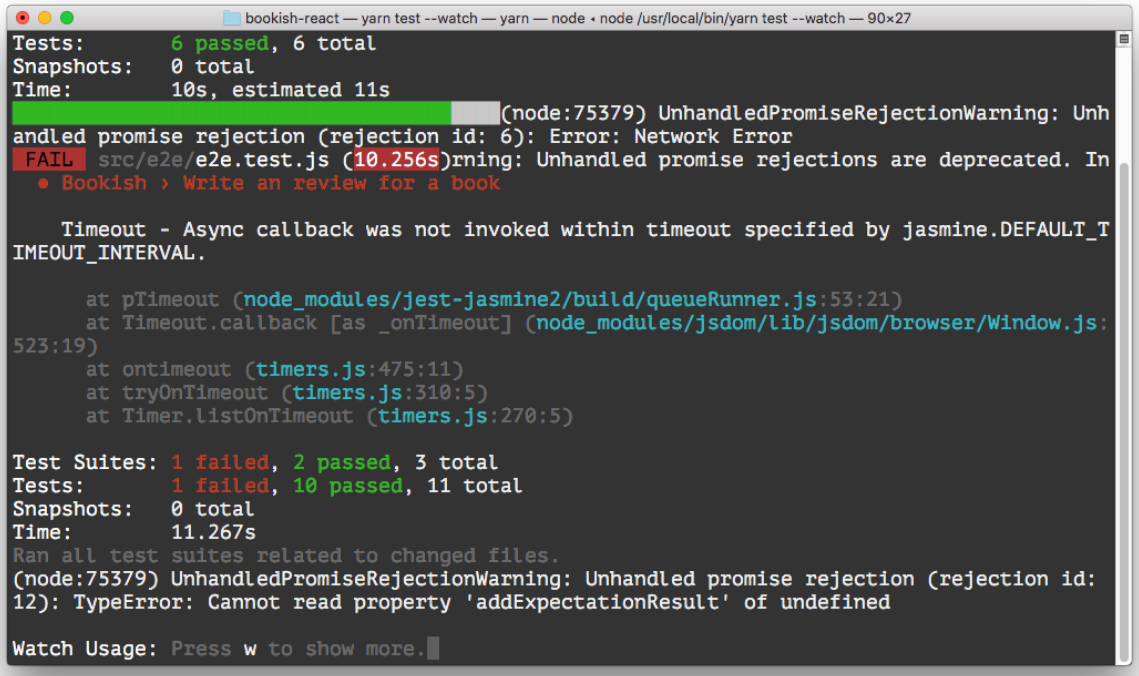
End to end test

You may have already noticed that when we approach this function, we started from the unit test of the ReviewList component. That is because of all the changes, for now, are static, there are no behavior interactions for now. In this case, you can either start from end-to-end test – from top to bottom – or from bottom to top. I prefer to start with the component itself because it provides feedback more rapidly, thus more interesting.

The end-to-end test can be described like this: goto detail page, find the input, fulfill some content, and click the submit button. Finally, we would expect the content submitted can be displayed on the page:

```
  test('Write an review for a book', async () => {
    await page.goto(`${appUrlBase}/`)
    await page.waitForSelector('a.view-detail')

    const links = await page.evaluate(() => {
      return [...document.querySelectorAll('a.view-detail')].map(el => el.getAttribu\
te('href'))
    })

    await Promise.all([
      page.waitForNavigation({waitUntil: 'networkidle2'}),
      page.goto(`${appUrlBase}${links[0]}`)
    ])

    const url = await page.evaluate('location.href')
    expect(url).toEqual(`${appUrlBase}/books/1`)

    await page.waitForSelector('.description')
    const result = await page.evaluate(() => {
      return document.querySelector('.description').innerText
    })
    expect(result).toEqual('Refactoring')

    await page.waitForSelector('input[name="name"]')
    page.type('input[name="name"]', 'Juntao Qiu')

    await page.waitForSelector('textarea[name="content"]')
    page.type('textarea[name="content"]', 'Excellent works!')

    const button = await page.waitForSelector('button[name="submit"]')
    page.click('button[name="submit"]');

    await page.waitForSelector('.reviews-container')
    const reviews = await page.evaluate(() => {
      return [...document.querySelectorAll('.review')].map(el => el.innerText)
    })

    expect(reviews.length).toEqual(1)
    expect(reviews[0]).toEqual('Excellent works!');
  })
```

And the test will fail miserably after the click since it neither send the data to server nor get response and re-rendering.



To make the test pass, we need to go back to `redux` and try to define some new type of `actions`:

action in redux

We have learned that all the network activity and other chores are handled by `actions` in `redux`. So let's firstly define an action to create a `review`:

```
1     it('Save a review for a book', () => {
2       const review = {
3         name: 'Juntao Qiu',
4         content: 'Excellent work!'
5       }
6       axios.post = jest.fn().mockImplementation(() => Promise.resolve({}))
7
8       const store = mockStore({list: { books: [], term: '' }})
9
10      return store.dispatch(saveReview(1, review)).then(() => {
11        expect(axios.post).toHaveBeenCalledWith('http://localhost:8080/books/1', rev\
12  iew)
13      })
14    })
```

Let's presume that when we `POST` some data to backend endpoint `http://localhost:8080/books/1`, there will be a new review created for the book with id `1`:

```
{
  "name": "Juntao Qiu",
  "content": "Excellent work!"
}
```

Create an async action by using `axios` should be very easy for us now:

```
export const saveReview = (id, review) => {
  return (dispatch) => {
    dispatch({type: types.SAVE_BOOK_REVIEW_PENDING})
    return axios.post(`http://localhost:8080/books/${id}`, review).then((res) => {
      dispatch({type: types.SAVE_BOOK_REVIEW_SUCCESS, payload: res.data})
    }).catch((err) => {
      dispatch({type: types.SAVE_BOOK_REVIEW_FAILED, err: err.message})
    })
  }
}
```

So we then add `onSubmit` event handler in `BookDetail` component:

```
  saveReview = (e) => {
    e.preventDefault()
    const id = this.props.book.id
    this.props.saveReview(id, {
      name: this.state.name,
      content: this.state.content
    })
  }
```

the `saveReview` callback could be passed from parent component `BookDetailContainer` directly.

json-server customization

We've been using `json-server` to simplify the backend API work for us. Now we need to do a little bit customization on that again to fit our requirement. We expect that `review` is a sub-resource to `book`, and that allows us to access all the reviews belong to a particular book by saying `/books/1/reviews`.

Additionally, we would like `/books/1` to carry all `reviews` as an embedded resource in the response, that can make the rendering of the book detail page very easy and convenient. To make that work, we need to define a `route` in `json-server` like this:

```
server.use(jsonServer.rewriter({
  '/books/:id':  '/books/:id?_embed=reviews'
}))

server.use(router)
```

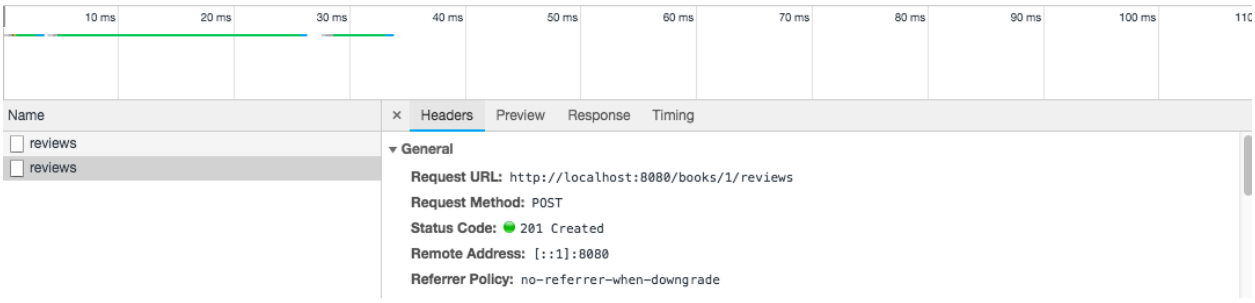
Then whenever you access `/books/1`, it returns all the reviews along with the response. A request like this:

```
curl http://localhost:8080/books/1
```

would get the response like:

```
{
  "name": "Refactoring",
  "id": 1,
  "description": "Refactoring",
  "reviews": [
    {
      "name": "Juntao",
      "content": "Very great book",
      "bookId": 1,
      "id": 1
    }
  ]
}
```

Great work! Also, when we `POST` some data to `http://localhost:8080/books/1/reviews` will create `review` under the book with id `1`. Now, we can create the review via the form:



Of course, we have to make a refresh after the submit to see the newly created review:

```
export const saveReview = (id, review) => {
  const config = {
    headers: { 'Content-Type': 'application/json' }
  }

  return (dispatch) => {
    dispatch({type: types.SAVE_BOOK_REVIEW_PENDING})
    return axios.post(`http://localhost:8080/books/${id}/reviews`, JSON.stringify(re\
view), config).then((res) => {
      dispatch({type: types.SAVE_BOOK_REVIEW_SUCCESS, payload: res.data})
      dispatch(fetchABook(id));
    }).catch((err) => {
      dispatch({type: types.SAVE_BOOK_REVIEW_FAILED, err: err.message})
    })
  }
}
```

Note here we add `dispatch(fetchABook(id))` in the success callback. It refreshes the `reviews` for us. However, when you rerun the test, the creation of `review` could fail because we didn't clean up after the test case execution.

