Covers RxJS 5

SAMPLE CHAPTER

RxJS IN ACTION

Paul P. Daniels
Luis Atencio

FOREWORD BY Ben Lesh

MANNING

RxJS in Action

by Paul P. Daniels

Luis Atencio

Chapter 9

brief contents

9 Toward testable, reactive programs

This chapter covers

- Understanding functional programming's inherent testability
- Testing asynchronous code with Mocha.js
- Exploring the tools for testing observables
- Understanding the need for using virtual time instead of physical time
- Introducing RxJS schedulers
- Refactoring streams to enhance testability

If you've been in the software industry for any appreciable amount of time, you've likely encountered some form of testing. In production software, there's no escaping the need for tests (or there shouldn't be), whether they target newly written code or a system-wide refactoring. Changes to complex applications can easily produce unforeseen consequences in different paths of execution; it's particularly problematic when multiple developers work with code that they're not intimately familiar with. For instance, when a user types a negative number in the withdraw

field or presses this number rapidly many times, your banking application should handle it gracefully. As you know, in JavaScript, a misspelled variable or a forgotten return statement means that certain execution paths may produce undefined values. These sorts of errors may be obvious or subtle, and no developer—no matter how experienced—is safe from them.

Tests not only help catch programmatic errors and find places where code is brittle, but they also ensure that there's a unified understanding of the requirements. In other words, tests also document the expected behavior of your code.

There are multiple types of testing methods, probably more than we can keep track of, but in this chapter, we'll focus strictly on unit tests. Unit tests are used to create expectations or assertions about the functionality of a single unit of work—a function.

We'll begin this chapter by demonstrating that pure functions are inherently much easier to test than stateful functions, because they have clear inputs and predictable outputs—known as *boundary conditions*. Likewise, observables are functional data types that can be tested in the same manner as pure functions by translating these pure function boundaries to the world of producers and observers. But this isn't always easy. In JavaScript, with so many asynchronous processes to coordinate, testing can be difficult to wrap your head around. You'll learn to use RxJS's observable-based testing to make asynchronous testing easier. With the help of a JavaScript testing framework, Mocha.js, as well as an RxJS instrumentation tool known as a *virtual scheduler*, you can learn to test streams that compose any sort of asynchronous code easily. Toward the end of this chapter, you'll learn about RxJS schedulers. Although they can be powerful, using schedulers in JavaScript applications, especially client-side, is not all that common and intended only for edge cases where the schedulers that accompany the RxJS operators aren't sufficient.

In the end, one of the main advantages of writing your programs functionally is that you've organized the code in such a way that favors testability. Let's start here.

9.1 Testing is inherently built into functional programs

Think back to when you last wrote a set of unit tests for some complex functionality. Do you remember running into any challenges? If this application was written using OOP, most likely you experienced at least one of the following:

- Methods rely on external state that must be properly set up and destroyed for each test.
- Methods are tightly coupled to other modules of the system, making it impossible to test each one independently.
- Your application design lacks a proper dependency injection strategy, so you're unable to properly mock calls to all third-party dependencies.
- Methods are long and complex, so they contain many internal logic paths (lots of `if/else` blocks), which requires you to write multiple tests against the same method just to cover all the flows.

- The order in which tests are run can impact the results that output from the functions under test, so changing the order or possibly commenting out a unit can cause others to fail.

This is by no means an exhaustive list, just some of the more common pain points that we've all experienced while unit testing. Now we don't mean to say that functional tests won't ever have these problems, but what you'll begin to see is that by using pure functions, you can significantly diminish their occurrence.

Pure functions tend to be small in scope, have at most three clearly defined parameters (rarely more), and have a predictable, consistent output—like a black box with simple boundary conditions, as shown in figure 9.1. Moreover, a pure function is deterministic, which means its result is directly determined from the arguments that are passed to it, so half of the testing battle is just coming up with comprehensive sets of inputs. These can be any primitive type like a number or a string, or complex types such as objects and mocks (object impersonators), also shown in figure 9.1.

The other half of the battle is asserting that the return value matches solely the logic behind the function under test, which isn't influenced by what's happening externally. In this book, we'll use Mocha.js as our unit test framework (you can find setup information in appendix A). Let's look at a quick example of its basic usage. Aside from loading the necessary scripts on the page, there's a minor setup step for you to specify the API style you'll use for your assertions and expectations, known as the UI of the test. You'll use the BDD UI, which is the default. If you're using it in the browser, you can use this:

```
mocha.setup({ ui: 'bdd', checkLeaks: true});
```

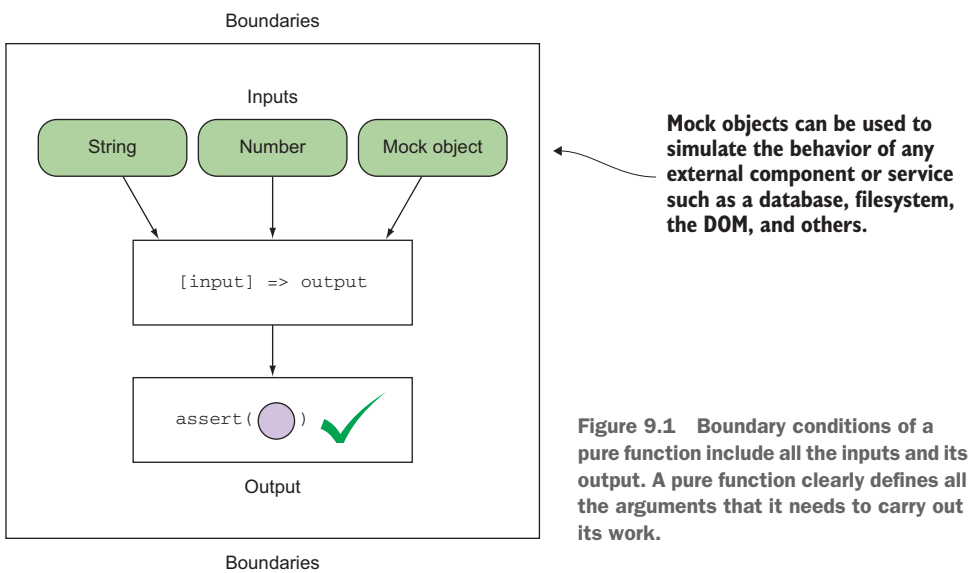


Figure 9.1 Boundary conditions of a pure function include all the inputs and its output. A pure function clearly defines all the arguments that it needs to carry out its work.

On the server, you can run your scripts as

```
mocha --check-leaks --ui tests.js
```

The second parameter is very interesting (as a functional programmer, you'll particularly appreciate this). Mocha also has the ability to detect if global variables "leak" during a single test. A leaked variable is global to the entire test suite with a lifespan that exceeds the test that created it. You may want to leak variables in order to share them with another test, but more often than not, that could cause a programmatic error. For instance, can you spot the leak in this function?

```
function average(arr) {
    let len = arr.length;
        total = arr.reduce((a, b) => a + b);
    return Math.floor(total / len);
}
average([80, 90, 100]) //-> 90
```

Accidentally used ";" instead of "," for multivariable assignment. As a result, the "total" variable is declared globally.

As you know by now, a leaked variable is a side effect that can compromise both the order and the results of your unit tests. Each `it()` block in your tests should be an isolated set of expectations, which is to say that the order and outcome of other tests within the suite should not affect the outcome of any one test. Each test case must start and end with a clean environment, sometimes referred to as a *sandbox*.

All the same principles of pure functions apply as best practices for test development, particularly the property of idempotency that states you should be able to run the tests as many times as needed and always obtain the same results.

Now, let's look at the first function you want to test. Earlier in developing your search widget, you used a function to validate the user's input typed in the search field. Here's that function again to refresh your mind:

```
const notEmpty = input => !!input && input.trim().length > 0;
```

This function is pure because it doesn't rely on any external state or mutate any of the inputs, so it's easy to test. Listing 9.1 shows your first Mocha test. With Mocha, you can create nested suites of behavioral tests. A suite is marked as a `describe` block with a brief description that should tie together the focus of the suite. These blocks can usually be nested so that tests can be further grouped by focus area. At the bottom level is a test case encapsulated within an `it` block; this is where the application logic is actually exercised. Each of these blocks should ideally target a specific aspect of a specific behavior—input validation, in this case.

Listing 9.1 First unit test of a pure function `notEmpty`

```
const expect = chai.expect;
```

Sets up the expect framework

```
describe('Validation', function () {
    it('Should validate that a string is not empty', function() {
        expect(notEmpty('some input')).to.be.equal(true);
```

Asserts the positive use case

```
        expect(notEmpty(' ')).to.be.equal(false);
        expect(notEmpty(null)).to.be.equal(false);
        expect(notEmpty(undefined)).to.be.equal(false);
    });
});
```

Asserts the negative use cases

Finally, to run this unit test you invoke

```
mocha.run();    or      mocha --check-leaks --ui validation.js
```

And everything works as expected:

```
✓ Should validate that a string is not empty
```

CODE SAMPLES Remember that all the code for this chapter can be found in the RxJSinAction GitHub repository, https://github.com/RxJSInAction/rxjs-in-action.

As you can see, testing this pure function was easy, and setup was minimal. Now that you know what a simple Mocha test looks like, let's play with the leak-detection feature for a bit. Running a test for our fishy average function

```
describe('Average numbers', function () {
  it('Leak the variable total', function () {
     expect(average([80, 90, 100])).to.be.equal(90);
   });
});
```

causes the result

```
Error: global leak detected: total
```

identifying exactly which variable caused a side effect. Inadvertently changing `total`, a globally declared variable, because of a subtle code bug could have caused any other tests that depended on it to fail. So as a general rule of thumb, try not to read from or mutate any global state.

Mocha with Chai

Mocha.js is a full-fledged JavaScript testing framework built for both the browser and Node.js. It runs all of your unit tests serially and creates detailed reports. One of the nice features of Mocha is that it allows you to easily plug in any assertion library you want, whether you're familiar with the xUnit assertion APIs like assert.js or other varieties such as expect.js (used previously) and should.js, to name a few. In this book, because we have synchronous as well as asynchronous test requirements, we'll use a flexible API or a domain-specific language (DSL) called Chai.js, which includes support for all the testing APIs mentioned previously. Should.js will be instrumental when running tests involving `Promise`s.

(continued)

Mocha also has great reporting capabilities. It prints out the results as human-readable sentences—allowing you to tell exactly which behaviors are failing in the application—and lets you isolate debugging efforts to a specific region.

One of the main reasons for using Mocha is its ample support for asynchronous testing and promises. Hence, it's the framework with which core RxJS code is tested. More details about installing Mocha can be found in appendix A. To explore the RxJS test suites, you can visit http://reactivex.io/rxjs/test.html.