To solve this problem (duplicated id), first we need to define a map in `server.js`:

```
const relations = {
  'books': 'reviews'
}
```

And a function just used to generate the `embed` definition, so what it does is generate a `route` dynamically by the given `relations`:

```
function buildRewrite(relations) {
  return _.reduce(relations, (sum, embed, resources) => {
    sum[`/${resources}/:id`] = `/${resources}/:id?_embed=${embed}`
    return sum;
  }, {})
}

server.use(jsonServer.rewriter(buildRewrite(relations)))
```

Now, we can clean the embedded resources by adding an extra step in `DELETE`. Firstly we check if the resource need to be deleted has `embed` resource, if any, we'll clean them along with the resource.

```
server.use((req, res, next) => {
  if (req.method === 'DELETE' && req.query['_cleanup']) {
    const db = router.db
    db.set(req.entity, []).write()

    if (relations[req.entity]) {
      db.set(relations[req.entity], []).write()
    }

    res.sendStatus(204)
  } else {
    next()
  }
})
```

Then we can safely use the `afterEach` to do all the clean up just like before:

```
  afterEach(() => {
    return axios.delete('http://localhost:8080/books?_cleanup=true').catch(err => er\
r)
  })
```

All the tests are now re-enterable with no problem.

Refactoring

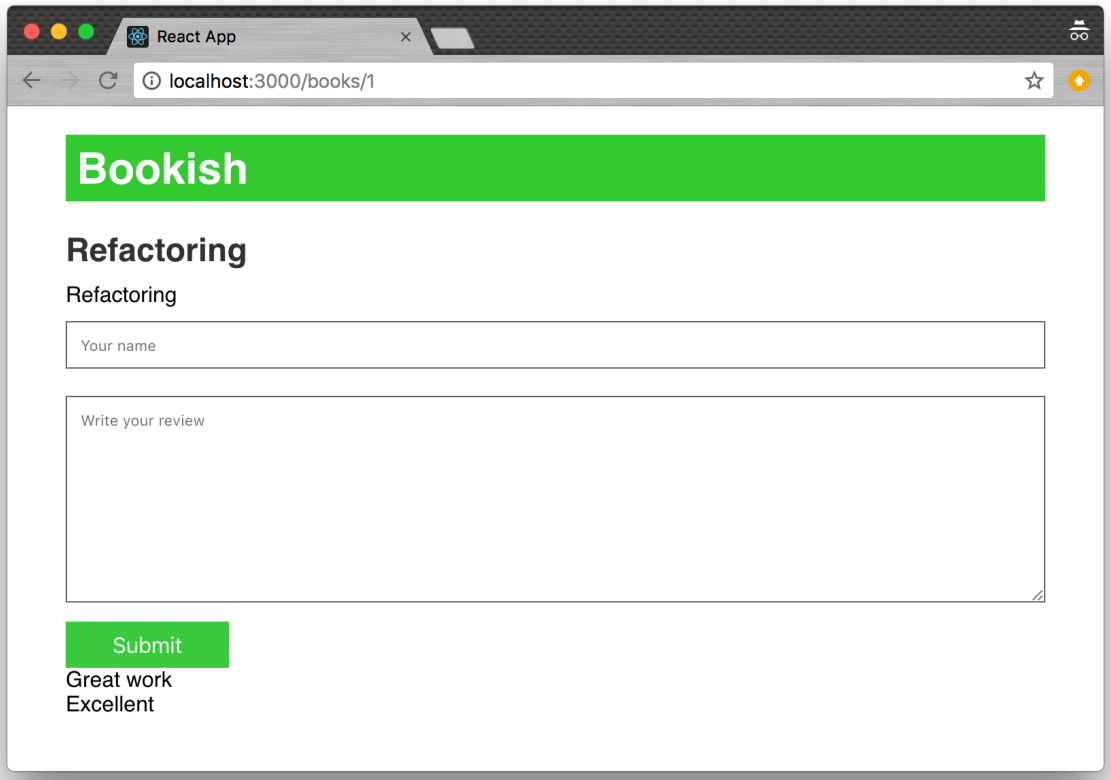
We have already finished the functionality of the `Review` creation and retrieve, our test coverage remains of good quality, which is good. Along with those tests, we can do the refactor work confidently. For `BookDetail`, the `render` method is huge, and it's time to extract some method to make that concise:

```
  renderReviewList = () => {
    const {book} = this.props

    if (book.reviews) {
      return <ReviewList reviews={book.reviews} />
    }
  }

  renderReviewForm = () => {
    return (<form className="review-form">
      <input type="text" name="name" value={this.state.name} onChange={this.updateNa\
```

```
12  me} placeholder="Your name"/>
13        <textarea name="content" cols="30" rows="10" value={this.state.content} onChan\
14  ge={this.updateContent} placeholder="Write your review"/>
15        <button name="submit" onClick={this.saveReview}>Submit</button>
16      </form>)
17    }
18
19    render() {
20      const {book} = this.props
21
22      return (<div className="detail">
23        <h2 className="name">{book.name}</h2>
24        <div className="description">{book.description ? book.description : book.name}\
25  </div>
26
27        {this.renderReviewList()}
28        {this.renderReviewForm()}
29      </div>)
30    }
```

styling

After some styling adjustment, our book detail page turns to this:



Add more fields

If you take a close look at the Review, you'll find there are some essential information are missing: username and created time. We need to complete them:

```
+
+    const name = wrapper.find('.review .name').at(0);
+    expect(name.text()).toEqual('Juntao')
+
+    const date = wrapper.find('.review .date').at(0);
+    expect(date.text()).toEqual('2018/06/21')
```

The implementation should be effortless

```
class ReviewList extends  Component {
  render () {
    const { reviews } = this.props
    return (<div className="reviews-container">
      {reviews.map(review => <div className="review" key={`${review.name}-${review.d\
ate}`}>
        <div>
          <span className="name">{review.name}</span>
          <span className="date">{review.date}</span>
        </div>
        <p>{review.content}</p>
      </div>)}
    </div>)
  }
}
```

It would be a better idea if we can extract the anonymous function in `map` into a separate function when it grows:

```
class ReviewList extends  Component {
  renderReview = (review) => {
    return (<div className="review" key={`${review.name}-${review.date}`}>
      <div>
        <span className="name">{review.name}</span>
        <span className="date">{review.date}</span>
      </div>
      <p>{review.content}</p>
    </div>)
  }

  render () {
    const { reviews } = this.props
    return (<div className="reviews-container">
      {reviews.map(this.renderReview)}
    </div>)
  }
}
```

Then we find that `renderReview` is just a stateless rendering function, that's why we can simplify it like this:

```
import React from 'react'

const Review = ({review}) => {
  return (<div className="review">
    <div>
      <span className="name">{review.name}</span>
      <span className="date">{review.date}</span>
    </div>
    <p>{review.content}</p>
  </div>)
}

export default Review
```

The test for `ReviewList` can be simplified correspondingly:

```
  it('Render List', () => {
    const props = {
      reviews: [
        { name: 'Juntao', date: '2018/06/21', content: 'Excellent work, really impre\
ssive on the efforts you put'},
        { name: 'Abruzzi', date: '2018/06/22', content: 'What a great book'}
      ]
    }

    const wrapper = shallow(<ReviewList {...props}/>)
    expect(wrapper.find(Review).length).toBe(2)
  })
```

So implements `ReviewList` as:

```
import Review from './Review'

class ReviewList extends  Component {
  render () {
    const { reviews } = this.props
    return (<div className="reviews-container">
      {reviews.map(review => <Review review={review} key={`${review.name}-${review.d\
ate}`} />)}
    </div>)
  }
}
```

```
12
13 export default ReviewList
```

As you can see, the `ReviewList` itself is nothing more than a presentational component, this could be simplified as:

```
1  import Review from './Review'
2
3  const ReviewList = ({reviews}) => {
4    return (<div className="reviews-container">
5      {reviews.map(review => <Review review={review} key={`${review.name}-${review.dat\
6  e}`} />)}
7    </div>)
8  }
9
10 export default ReviewList
```

Brilliant! Since all the detail of how to render a book review is encapsulated in `Review`, all the tests related could be moved into the test file to the `Review` component.

```
1  describe('Review', () => {
2
3    it('Render Review', () => {
4      const props = {
5        review: { name: 'Juntao', date: '2018/06/21', content: 'Excellent work, really\
6   impressive on the efforts you put'}
7      }
8
9      const wrapper = shallow(<Review {...props}/>)
10     const firstReview = wrapper.find('.review p').at(0);
11     expect(firstReview.text()).toEqual('Excellent work, really impressive on the eff\
12 orts you put');
13
14     const name = wrapper.find('.review .name').at(0);
15     expect(name.text()).toEqual('Juntao')
16
17     const date = wrapper.find('.review .date').at(0);
18     expect(date.text()).toEqual('2018/06/21')
19   })
20 })
```

Now our code turns out to be more concise and cohesive, with cleaned responsibility. Additionally, in favor of the test coverage, we don't have to worry about break the existing function when we are refactoring.