Undeniably, the world would be a better place if all your code was this easy to unit test (certainly people would be less afraid of it). But asynchronous functions throw a monkey wrench into the whole process, and JavaScript applications are notorious for dealing with lots of asynchronous behavior. So let's talk about how you can use Mocha to test these types of programs.

9.2 Testing asynchronous code and promises

Asynchronous code creates a big wrinkle in your ability to write unit tests. Although it's true that Mocha is designed to run your individual test cases serially (one by one), how can you instruct it to wait for the completion of some long-running computation instead of sweeping through your entire test suite synchronously? In this section, we'll examine two testing scenarios: invoking AJAX requests directly and working with `Promises`.

9.2.1 Testing AJAX requests

The smart search widget we developed in chapter 5 made AJAX requests against the Wikipedia API to suggest potential search results using the RxJS DOM operator called `Rx.Observable.ajax()`. As you can imagine, under the hood, this operator uses the common `XmlHttpRequest` object to communicate with the server. Before you work your way up to testing entire observables, let's focus on testing plain asynchronous calls for now. Consider this simple alternative:

```
const ajax = (url, success, error) => {
  let req = new XMLHttpRequest();
  req.responseType = 'json';
  req.open('GET', url);
  req.onload = function() {
    if(req.status == 200) {
      let data = JSON.parse(req.responseText);
      success(data);
    }
    else {
      req.onerror();
    }
  }
  req.onerror = function () {
```

```
    if(error) {
      error(new Error('IO Error'));
    }
  };
  req.send();
};
```

You'll use Mocha to set up a unit test for this just like before:

```
describe('Asynchronous Test', function () {
   it('Should fetch Wikipedia pages for search term +
        `"reactive programming"', function() {

     const searchTerm = 'reactive+programming';
     const url = `https://en.wikipedia.org/w/api.php?action=query +
                 `&format=json&list=search&utf8=1&srsearch=${searchTerm}`;

     let result = undefined;

     ajax(url, response => {
       result = response;
     });

     expect(result).to.not.be.undefined;
   });
});
```

Sets up initial conditions

Makes the request and assigns the response to the result variable

Asserts the result variable has a value

WATCH OUT: CORS Remember, if you're running any of these examples in the browser, make sure you disable CORS so that you can access the tested endpoints. Otherwise, just use the example directory located at https://github.com/RxJSInAction/rxjs-in-action, which handles these issues for you.

At a glance, this test seems pretty simple. Set up the initial conditions, make the asynchronous request, capture its response, and assert it. Nothing to it, yet running it prints this:

```
AssertionError: expected undefined not to be undefined
```

What happened? To be and not to be? The issue here is that your unit test is not async-aware. In other words, it thinks it can run synchronously and execute every single statement top to bottom, disregarding the latency present in the HTTP request.

Luckily, Mocha provides excellent support for testing functions that execute asynchronously. It's pretty straightforward: provide a function (usually) called `done()` into the callback passed to `it()`, and Mocha will understand that it needs to wait for this function to be called. Instead of running these tests in parallel and printing randomly ordered test reports, it's advantageous that Mocha runs your tests serially and properly waits for one test to finish before proceeding to the next (if you were thinking for a second that the use of a `done` function looks familiar, it's because you've gotten used to the `complete()` function of observers by now). Let's write a test suite that checks for the success and error cases of `ajax()`.

Listing 9.2 Using Mocha/Chai to test an asynchronous function

```
const assert = chai.assert;

describe('Ajax test', function () {
   it('Should fetch Wikipedia pages for search term +
      `"reactive programming"',
      function (done) {
     const searchTerm = 'reactive+programming';
     const url = `https://en.wikipedia.org/w/api.php?action=query& +
                 `format=json&list=search&utf8=1&srsearch=${searchTerm}`;

     const success = results => {
       expect(results)
              .to.have.property('query')
              .with.property('search')
              .with.length(10);
       done();
     };

     const error = (err) => {
        done(err);
     };

     ajax(url, success, error);
   });

   it('Should fail for invalid URL', function (done) {

     const url = 'invalid-url';

     const success = data => {
       done(new Error('Should not have been successful!'));
     };

     const error = (err) => {
       expect(err).to.have.property('message').to.equal('IO Error');
       done();
     };

     ajax(url, success, error);
   });
});
```

Loads the assert style of assertions

Passing the done function instructs Mocha to halt, waiting for the async ajax() function to return.

Sets up the success function and the assertion

In the successful case, you don't expect the call to fail.

Within the same test, includes the error case

In the error case, you don't expect the call to be successful.

Asserts that the failure occurred and that you received the correct error message

The suite in listing 9.2 contains two test cases: one to test the Wikipedia response object returned from invoking a successful AJAX query with matched results, and the other asserting the error condition when no search results match.

As you can imagine, the ability to test asynchronous functions is a necessity for applications involving RxJS. But recall that with promises you have several options when working with these longer-running tasks. You could use the AJAX directly with RxJS:

```
Rx.Observable.ajax(query)
```

Or, if your AJAX function uses `Promises` or a promise-like (deferred) interface (like jQuery's popular `$.get()`), then you can also use

```
Rx.Observable.fromPromise(ajax(query))
```

Using Promises to wrap these types of operations is the more functional approach because it provides an abstraction over the factor of time, which is a form of side effect. Also, many third-party libraries are wrapping their APIs with promises. Let's discuss this a bit more.

9.2.2 Working with Promises

In this section, we'll continue with our running example of invoking the ajax() function, except this time using Promises. As stated before, a Promise is a functional, continuation data type that allows you wrap any long-running operation, so that you can map functions via then() to the eventually created value. It's proven to be so successful that Mocha includes support for working natively with Promises through a Chai extension called chai-as-promised.js and the should.js fluent API (setup information available in appendix A).

Let's start by refactoring ajax() to use Promises. This is simple; just wrap the body of the function within the Promise and delegate the success and error conditions to the Promise's resolve and reject callbacks:

```
const ajax = url => new Promise((resolve, reject) => {
    let req = new XMLHttpRequest();
    req.responseType = 'json';
    req.open('GET', url);
    req.onload = () => {
      if(req.status == 200) {
        let data = JSON.parse(req.responseText);
        resolve(data);
      }
      else {
        reject(new Error(req.statusText));
      }
    };
    req.onerror = () => {
      reject(new Error('IO Error'));
    };
    req.send();
  });
```

Now you're going to tell Chai to use the Promise extensions and load the should.js APIs into your tests. This is a quick setup at the top of the file:

```
chai.use(chaiAsPromised);
const should = chai.should();
```

To use Chai in environments that don't support Node.js-like CommonJS modules (like the browser), you'll need to use Browserify to create the compatible bundle. We've done that for you in the GitHub repo accompanying this book.

You can see that the test in listing 9.3 is similar to listing 9.2. The abstraction provided by the Promise allows the test framework to instrument the result of the test much better. Using the should.js APIs, you can wire up semantically meaningful expectations for Promises such as should.be.fulfilled to assert the call completed and

should.enventually.have to inspect the results. Also, instead of passing done, Mocha expects you to return the Promise object under test to the engine to run the specified expectations.

Listing 9.3 Asynchronous testing with Promises

Instead of using the done() function, you return the Promise to Mocha so that it knows to fulfill the Promise and run the necessary assertions.

```
describe('Ajax with promises', function () {
     it('Should fetch Wikipedia pages for search term +
       `"reactive programming"', function () {

     const searchTerm = 'reactive+programming';
     const url = `https://en.wikipedia.org/w/api.php?action=query& +
                 `format=json&list=search&utf8=1&srsearch=${searchTerm}`;

     return ajax(url)
         .should.be.fulfilled
         .should.eventually.have.property('query')
           .with.property('search')
           .with.length(10);
   });
});
```

Uses the should.js support with Promises

Asserts the eventual value resolved through the Promise

Nothing much changes with this test compared to the previous one, except that you can work directly with the Promise returned from ajax(). It's incredible to see how descriptive and fluent tests can be using Mocha. Now that you've asserted ajax() works as expected, let's see how this function is used within the observable pipeline. The following listing shows a snippet of the search$ observable again.

Listing 9.4 Search stream used in the smart search component

```
const search$ = Rx.Observable.fromEvent(inputText, 'keyup')
  .debounceTime(500)
  .pluck('target','value')
  .filter(notEmpty)
  .do(term => console.log(`Searching with term ${term}`))
  .map(query => URL + query)
  .switchMap(query =>
      Rx.Observable.fromPromise(ajax(query))
          .pluck('query', 'search')
          .defaultIfEmpty([]))
  .do(result => {
    count.innerHTML = `${result.length} results`;
  })
  .subscribe(arr => {
    clearResults(results);
    appendResults(results, arr);
  });
```

Your main area of focus when testing this program

Pay attention to how the code branches off in the call to switchMap(). This additional flow will make your tests complex. Because most of the data flow logic is handled by

the observable itself, which you trust has already been tested extensively, all you need to worry about is testing that your own functions work as expected. In this case, you've tested that `notEmpty()` and `ajax()` work, and now you can test that this entire code block integrated with your functions works as well. Before you can do this, in the next sections, you'll try to split the AJAX stream into its own observable and test that independently. This will drastically simplify your tests and allow your code to be more modular and reusable.

Because observables are also pure functions (you can translate the black box analogy of inputs and output to be producer and consumer, respectively), you should be able to test them with some confidence. You'll need this for the stream projected into `search$` as well. In the next section, you'll explore how to test reactive streams.

9.3 Testing reactive streams

Reactive testing follows a similar format to how you normally test functional programs as described earlier. Because observables are pure functional data types, the transitive property of purity applies, which states that if an observable is made up solely of pure functions, the entire observable sequence is itself pure. Let's begin with a cold observable that synchronously adds the numbers in an array.

Listing 9.5 Testing a stream that adds up all numbers of an array

```
describe('Adding numbers', function () {
  it('Should add numbers together', function () {

    const adder = (total, delta) => total + delta;

    Rx.Observable.from([1, 2, 3, 4, 5, 6, 7, 8, 9])
      .reduce(adder)
      .subscribe(total => {
          expect(total).to.equal(45);
      });
   });
});
```

Notice that because the semantics of observables are designed for asynchronicity with the producer/consumer model, you're able to place all the assertions into the downstream observer, which is intuitive because that's where the outcome of the stream is. Again, this works only with synchronous functions. Here's a similar program using generators:

```
  it('Should add numbers from a generator', function () {

    const adder = (total, delta) => total + delta;

    function* numbers() {
       let start = 0;
       while(true) {
         yield start++;
       }
    }
```

```
    Rx.Observable.from(numbers)
      .take(10)
      .reduce(adder)
      .subscribe(total => {
          expect(total).to.equal(45);
      });
  });
```