Review Editing

`Review` component now provides the basic presentation. However, in the real world, the user could typo some content, or would rather rewrite the content. So we need to allow the user to edit the `Review` they already posted.

There should be an `Edit` button when a user clicks it, the text on the button changed to `Submit` (waiting for the user to submit). Likely, when a user clicked `Submit`, the text turns to `Edit` again. So the first test could be:

```
  it('Editing', () => {
    const props = {
      review: { name: 'Juntao', date: '2018/06/21', content: 'Excellent work, really\
 impressive on the efforts you put'}
    }

    const wrapper = shallow(<Review {...props}/>)

    expect(wrapper.find('button.submit').length).toEqual(0)
    expect(wrapper.find('button.edit').length).toEqual(1)

    wrapper.find('button.edit').simulate('click');

    expect(wrapper.find('button.submit').length).toEqual(1)
    expect(wrapper.find('button.edit').length).toEqual(0)
  })
```

By using `simulate`, we can easily simulate the click event on `.edit` button, and verify the text changes on the button. We can achieve that by introducing a `state` in the component:

```
  constructor(props) {
    super(props)
    this.state = {
      editing: false
    }
  }
```

All we need to do is toggle the status of `editing`:

```
  edit = () => {
    this.setState({
      editing: true
    })
  }

  submit = () => {
    this.setState({
      editing: false
    })
  }
```

So for rendering, we can decide which text to display by the `editing` state like this:

```
      {this.state.editing ?
        <button className="submit" onClick={this.submit}>Submit</button>:
        <button className="edit" onClick={this.edit}>Edit</button>}
```

What's more, we'd like there is a `textarea` shows up once the user clicked `Edit`, and copy all the review content into the `textarea` for editing:

```
    expect(wrapper.find('button.submit').length).toEqual(0)
    expect(wrapper.find('button.edit').length).toEqual(1)
+   expect(wrapper.find('.review p').text()).toEqual('Excellent work, really impress\
ive on the efforts you put');

    wrapper.find('button.edit').simulate('click');

    expect(wrapper.find('button.submit').length).toEqual(1)
    expect(wrapper.find('button.edit').length).toEqual(0)
+
+   expect(wrapper.find('.review textarea').props().value).toEqual('Excellent work, \
really impressive on the efforts you put');
})
```

To implement that, we first have to maintain those content in `state` as well:

```
  this.state = {
      editing: false,
+     content: ''
    }
```

Also, render the `textarea` and `static text` based on `editing` state:

```
      {this.state.editing ?
        <textarea value={this.state.content} cols="30" rows="10" className="review-c\
ontent" onChange={this.updateContent}/>:
        <p>{review.content}</p>}
```

So `edit` method would be modified as:

```
  edit = () => {
    const { content } = this.props.review
    this.setState({
      editing: true,
      content: content
    })
  }
```

Additionally, we update the `this.stat.content` whenever a user is typing something:

```
  updateContent = (e) => {
    this.setState({
      content: e.target.value
    })
  }
```

Now the Review has 2 different status: `viewing` and `editing`, and can be switched by clicking the `.edit` button on the fly.

Save the review - action

Just like how we create a review, save a review need send a request to the backend. The good news is that `json-server` has already provided that. We send a `PUT` request to `http://localhost:8080/reviews/{id}` would update a review. Of course, we have to write a test for the redux action first:

```
    it('Update a review for a book', () => {
      const config = {
        headers: { 'Content-Type': 'application/json' }
      }

      const review = {
        name: 'Juntao Qiu',
        content: 'Excellent work!'
      }

      axios.put = jest.fn().mockImplementation(() => Promise.resolve({}))

      const store = mockStore({list: { books: [], term: '' }})

      return store.dispatch(updateReview(1, review)).then(() => {
        expect(axios.put).toHaveBeenCalledWith('http://localhost:8080/reviews/1', JS\
ON.stringify(review), config)
      })
    })
```

Note that we mocked `axios.put` here. Generally speaking, when you update some existing resource, you use `PUT` as the HTTP verb.

```
export const updateReview = (id, review) => {
  const config = {
    headers: { 'Content-Type': 'application/json' }
  }

  return (dispatch) => {
    dispatch({type: types.SAVE_BOOK_REVIEW_PENDING})
    return axios.put(`http://localhost:8080/reviews/${id}`, JSON.stringify(review), \
config).then((res) => {
      dispatch({type: types.SAVE_BOOK_REVIEW_SUCCESS, payload: res.data})
    }).catch((err) => {
      dispatch({type: types.SAVE_BOOK_REVIEW_FAILED, err: err.message})
    })
  }
}
```

We are reusing the `SAVE_BOOK_REVIEW` type here

Integration

So all the parts for editing a review are ready, it's time for us to put all those parts together. We need to make sure that when `Submit` is clicked, the `save action` is invoked:

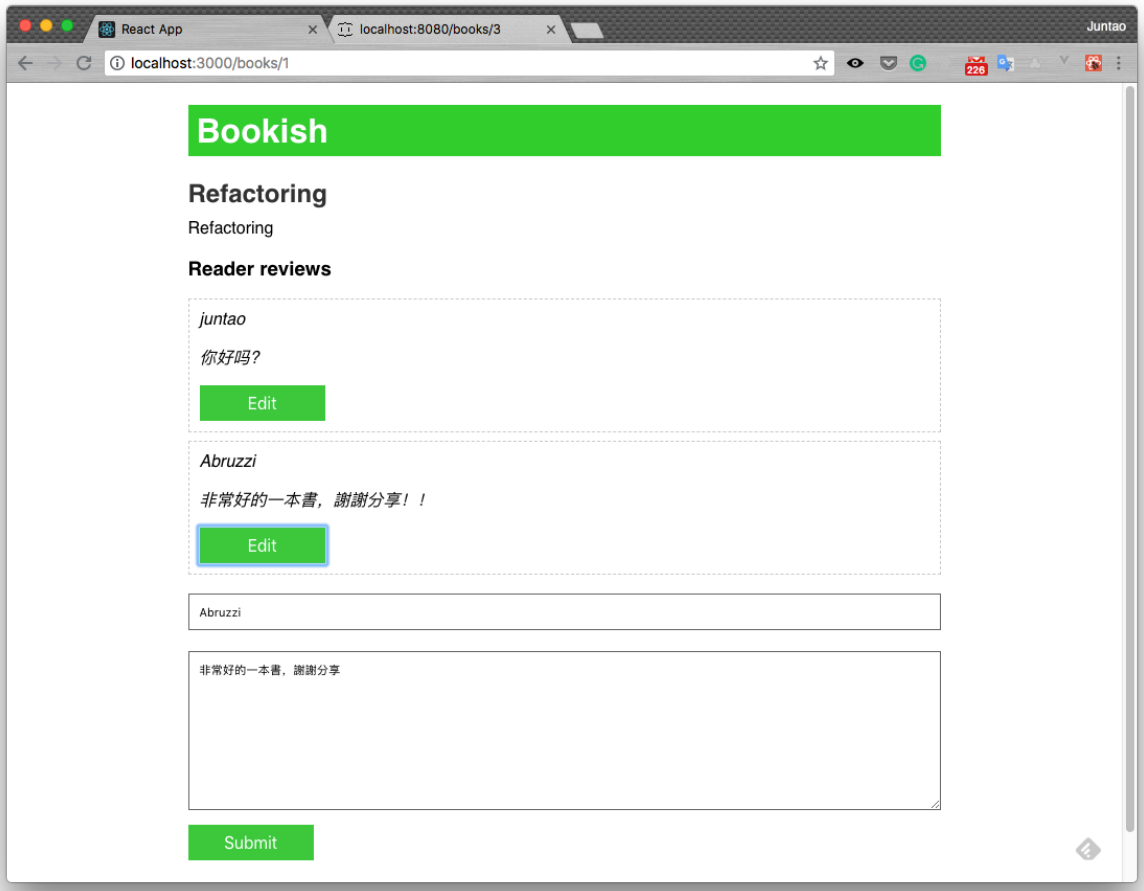
```
  it('Submit the updates', () => {
    const props = {
      review: { name: 'Juntao', date: '2018/06/21', content: 'Excellent work, really\
 impressive on the efforts you put'},
      updateReview: jest.fn()
    }

    const wrapper = shallow(<Review {...props}/>)

    wrapper.find('button.edit').simulate('click')
    expect(wrapper.find('.review textarea').props().value).toEqual('Excellent work, \
really impressive on the efforts you put');
    wrapper.find('button.submit').simulate('click')
    expect(props.updateReview).toHaveBeenCalled()
  })
```

Because the correctness of `updateReview` is already verified in `action` tests, so we simply use `jest.fn()` here:

```
  submit = () => {
    this.setState({
      editing: false
    })
    const { id, bookId, name, date } = this.props.review
    const content = this.state.content
    this.props.updateReview(id, {
      bookId, name, date, content
    })
  }
```

When a user clicked `Submit` button, we firstly set `editing` as `false`, then arrange parameters for `updateReview` invocation. `updateReview` is passed from redux by `connect` API.

```
export default connect(null, { updateReview })(Review)
```



Refactoring the tests

In the previous chapters, what we do in the tests are simply copy and paste, which is not an appropriate way to code. We now have all the tests are passing, we can step to the next phase of TDD: `refactoring`.

Test code is as important as the production code by any means, that's what lots of developers are not aware, or pretend not to be aware. If you choose to ignore the quality of the test code itself, it will corrupt very fast, faster than you would expect, and then you got unreadable tests cases, meaningless variable names, and misleading test descriptions. As a consequence, those tests will be soon dropped, then lost the responsibility of protecting the functionality from being the break.

Hide the details

There are too many details exposed to the tests currently, a well designed `PageObject` would reduce this problem to a vast extent:

```
  test('Goto book detail page', async () => {
    await page.goto(`${appUrlBase}/`)
    await page.waitForSelector('a.view-detail')

    const links = await page.evaluate(() => {
      return [...document.querySelectorAll('a.view-detail')].map(el => el.getAttribu\
te('href'))
    })

    await Promise.all([
      page.waitForNavigation({waitUntil: 'networkidle2'}),
      page.goto(`${appUrlBase}${links[0]}`)
    ])

    const url = await page.evaluate('location.href')
    expect(url).toEqual(`${appUrlBase}/books/1`)

    await page.waitForSelector('.description')
    const result = await page.evaluate(() => {
      return document.querySelector('.description').innerText
    })
    expect(result).toEqual('Refactoring')
  })
```

We don't have to verify the click-and-redirect function, that's the built-in function of an `a` tag in HTML. Instead, we just need to make sure the URL passed to `a` tag is correct, then the browser will do the redirect work for us with no problem.

The test case can be simplified like:

```
  test('Goto book detail page', async () => {
    const detailPage = new DetailPage(browser, 1)
    await detailPage.initialize()

    const desc = await detailPage.getDescription()
    expect(desc).toEqual('Refactoring')
  })
```

So we can put the implementation detail into `DetailPage` file instead:

```
import { APP_BASE_URL } from '../constants'

export default class DetailPage {
  constructor(browser, id) {
    this.browser = browser
    this.id = id
  }

  async initialize() {
    this.page = await this.browser.newPage()
    await this.page.goto(`${APP_BASE_URL}/books/${this.id}`)
  }

  async getDescription() {
    await this.page.waitForSelector('.description')
    return await this.page.evaluate(() => {
      return document.querySelector('.description').innerText
    })
  }

}
```

Likewise, we can make the `add a review` test much more concise by move the details to `DetailPage`, what's more, the readability is increased significantly:

```
  test('Write an review for a book', async () => {
    const detailPage = new DetailPage(browser, 1)
    await detailPage.initialize()

    const review = {
      name: 'Juntao Qiu',
      content: 'Excellent works!'
    }

    await detailPage.addReview(review)

    const result = await detailPage.getReview(0)
    expect(result).toEqual('Excellent works!');
  })
```

The method `addReview` and `getReview` are all moved into `DetailPage`:

```
  async addReview(review) {

    await this.page.waitForSelector('input[name="name"]')
    await this.page.type('input[name="name"]', review.name, {delay: 20})

    await this.page.waitForSelector('textarea[name="content"]')
    await this.page.type('textarea[name="content"]', review.content, {delay: 20})

    await this.page.waitForSelector('button[name="submit"]')
    await this.page.click('button[name="submit"]');
  }

  async getReview(index) {
    await this.page.waitForSelector('.review')
    const reviews = await this.page.evaluate(() => {
      return [...document.querySelectorAll('.review p')].map(el => el.innerText)
    })
    return reviews[index]
  }
```

We have achieved a more neat and short test code and separated the different concerns into different files. The `PageObject` could potentially be reused in different test scenarios.

```
src/e2e
├── constants.js
├── e2e.test.js
└── pages
    ├── DetailPage.js
    └── ListPage.js

1 directory, 4 files
```

That's kind of the first step of `Live Document` we are going to discuss in next chapter which will improve the maintainability and readability of test code, and both business team and develop team could benefit.

Behavior Driven Development

The definition of BDD in Wikipedia is:

> In software engineering, behaviour-driven development (BDD) is a software development process that emerged from test-driven development (TDD). Behaviour-driven development combines the general techniques and principles of TDD with ideas from domain-driven design and object-oriented analysis and design to provide software development and management teams with shared tools and a shared process to collaborate on software development.

Surrounding BDD itself, there are a variety of practices in real projects, such as *Specification by Example*, *Live Document*, and so on. Those techniques can improve collaboration among different roles in a team. On the one hand, developers can understand the big picture of the business and could make a better decision regarding business restrictions. On the other hand, `Live Document` can make sure the software behaves correctly when there are some changes come from the business requirements. It potentially could protect situations like - when all tests are passing, but the behaviour of the system is incorrect - from happening.

There are a lot of tools available when you try to adopt `BDD` as a practice in your team, what we are going to demonstrate here is `cucumber`, which is a powerful tool that uses a `DSL`(Domain Specific Language) for developers to compose human-readable document first, and yet executable code as a side effect (by some pure magic we will address soon).

It could be used as a tool for communication between business analyst and developers who write the business rules in code. As we all aware of, most of the defects come from the miscommunication, so it is crucial to have a dedicated tool to help us with that. Even in some cases, the `Live Document` written by `cucumber` is not executable, or too expensive to run regularly. It's still a valid tool that could help `QA`, as a guide, do the manual tests.

Enough theory now, let's get started.

Play with cucumber

Firstly, we will create a folder for the `cucumber` and `Live Documents`:

```
1 mkdir -p bookish-react-cucumber
2 cd bookish-react-cucumber
```

And then we can install `cucumber` package by the following command:

```
yarn add cucumber --save
```

Since we are going to keep using `puppeteer` to drive the headless browser under the hood and use `chai` as the test runner, we need to install those packages as well. In the same folder:

```
yarn add puppeteer chai --save
```

As what we saw in previous chapters, we have already learnt how to work with the `ES6`, that's no reason to stop doing that with `cucumber` since the code would be very concise and readable compared with `ES5`. Lets' install the `babel` and its companions that allow us to compile and translate `ES6` to `ES5` underlying:

```
yarn add @babel/preset-env @babel/core @babel/register babel-polyfill --save
```

Finally, define a `.babelrc` file like:

```
{
  "presets": ["@babel/preset-env"]
}
```

This file tells babel to pick up `@babel/preset-env` as preset, you can specify other `preset` as well enable particular features for `babel`.

Live Document

`cucumber` honour the `convention over configuration` principle, all you have to do is put files in certain places, and then `cucumber` can do the rest and make sure all parts work together smoothly. That principle is pretty popular now and lots of framework work under this manner, the most famous one would be `Ruby on Rails`, or `Spring Boot` in Java world.

The convention which `cucumber` is pretty straightforward, I've created one and you can use `tree` command to see it visually:

```
$ tree features
```

The output looks like this:

```
features
├── book-list.feature
├── pages
|   ├── ListPage.js
├── step_definitions
|   └── book-list.steps.js
└── support
    └── world.js
```

To run the cucumber, all we do is type this in the command line:

```
./node_modules/.bin/cucumber-js
```

Under the hood, `cucumber-js` will search all files end with `.feature` under `features` folder automatically, and load all the files end with `.steps.js` and other global settings for set all environment up, after that it will try to run each test individually.

The first feature specification

Because you can describe your test in plain English, it should be straightforward to put the AC we described in Chapter 2 here, but in a particular format:

```
Feature: Book List
  As a reader
  I want to see books in the trend
  So I can learn what to read next

  Scenario: Heading
    Given I am a bookish user
    When I open the "list" page
    Then I can see the title "Bookish" is showing
```

Some of the text above is just for human beings. For example, the interpreter isn't interested with the `As a <role>, I want to <do something>, So that<business values>` part at all, that part is like comments in the code. In contrast, it would start right from the `Scenario` section beneath.

Define the Steps

All the sentences above (in `Scenario` section) as a whole, is called a `step` definition, and need to be translated to executable code in some way. `cucumber` use the regular expression to match the sentence. Moreover, it's trying to extract some parameter from the sentence and then passed them into the `step` function.