And you obtain the same results. It's clear that testing synchronous observables is as simple as testing regular pure functions—you expect cold observables to behave like this. Let's mix it up a bit by injecting a time delay into your tests:

```
it('Should add numbers together with delay', function () {
     Rx.Observable.from([1, 2, 3, 4, 5, 6, 7, 8, 9])
        .reduce((total, delta) => total + delta)
        .delay(1000)
        .subscribe(total => {
            expect(total).to.equal(45);
        });
});
```

Running this code prints out the following:

```
✓ Should add numbers together with delay
```

It worked! But there's a red herring. Although you get the impression the test is passing, the `subscribe()` block or the observer isn't actually executing; it runs after a whole second has passed, and the result is ignored. Try failing the test case by changing the result to some nonsense value:

```
Rx.Observable.from([1, 2, 3, 4, 5, 6, 7, 8, 9])
   .reduce((total, delta) => total + delta)
   .delay(1000)
   .subscribe(total => {
     expect(total).to.equal('non-sense!');
   });
```

Now, instead of passing the test, you expect that Mocha will throw an error and fail. But you see the same outcome as in your test report. What happened? The obvious culprit seems to be that the delay operator introduces something into the test mixture that isn't properly handled by the test. This intuition is correct, and it's at the heart of what you're trying to accomplish with reactive testing. Because you've added an asynchronous time element that isn't being handled by the test, the test reports completion before the asynchronous block has completed running and you get a false positive. You were deceived by RxJS's abstraction over time. Observables make working with latency and time so simple that it seemed as though the operators were executing synchronously to the test. Of course, this isn't the case.

No fear, grab a cup of Mocha and get to it. Here, you'll need to come back to using `done()` with the `it()` callback. Do you recall how similar Mocha's concept of `done()` is to the observer's `complete()`? Try making them the same, as in the following listing.

Listing 9.6 Testing an observable with a delay

```
it('Should add numbers together with delay', function (done) {
     Rx.Observable.from([1, 2, 3, 4, 5, 6, 7, 8, 9])
        .reduce((total, delta) => total + delta)
        .delay(1000)
        .subscribe(total => {
           expect(total).to.equal(45);
         }, null, done);
});
```

Changing this output to anything other than 45 will break the test.

Uses done to signal the completion of the stream and hence the test. Because this code will never produce errors, you skip it by passing null.

Running it now prints this:

```
✓ Should add numbers together with delay (1008ms)
```

The time label in milliseconds next to the output should hint to you that Mocha waited for this test to complete and actually ran the expectations. Armed with the knowledge of how to test asynchronous observables, let's go back to the search stream `search$` in listing 9.4. You can recognize that most of the observable pipeline in this code is synchronous, until this:

```
.switchMap(query =>
      Rx.Observable.fromPromise(ajax(query))
          .pluck('query', 'search').defaultIfEmpty([]))
```

This segment spawns an AJAX request against Wikipedia for search results that match the user's input, which is actually its own observable stream. You can test this function that's being mapped to the source observable and apply the same technique as you did in listing 9.6. The stream function under test this time is

```
query => Rx.Observable.fromPromise(ajax(query))
                .pluck('query', 'search').defaultIfEmpty([])
```

The next listing shows how to test your asynchronous, `Promise`-based observable mapped to the source observable.

Listing 9.7 Testing a promise AJAX call within an observable

Uses done to notify Mocha this will be an asynchronous test

Defines the function under test

```
it('Should fetch Wikipedia pages for search term "reactive programming" +
   `using an observable + promise', function (done) {

   const searchTerm = 'reactive+programming';
   const url = `https://en.wikipedia.org/w/api.php?action=query& +
               `format=json&list=search&utf8=1&srsearch=${searchTerm}`;

   const testFn = query => Rx.Observable.fromPromise(ajax(query))
      .subscribe(data => {
            expect(data).to.have.property('query')
```

```
            .with.property('search')
            .with.length(10);
        }, null, done);
  testFn(url);
});
```

Passes done in place of the completed observer method to signal the end of this observable sequence and hence the end of the test

Calls the function being tested

So far, you've covered lots of ground by testing all the functions that make up your business logic as well as the asynchronous branch of the search component, in isolation. This is certainly the right direction, but you shouldn't have to rebuild or copy and paste a testable version of your observable sequence into your unit tests; that duplicates your efforts. Instead, it's convenient to split these concerns so as not to mix browser-specific details like emitting a DOM event and rendering to the screen with actual data transformation and event processing. Let's refactor the existing observable to be testable, and we'll show how to write reactive code with testing in mind.

9.4 Making streams testable

As visually pleasant as long observable sequences are (at least for two of us), for matters of testability and even sometimes reusability, it's important to separate the observer from the pipeline and the subscription. Decoupling these main parts will allow you to inject any assertions that you need to make, depending on the stream under test. The goal is to not have to modify or rebuild the observable sequence in the application as well as in the unit test and have code duplicated in both areas. Continuing with the same mindset with which you started the chapter, to make this code more testable, you'll split up your functions so that they can be tested independently from the stream, as well as decompose the stream into its three main parts: producer, pipeline, and consumer. This will allow you to separate the pure (testable) part of the stream from the impure. The impure sections involve writing to a database, making actual AJAX calls, or writing to the DOM, all of which should be outside of your scope of test.