For example:

```
import { Before, After, Given, When, Then } from 'cucumber'

Given(/^I am a bookish user$/, function () {

})

When(/^I open the "([^"]*)" page$/, function (page) {
  console.log(page)
});

Then(/^I can see the title "([^"]*)" is showing$/, function (title) {
  console.log(title)
})
```

The `parameters` passed into function `Given`, `When` and `Then` are pretty similar, the first one is a regular expression, which used to match a sentence in `.feature` files. And the second one, which is a callback, will be invoked once there is a match. If there are some patterns in the regular expression, the value will be extracted and passed to the callback. This is a simple but powerful mechanism for us, as a developer, to do some interesting work - such as, launch the browser and check if a particular elements are showing on the page.

Note here we are using `ES6`, so we need to pass some options to `cucumber-js` to do the translation before running the actual code:

```
$ ./node_modules/.bin/cucumber-js --require-module @babel/register --require-module \
babel-polyfill
```

Then you should see something like this in the console:

```
$ ./node_modules/.bin/cucumber-js --require-module @babel/register --require-module \
babel-polyfill
..list
.Bookish
..

1 scenario (1 passed)
3 steps (3 passed)
0m00.185s
```

That's to say, our `Feature` is interpreted correctly, and the parameters are extracted and passed to method correspondingly.

Global settings - World

To make the information be passed from one step to another, there should be some kind of `context` be involved. For example, when we open a page for the first step and want to read some information on the next step. `cucumber-js` provides a mechanism just to do that, you can create a file named `world.js` in `support` folder, and you can define the `context` in this file.

Let's take a look at how to use it by an example:

```
import { setWorldConstructor } from 'cucumber'

class CustomWorld {
  constructor() {
    this.shared = []
  }

  put(element) {
      this.shared.push(element)
  }

  get(index) {
      return this.shared[index]
  }
}

setWorldConstructor(CustomWorld)
```

Here we defined a globally accessible variable named `shared` as an array. And two corresponding method `get` and `set`. `cucumber` will initialise the global variable and you can access it from each step via `this` like the code following:

```
When(/^I open the "([^"]*)" page$/, function (page) {
  this.put(page)
});

Then(/^I can see the title "([^"]*)" is showing$/, function (title) {
  const page = this.get(0)
  console.log(page)
})
```

Note that you can use `this.get(0)` to access the data saved from the previous step (the value of `page` variable).

Let's modify it a little bit to fit our purpose to test against a Web application:

```
class CustomWorld {
  constructor() {
    this.browser = null
  }

  async start() {
    this.browser = await puppeteer.launch({})
  }

  async close() {
    await this.browser.close()
  }
}
```

Now, the `CustomWorld` can `start` and `close` a headless browser powered by `puppeteer`. And we can start the browser in any step by just calling:

```
this.start()
```

However, that is not always the case. In most scenarios, you would just launch the browser at the beginning of any tests. Luckily, `cucumber` provides some hooks you can use `before` or `after` any steps. For example, we can start and initialise a browser before each `scenario`:

```
Before(async function() {
  await this.start()
})

After(async function() {
  await this.close()
})
```

Ok, that's all for the cucumber basics. Just before we dive into any `BDD` cases, let take a look at one wild used pattern that can improve the flexibility of our tests: Page Object.

Page Object

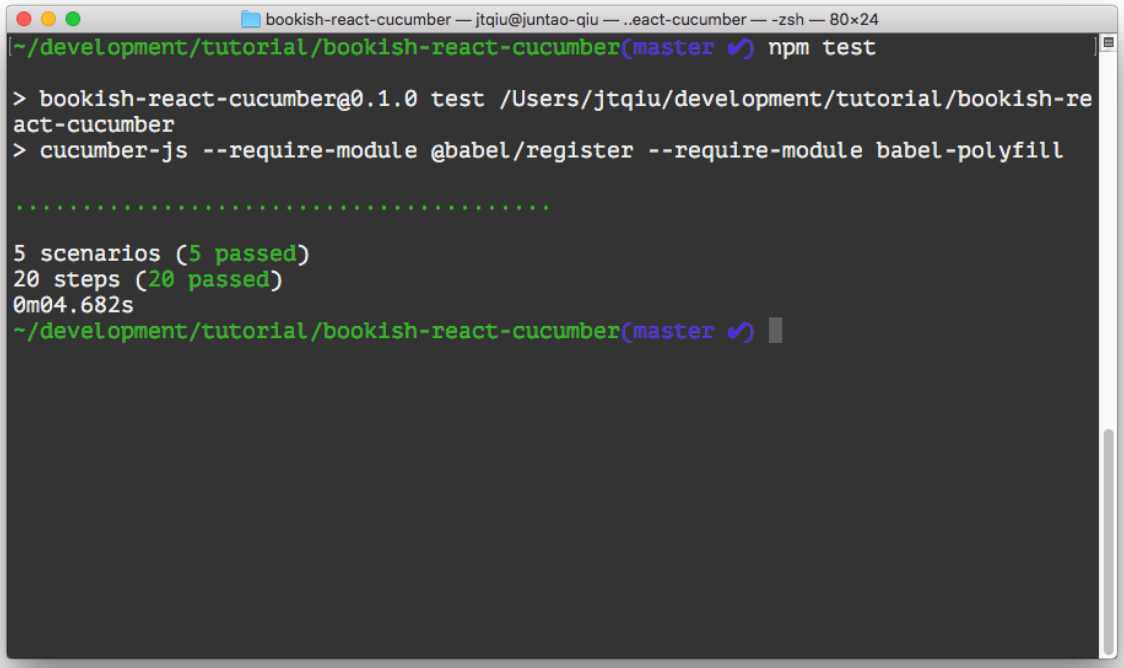
We can definitely use the `Page Object` created in the previous Chapter, and invoke the methods in it:

```
import puppeteer from 'puppeteer'
import { setWorldConstructor } from 'cucumber'

import ListPage from '../pages/ListPage'

class CustomWorld {
  constructor() {
    this.browser = null
    this.listPage = null
  }

  async start() {
    this.browser = await puppeteer.launch({})
  }

  async gotoListPage() {
    this.listPage = new ListPage(this.browser)
    await this.listPage.initialize()
  }

  async getListPage() {
    return this.listPage
  }

  async close() {
    await this.browser.close()
  }
}

setWorldConstructor(CustomWorld)
```

Then in the `step` definition, we can simply access those methods:

```
When(/^I open the "([^"]*)" page$/, async function (page) {
  await this.gotoListPage()
})

Then(/^I can see the title "([^"]*)" is showing$/, async function (title) {
  const page = await this.getListPage()
  const heading = await page.getHeading()
  expect(heading).to.eql(title);
})
```

In the first step, we created a `ListPage` object, and use it to access `heading` by invoking `getHeading`, finally we use `expect` provided by `chai` to do the assertion.



With all of that knowledge, I bet we can get started on writing some real-world BDD cases now.

Book List

Now, let's start with a more complicated example.

```
1  Scenario: Book List
2    Given I am a bookish user
3    When I open the "list" page
4    Then I can see "3" books
5    And there are
6      | name                  |
7      | Refactoring           |
8      | Domain-driven design  |
9      | Building Micro-service |
```

As you may be noticed, you can define a more complex data structure in `feature` file by using `table`: the structure enclosed by pipe `|`. Each row will be treated as a row in a table, and you can actually define many columns for each row like:

```
    And there are
      | name                  | price |
      | Refactoring           | $100  |
      | Domain-driven design  | $120  |
      | Building Micro-service | $80   |
```

cucumber provides a compelling DTI(Data Table Interface) to help developers to parse and use those data table. For example, if we want to get the BookList defined in feature file within step, just using table.rows:

```
Then(/^there are$/, async function (table) {
    console.log(table.rows())
});
```

And you can get data in this shape:

```
[ [ 'Refactoring' ],
  [ 'Domain-driven design' ],
  [ 'Building Micro-service' ] ]
```

Alternatively, if you perfer JSON, you can call table.hashes() instead:

```
[ { name: 'Refactoring' },
  { name: 'Domain-driven design' },
  { name: 'Building Micro-service' } ]
```

Thus, in our step definition, we can use the DTI to do the assertion:

```
Then(/^I can see "([^"]*)" books$/, async function (number) {
  const page = await this.getListPage()
  const books = await page.getBooks()
  expect(books.length).to.eql(parseInt(number))
});

Then(/^there are$/, async function (table) {
  const page = await this.getListPage()
  const books = await page.getBooks()

  const actual = table.rows().map(x => x[0])

  expect(books).to.include.members(actual);
});
```

Just as what we have done in the raw puppeteer code, we need to setup/teardown fixture data by using Before and After hooks:

```
Before(async function() {
  const books = [
    {"name": "Refactoring", "id": 1, "description": "Refactoring"},
    {"name": "Domain-driven design", "id": 2, "description": "Domain-driven design"},
    {"name": "Building Micro-service", "id": 3, "description": "Building Micro-servi\
ce"}
  ]

  await books.map(item => axios.post('http://localhost:8080/books', item, {headers: \
{ 'Content-Type': 'application/json' }}))
})

After(async function() {
  await axios.delete('http://localhost:8080/books?_cleanup=true').catch(err => err)
})
```

It should be pretty straightforward for you since we saw similar stuff already in `puppeteer`.

Searching functionality

The next scene we can test on is the feature `searching`, we could describe the business requirement in plain English:

```
  Scenario: Search by keyword
    Given I am a bookish user
    When I open the "list" page
    And I typed "design" to perform a search
    Then I should see "1" book is matched
    And its name is "Domain-driven design"
```

It could be effortless to implement those provided we have the `Page Object` in position:

```
When(/^I typed "([^"]*)" to perform a search$/, async function (keyword) {
  const page = await this.getListPage()
  await page.search(keyword);
});

Then(/^I should see "([^"]*)" book is matched$/, async function (number) {
  const page = await this.getListPage()
  const books = await page.getBooks();
  expect(books.length).to.eql(parseInt(number))
});
```

```
11
12 Then(/^its name is "([^"]*)"$/, async function (name) {
13   const page = await this.getListPage()
14   const books = await page.getBooks();
15   expect(books[0]).to.eql(name)
16 });
```

Neat, the `step` functions are almost self-explaining.

Reviews page

Similarly, we can rewrite the `review` feature tests in the following sentence, in English:

```
1   Scenario: Write a review
2     Given I am a bookish user
3     When I open the book detail page with id "1"
4     And I add a review to that book
5       | name       | content          |
6       | Juntao Qiu | Excellent works! |
7     Then I can see it shows beneath the description section
8     And the content is "Excellent works!"
```

We could reuse a lot of steps defined previously, also note that we use `Data Table Interface` to extract parameters passed in:

```
1 When(/^I add a review to that book$/, async function (table) {
2   const reviews = table.hashes()
3   const detailPage = await this.getDetailPage()
4   await detailPage.addReview(reviews[0])
5 })
```

This is actually very convenient for us given that the `addReview` function takes an object in that particular format:

```
async addReview(review) {
  await this.page.waitForSelector('input[name="name"]')
  await this.page.type('input[name="name"]', review.name, {delay: 20})

  await this.page.waitForSelector('textarea[name="content"]')
  await this.page.type('textarea[name="content"]', review.content, {delay: 20})

  await this.page.waitForSelector('button[name="submit"]')
  await this.page.click('button[name="submit"]');
}
```

It's pretty straightforward to reuse the method we defined in `Page Object`. And it's effortless to make sure the `content` is correct by use `detailPage.getReview`:

```
Then(/^the content is "([^"]*)"$/, async function (content) {
  const detailPage = await this.getDetailPage()
  const result = await detailPage.getReview(0)
  expect(result).to.eql(content);
})
```

As you can see here, by extract behaviour into its `PageObject`, we can make it much more concise and meaningful when read the `step` function. And by putting all related code together would make the future changes much more manageable, say, if there are any UI elements changes, we could easily navigate to the corresponding file and modify it in place without effect other pages.

Test Report

What's more, `cucumber` provides a fantastic way to generate a report in different formats, my favourite format is `json` because that allows us to visualize the data in our way. To output the test results in `json`, you can simply specify an option like this:

```
mkdir -p reports
./node_modules/.bin/cucumber-js --require-module @babel/register --require-module ba\
bel-polyfill -f json:reports/report.json
```

That will create a `reports` folder, and the test result will be generated in file `report.json` under the folder.

The `report.json` generated should look like this:

```
[
  {
    "description": "  As a reader\n  I want to see detail for a particular book\n  S\
o I can make the decision whether to buy it more easily",
    "keyword": "Feature",
    "name": "Book List",
    "line": 1,
    "id": "book-list",
    "tags": [],
    "uri": "features/book-detail.feature",
    "elements": [/*...*/]
  },
  {
    "description": "  As a reader\n  I want to see books in trend\n  So I can learn \
that what to read next",
    "keyword": "Feature",
    "name": "Book List",
    "line": 1,
    "id": "book-list",
    "tags": [],
    "uri": "features/book-list.feature",
    "elements": [/*...*/]
  }
]
```

And in each `elements` there is `scenario` execution result, like this:

```
{
        "id": "book-list;book-detail",
        "keyword": "Scenario",
        "line": 6,
        "name": "Book Detail",
        "tags": [],
        "type": "scenario",
        "steps": [
          {
            "arguments": [],
            "keyword": "Given ",
            "line": 7,
            "name": "I am a bookish user",
            "match": {
              "location": "features/step_definitions/book-list.steps.js:26"
            },
```

```
            "result": {
              "status": "passed"
            }
          },
          {
            "arguments": [],
            "keyword": "When ",
            "line": 8,
            "name": "I open the book detail page with id \"1\"",
            "match": {
              "location": "features/step_definitions/book-list.steps.js:34"
            },
            "result": {
              "status": "passed",
              "duration": 617000000
            }
          },
          {
            "arguments": [],
            "keyword": "Then ",
            "line": 9,
            "name": "I can see the description \"Refactoring\" is showing",
            "match": {
              "location": "features/step_definitions/book-list.steps.js:38"
            },
            "result": {
              "status": "passed",
              "duration": 29000000
            }
          }
        ]
      }
```

Those metadata could be used to generate the final `HTML` report or report in other formats. To generate an HTML report in our case, we need to have `cucumber-html-reporter` installed as a plugin to `cucumber`:

```
yarn add cucumber-html-reporter --save
```

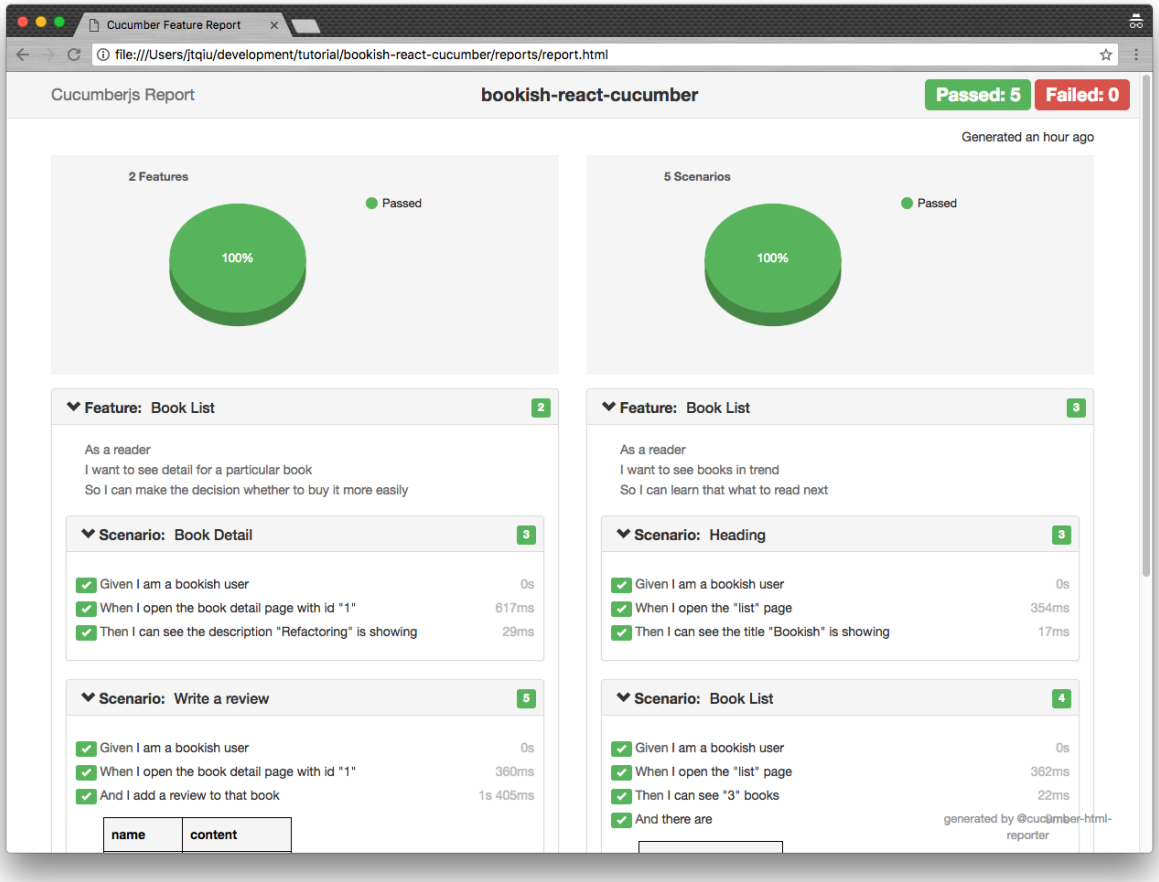
And we can write a simple script to generate an HTML report based on the json:

```
var reporter = require('cucumber-html-reporter');

var options = {
  theme: 'bootstrap',
  jsonFile: 'reports/report.json',
  output: 'reports/report.html',
  reportSuiteAsScenarios: true,
  launchReport: true
};

reporter.generate(options);
```