Start out with this simple program that generates 10 consecutive numbers every second and performs the sum of all the even numbers:

```
Rx.Observable.interval(1000)
    .take(10)
    .filter(num => num % 2 === 0)
    .map(num => num * num)
    .reduce((total, delta) => total + delta)
    .subscribe(console.log);
```

In order to make this program testable you need to do a few things:

1. Split out the business logic from the observable pipeline.
2. Decouple the consumer and producer and isolate the stream pipeline. This will allow you to inject your assertion code.
3. Wrap the stream into a function that you can call with the proper observer.

By applying these steps to the previous code, this program becomes a more generic set of functions that you can test thoroughly:

```
const isEven = num => num % 2 === 0;
const square = num => num * num;
const add = (a, b) => a + b;

const runInterval = (source$) =>
  source$
     .take(10)
     .filter(isEven)
     .map(square)
     .reduce(add);
```

Separates producer (source) and subscriber from the business logic by making an argument

Notice how you also wrap the stream into a function that can be called from within your test with whatever event producer you want. It could be a literal sequence of numbers, an array, a generator, and others. The function allows you to pass in test input arguments. Without refactoring it this way, if all these functions were embedded into the observable itself as in the original version, you wouldn't have the flexibility to cover all the possible use cases required to run through all paths of this code. This is also much more efficient because you don't need to execute the entire sequence every time. Now, with a more testable version of this stream, let's proceed.

The functions `isEven()`, `square()`, and `add()` are straightforward to test. We'll leave those as an exercise for you and focus on the observable. Because observables are feed-forward, unidirectional flows that rely on side effect–free functions, you can just as easily consider the entire stream as being pure.

Instead of rewriting another version of the same stream in your test, just call it from within your test, provide a producer into it, and place your assertions into the `subscribe` block:

```
it('Should square and add even numbers', function (done) {

     this.timeout(20000);
     runInterval(Rx.Observable.interval(1000))
       .subscribe({
         next: total => expect(total).to.equal(120),
         err:  err   => assert.fail(err.message),
         complete: done
     });
});
```

Increases Mocha's timeout setting to allow the stream to complete

The expectations are wired up in the test, decoupled from the stream code.

The producer and the subscriber are the boundaries of this pure stream. Figure 9.2 highlights the sections of code that got decoupled from the observable pipeline. By ensuring your functions work and trusting in RxJS to do the right thing, you can be confident in your expectations. Also, parameterizing the observer gives you the extra flexibility of directing the output of the stream toward a set of assertions (as in this case), the console, a filesystem, an HTML page, a database, and others.

Running this code prints the following:

```
✓ Should square and add even numbers (10032ms)
```

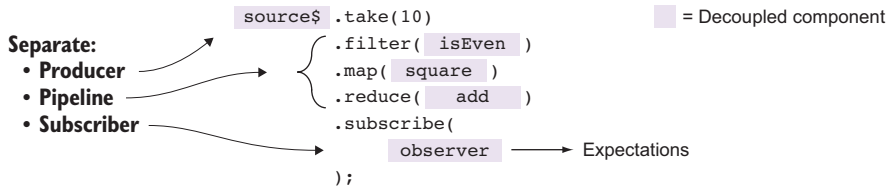


Figure 9.2 The areas from the stream that need to be decoupled in order to gain the maximum test coverage of the entire stream

This single unit test took 10 seconds to run, so you needed to tell Mocha that this test will surpass the default (two-second) timeout. Imagine having test suites with hundreds of these types of tests; it would easily render your CI pipeline useless. Unit tests should be quick; the culprit here is the `interval()` operator (the same would be true for `timer()`). How can you speed up tests of code that has explicit time values? The main reason for adding physical time into your stream is to create the illusion of movement for the user. For example, a panel slides to the right, a counter winds down, a color fades out, and so on. But this isn't important or relevant when running it as a unit test, so instead of refactoring your streams to use a synchronous producer or temporarily commenting out the timers, the proper way to solve this is to add a virtual timer or scheduler.

9.5 Scheduling values in RxJS

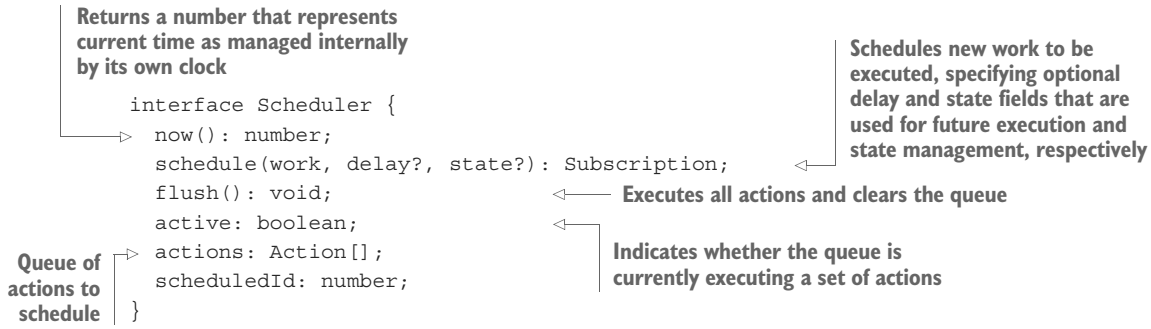
If you're dealing with observable sequences that publish values over an extended period of time, unit testing them can be time consuming. As you know, Mocha will run all your tests serially by design, so it's wasteful for Mocha to be sitting idle waiting for long intervals to complete. In RxJS, time is internally managed using an artifact called a *scheduler*. In this section, we'll briefly introduce this topic and then show how you can apply it to speed up the runtime of your tests. After we've finished introducing schedulers, we'll go back and fix our long-running unit test that uses a delay.