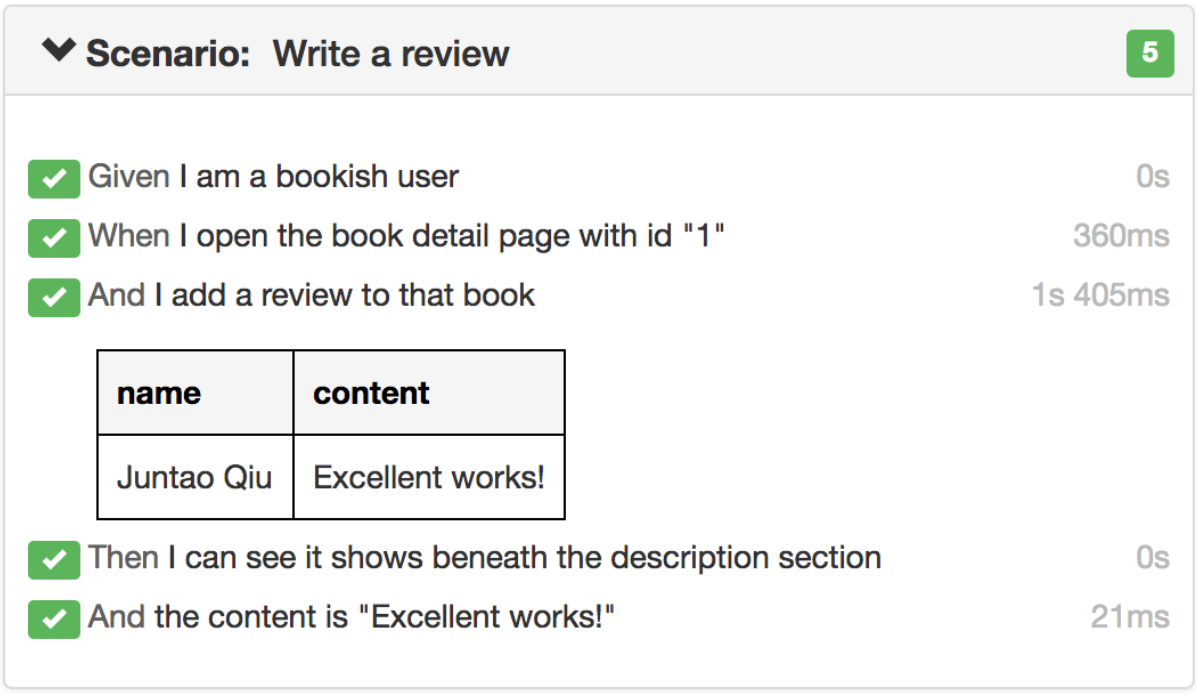
Finally, we run the following command to generate the report.

```
node report.js
```

The result of HTML would look like this:



Alternatively, you can investigate each scenario with more detail, with execution time for each step:



Design the state shape of your application

A good design of your application is crucial in many cases, it would enable you to access the data more efficient and in a more natural manner. In contrast, some design may lead you to to do unnecessary for-loop or using too many levels in a tree to fetch the data you want. In this chapter, we will discuss how we can refactor current data structure into a new one that may help us to organize code neater and easy to understand.

Before jump directly into the re-shaping, let's first take a close at how we handle errors in the whole application.

Error handling

In a perfect world everything goes well, the network is reliable, APIs always return whatever you're requesting, and it is guaranteed that responses are in the right order. However, in the real world, error happens, things like network issues, backend outage is quite often that you could expect.

That's why the error handling is so essential – although there are many parts are out of our control, we can always work something out for the failures. A try-catch pair can catch the unexpected error at runtime and allow us to try to fix by, say, retry the request several seconds later.

Since the errors are inevitable in most cases, we need some reliable mechanism to ensure we can recover and back to business when the error occurred. As you may already be noticed, we have many places in the code that could fail, such as fetch the `books` or a `book` when `id` is provided. The book we're requesting could not exist, or the request could fail by the network failure itself.

Currently, we're catching those error in `reducers` like this:

```
1  case types.FETCH_BOOKS_FAILED:
2    return {
3      ...state,
4      loading: false,
5      error: action.err
6    }
```

Eventually, we could end up with many cases to handle those kinds of failure like:

```
case types.FETCH_BOOK_LIST_FAILED:
//...
case types.FETCH_BOOK_FAILED:
//...
case types.FETCH_REVIEW_FAILED:
//...
case types.SAVE_BOOK_REVIEW_FAILED:
//...
```

And we can expect that there would be much boilerplate code in our application. Even worse, the shape of the `state` could turn to very complicated, which can lead to many consequence problems.

Intercept the errors

What about we define a `global` error handling that can intercept the actions spreading and inspect the action to see if it's an error and if it is then we can set some flag or even inject the error message into the global `store`. Individual components that responsible for rendering data, on the other hand, are unaware of this from happening.

I like the idea that **Sam Aryasa** shared on medium[9]. And I suppose that was just delight and no-invention for the existing code. Additionally, it could simplify the `reducer` significantly only to handle the happy-path and leave the dirty and drone work to the interceptor.

Let's define our interceptor in a separate file, as always, let's start writing the specification first:

```
describe('Errors handling', () => {
  it('Inject error message into global context', () => {
    const initState = {}
    const action = {type: 'FETCH_BOOK_FAILED', payload: { message: '404 - Not Found'\
 }}

    const state = errors(initState, action)

    expect(state['FETCH_BOOK']).toEqual('404 - Not Found')
  })
})
```

So in the test above, when the action `FETCH_BOOK_FAILED` is raised for a reason `404 - Not Found`. We would expect there is a new field(with the key of the action name `FETCH_BOOK`) in the state with value `404 - Not Found` to be set. Essentially, the function `errors` is just yet another reducer. However, it would inspect actions passing through it and check the `suffix` of the `action.type`, if `FAILED` is found, then it will extract the message out of `action.payload` and put it into the state. For all other cases, it just passes it to the next reducer and leaves the action as it was.

[9] https://medium.com/stashaway-engineering/react-redux-tips-better-way-to-handle-loading-flags-in-your-reducers-afda42a804c6

```
export default (state = {}, action) => {
  const { type, payload } = action;
  const matches = /(.*)_(FAILED)/.exec(type);

  if (!matches) return state;

  const [, name, ] = matches;
  return {
    ...state,
    [name]: payload.message
  }
}
```

Moreover, we should clear the error message when the corresponding request is re-send:

```
  it('Clear up error message when request is send', () => {
    const initState = {}
    const action = {type: 'FETCH_BOOK_PENDING', payload: { message: '404 - Not Found\
' }}

    const state = errors(initState, action)

    expect(state['FETCH_BOOK']).toEqual("")
  })
```

So we then say that if the action is in `PENDING` status, we will clean up the error message:

```
export default (state = {}, action) => {
  const { type, payload } = action;
  const matches = /(.*)_(PENDING|FAILED)/.exec(type);

  if (!matches) return state;

  const [, name, status] = matches;
  return {
    ...state,
    [name]: status === 'FAILED' ? payload.message : ''
  }
}
```

And we don't care about the action that without any `payload`:

```
  it('Pass it through when its not a request', () => {
    const initState = {}
    const action = {type: 'REALLY_SIMPLE_ACTION'}

    const state = errors(initState, action)

    expect(state).toEqual(initState)
  })


  it('Pass it through when request doesn\'t have payload', () => {
    const initState = {}
    const action = {type: 'FETCH_SOMETHING_PENDING'}

    const state = errors(initState, action)

    expect(state).toEqual(initState)
  })
```

That would require us to filter out actions without `payload`:

```
- if (!matches) return state;
+ if (!matches || !payload) return state;
```

The data shape

Currently, the state shape is a little bit rough. For example, in the `reducers.js`, we have the initialState in `listing` reducer in the following shape:

```
const initialState = {
  term: '',
  loading: true,
  books: [],
  error: '',
  current: {}
}
```

So we use this data in `BookListContainer` like:

```
const mapStateToProps = state => ({
  loading: state.list.loading,
  books: state.list.books,
  error: state.list.error,
  term: state.list.term
})
```

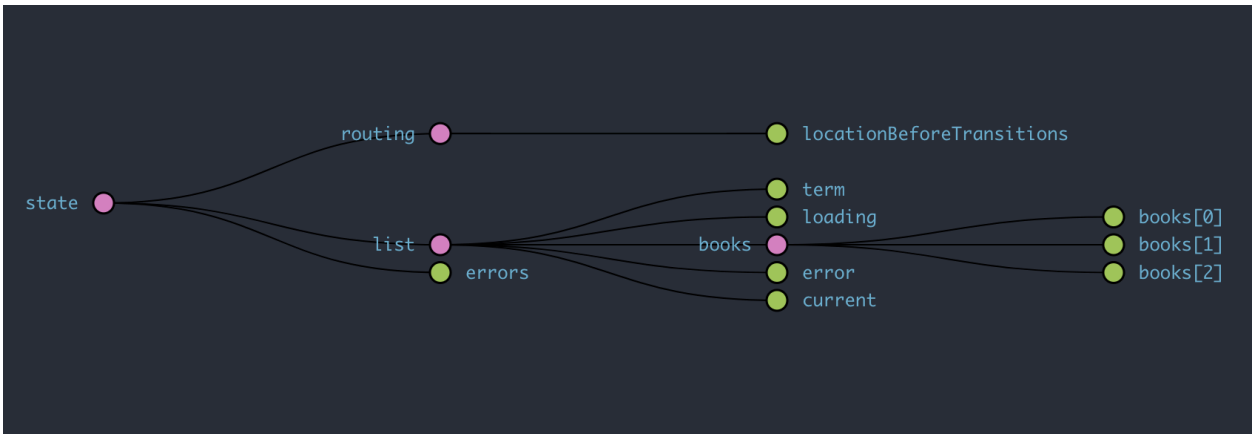
If you have [redux-devtools-extension](installed on your Chrome, you can get a very straightforward understanding of how the data is shaped in our application.

redux-devtools-extension

[redux-devtools-extension](https://github.com/zalmoxisus/redux-devtools-extension) is a fantastic tool that can visualize the `store` data along with the development. You can effortlessly understand what is going on at any time when you interact with the page, say, input something or click a button. It can track every `redux` action and show it in detail on tab of the Chrome dev-tools[10].

Moreover, it even gives you a chance to see in the big picture of how the whole application state is or was back at any point in time when some `action` has occurred. That gives you the opportunity to do the time travel, which is the greatest invention ever in frontend development / debugging.

As you can see, currently the shape is not that ideal since we combined error message with the list itself, and book detail comes from the `list` as well, which is quite confusing I suppose.



That's where we can have some improvement, let's extract the code out and put them where it supposed to be. Firstly we can restructure the folder as:

[10] https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd

```
redux
├── actions
│   ├── actions.js
│   └── actions.test.js
├── reducers
│   ├── books.js
│   ├── detail.js
│   ├── errors.js
│   ├── errors.test.js
│   └── search.js
└── types.js
```

We split reducers to parts that have more dedicated responsibility. For example, `books` would deal with all details about the book list, and detail would represent the detail page.

When given that all error handling code is now in `errors`, the reducer code could be reduced significantly.

```
export default (state = [], action) => {
  switch (action.type) {
    case types.FETCH_BOOKS_SUCCESS:
      return [ ...action.payload ]
    default:
      return state
  }
}
```

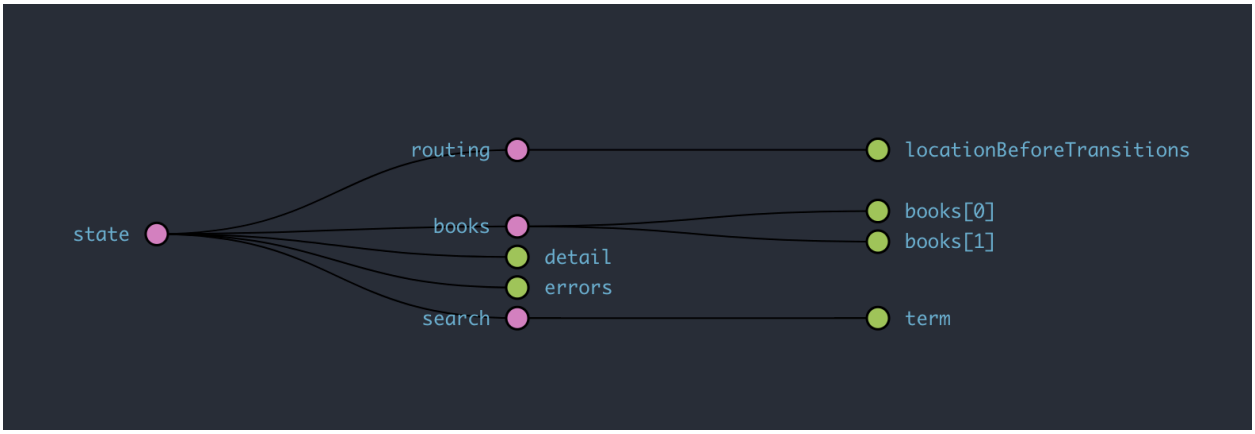
And detail page turns to:

```
export default (state = {}, action) => {
  switch (action.type) {
    case types.FETCH_BOOK_SUCCESS:
      return { ...action.payload }
    default:
      return state
  }
}
```

You could introduce those small reducers all in one in the `store.js` like:

```
import books from './redux/reducers/books'
import detail from './redux/reducers/detail'
import errors from './redux/reducers/errors'
import search from './redux/reducers/search'

const rootReducer = combineReducers({
  routing: routerReducer,
  books,
  detail,
  errors,
  search
})
```

The shape of the application state now became something like this:

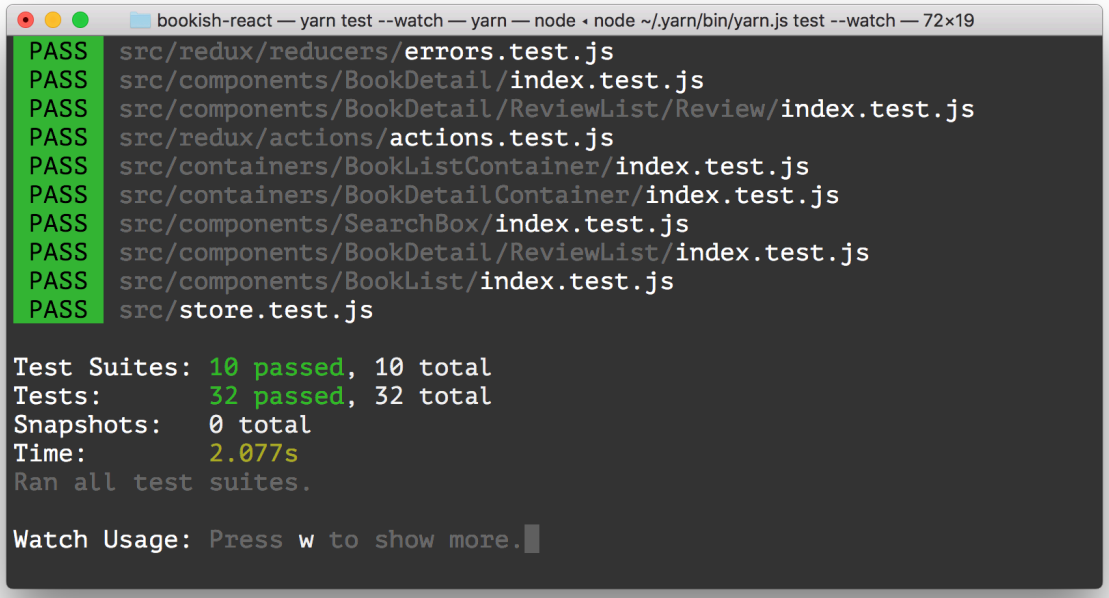


And we don't need `reducers.js` and its test `reducers.test.js` anymore, we can delete them from the codebase. Also, you have to update all the test that assume the store to have a particular state defined as:

```
    it('Save a review for a book', () => {
      const config = {
        headers: { 'Content-Type': 'application/json' }
      }

      const review = {
        name: 'Juntao Qiu',
        content: 'Excellent work!'
      }
      axios.post = jest.fn().mockImplementation(() => Promise.resolve({}))

      const store = mockStore({books: [], search: {term: '' }})
```

```
13
14        return store.dispatch(saveReview(1, review)).then(() => {
15          expect(axios.post).toHaveBeenCalledWith('http://localhost:8080/books/1/revie\
16  ws', JSON.stringify(review), config)
17        })
18      })
```

Just remember to update all the setup steps of `mockStore` to make it align with the new data shape, and then we got an all green tests suite.



Summary

We have briefly discussed some theories about a *Test Driven Development*, along with some tools to apply TDD like `tasking` in Chapter 1. And we have been applying `ATDD` from Chapter 2 to Chapter 5 to build a web application from scratch. We also have covered many best practices when you're working on React applications like *stateless components*, *separate of concerns* (say, split code for calculation and presentation separately).

Moreover, we have introduced a state management mechanism in Chapter 6 and have migrated the whole application to state-management version smoothly by using different *refactoring* techniques.

Additionally, we have introduced how to build a typical `CRUD`(Create Retrieve Update Delete) features in Chapter 7. During the build process, we also have tried to refactor the *end-to-end* tests to achieve both maintainability and readability, in both product code and test code.

To make the test code more readable, we have been applied `Live Document` as part of the practice of `BDD`(an enhanced version of `TDD`) in Chapter 8. And by utilising the *Page Object* and other patterns from Chapter 7, it can improve the quality of test code dramatically.

I hope you have already learnt how to apply `ATDD` in your project appropriately by the example we went through together. Such as how to split big requirements into smaller pieces by *tasking*, how to write *Acceptance Test* first, and then *Unit Tests*, and most importantly - when and how to do the code *refactoring*. For sure, it takes time to practice and masters the techniques demonstrated, but it will payback and improve your productivity significantly.

Happy testing and coding.