Schedulers control when a subscription starts and when notifications are published. This abstraction allows work to run immediately or in the future without the calling code being aware of it. Remember that RxJS is used to abstract the notion of time? At the heart of all this is a scheduler.

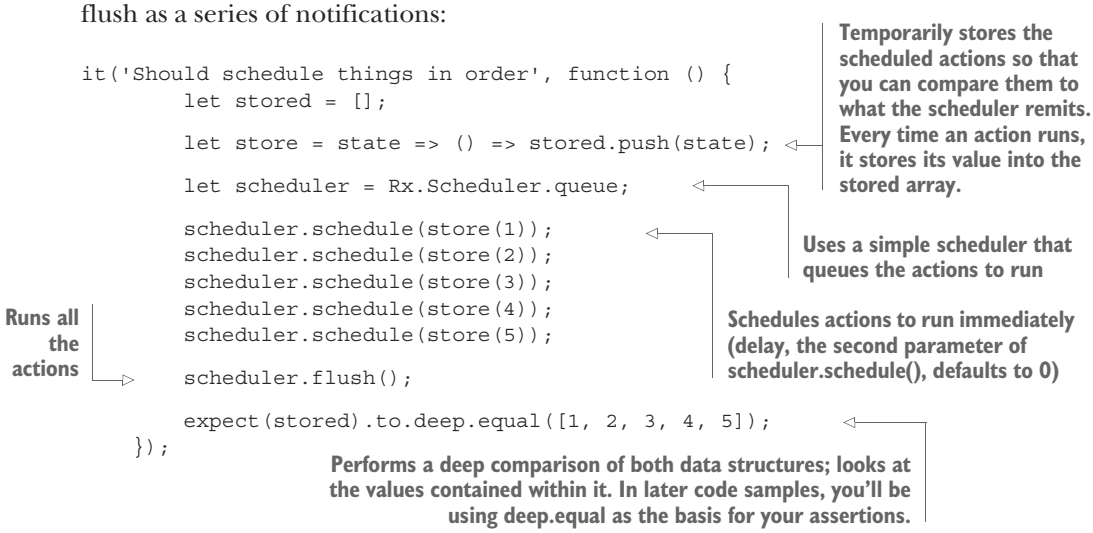
Generally speaking, a scheduler consists of three main parts:

- A data structure that stores all the actions queued to be executed.
- An execution context that knows where the action will be executed: timer, interval, immediately, callback, a different thread (for server-side Rx frameworks), and so on.
- A virtual clock that provides a notion of time for itself. This point will become very important for testing.

RxJS has different types of schedulers, but all abide by the same interface:



Here's how you can use it to schedule a set of actions to run synchronously and then flush as a series of notifications:



Just like observables, schedulers have a similar behavior in that you can push a set of actions that are internally queued or buffered. Every call to `schedule` returns a `Subscription` object that you can use to cancel the subscription if you wish to do so.

Up to this point, we haven't explicitly called out the fact that many of the RxJS factory operators you've seen in this book—`from()`, `generate()`, `range()`, `delay()`, `debounceTime()`, `interval()`, `timer()`, `of()`, and others—have an extra parameter for you to supply a scheduler. All operators make use of a single scheduler, if available. For synchronous data sources, typically a value of `null` is used so that notifications are delivered instantly. On the other hand, two often-used schedulers in RxJS are the `AsapScheduler` and the `AsyncScheduler`, which apply to delayed (`async`) actions (internally RxJS executes and manages these actions in the event loop through `setTimeout()` and `setInterval()`, respectively).

Let's spend some time looking at the effect of having a scheduler control the stream. In the same spirit as the previous code snippet, consider this simple range observable that pushes the values emitted into an external array:

```
it('Emits values synchronously on default scheduler', function () {
     let temp = [];
     Rx.Observable.range(1, 5)
     .do([].push.bind(temp))
     .subscribe(value => {
        expect(temp).to.have.length(value);
        expect(temp).to.contain(value);
     });
});
```

Side effect that pushes value into array temp that lives outside the context of your observable

This stream uses the default scheduler, so this test asserts that each value emitted by `range()` is pushed into `temp` and immediately propagated down to the subscriber. Your expectations check that the size of the array increases with every value and the array contains that value. This stream is fairly simple, and it's behavior that you're accustomed to. Now you're going to change the scheduler used to publish the value to an `AsyncScheduler`, and in the case of most factory operators, you can do this by passing an additional scheduler parameter. By doing so, as shown in the next listing, you change how the stream publishes the values produced by `range()` from synchronous to asynchronous. Let's introduce this new parameter and change your assertions to match this new behavior.

Listing 9.8 Publishing values on an `async` scheduler

Configures the stream to use an async scheduler to proxy the values emitted by the producer. This additional proxying will cause all values to be emitted before the subscription block.

```
it('Emits values on an asynchronous scheduler', function (done) {
   let temp = [];
   Rx.Observable.range(1, 5, Rx.Scheduler.async)
      .do([].push.bind(temp))
      .subscribe(value => {
         expect(temp).to.have.length(value);
         expect(temp).to.contain(value);
      }, done, done);
 });
```

Asserts that the array is growing at every asynchronous emitted value

You can also pass an error handler to done() to indicate an exception condition (the test failed).

Notice that, because it's asynchronous, you need to use the `done()` resolution callback to let Mocha know to wait for all values to be emitted. In sum, just by using a scheduler, you can manipulate how time flows through the stream and control how the events are published. In this case, you overrode the default synchronous event-publishing mechanism to emit asynchronously.

The observeOn() operator

Aside from passing schedulers into the observable factory operations to control how producers emit events, you can also use the `observeOn()` instance operator to transform the emission of events midstream:

```
Rx.Observable.range(1, 5)
      .do([].push.bind(temp))
      .observeOn(Rx.Scheduler.async)
      .subscribe(...)
```

It's important to note that configuring the scheduler midway controls the emission of events downstream only from the point of `observeOn()`, not before. In other words, in this code the execution of `range()` and `do()` still happens synchronously, and the results of those events are then emitted asynchronously to the subscriber. For the examples in this chapter, however, we'll keep it simple and apply schedulers at the factory operator level, just like in listing 9.8.

It's important to note that in server-side implementations of the Rx family, like Rx.Net or RxJava, schedulers can be extremely important to offload heavy processing onto different threads while keeping the active UI thread idle to react to user actions. In the single-threaded world of JavaScript, you'd normally use the default schedulers, and it's rare to choose otherwise. For this reason, in this book we don't cover schedulers in regular application-level code; here's a good resource to start with if you're interested: http://reactivex.io/rxjs/manual/overview.html#using-schedulers. But given their ability to control time, schedulers are very useful, if not necessary, for unit testing asynchronous streams. Let's begin writing some unit tests in virtual time with `Rx.TestScheduler`.

9.6 Augmenting virtual reality

Now that you know what schedulers are, let's circle back to our long-running unit test that used `delay()` and where you also had to set an arbitrarily long timeout value—you want to avoid doing that at all costs! The root of the problem here is that the unit test was using physical time. We mentioned recently that by using schedulers, you could manipulate how these values were emitted, so a physical delay could become a virtual (fake) delay and your tests could run instantly. You can use the `Rx.TestScheduler` class, which is derived from `VirtualTimeScheduler`. This almighty artifact can actually create time!

```
it('Create time from a marble diagram', function () {
   let scheduler = new Rx.TestScheduler();
   let time = scheduler.createTime('-----|');
   expect(time).to.equal(50);
});
```

An empty marble diagram with five time frames

Each time frame counts as 10 units of time (usually milliseconds), so 5 units amounts to 50.

Instead of passing in a set of notification objects or actions, you probably recognize the `"-----"` notation as segments of a marble diagram. In this section, you'll learn how to use the virtual scheduler provided in RxJS and how it's intimately related to the marble diagrams you've seen all along.

9.6.1 Playing with marbles

The `TestScheduler` is driven by the RxJS language of marbles, which, among other characters, primarily contains frames and notifications. In Rx parlance, you use marble diagrams to communicate how a particular operator works with respect to time. Every event that's pushed onto the stream is internally wrapped using a `Notification` object, which transports all of the necessary metadata for a particular event. They're more useful as testing artifacts because they make it easier to represent events that you can extend to add more behavior, such as timestamps or numerical ordering, that you'd want to assert. Here's a simple example of how you'd use notifications directly in your tests:

The dashes represent frames and the letters events (or notifications) that the stream will publish. Every dash represents 10 frames.

```
it('Should parse a marble string into a series of notifications',
   function () {
      let result = Rx.TestScheduler.parseMarbles(
        '--a---b---|',
        { a: 'A', b: 'B' });
      expect(result).deep.equal([
        { frame: 20, notification: Rx.Notification.createNext('A') },
        { frame: 60, notification: Rx.Notification.createNext('B') },
        { frame: 100, notification: Rx.Notification.createComplete() }
      ]);
});
```

The mapping comparisons used in your assertions

The marble diagrams are a convenience method of creating expectations and events. Under the hood, the test scheduler parses out the ASCII text, and from this it generates and queues the actions to perform, which then get published as notifications. The notification is an abstraction of the emission mechanism within RxJS. As you can see from this code, you have three types of emitted events in RxJS: a value, an error, and a completion—yes, this is the observer's API. Even though each type is fundamentally different, you can think of each one more generically as an event, similar to how all DOM events are an abstraction of a single base event type. In other words, you can create a data type to encapsulate an event type regardless of its underlying kind.

Luckily, this internal mechanism can also be abstracted even further by the test scheduler, which uses the high-level Marbles language, kind of like a DSL, to make testing even easier. Consider the `map()` operator we've been using extensively throughout the book. Representing a simple stream that uses it as a marble diagram in ASCII form looks like this:

```
source  --1--2--3--4--5--6--7--8--9--|
map             square => a  * a
subs    --1--4--9--16--25--36--49--64--81--|
```

Let's use the `TestScheduler` to verify that this diagram holds, literally. This class has a rich set of features that helps you create and wire expectations onto observables. Here's a unit test of `map()` using the `square()` function.

Listing 9.9 Testing the `map()` operator

```
function square(x) {
  return x * x;
}

function assertDeepEqual(actual, expected) {
  expect(actual).to.deep.equal(expected);
}

  describe('Map operator', function () {
      it('Should map multiple values', function () {
          let scheduler = new Rx.TestScheduler(assertDeepEqual);

          let source = scheduler.createColdObservable(
                '--1--2--3--4--5--6--7--8--9--|');

          let expected = '--a--b--c--d--e--f--g--h--i--|';

          let r = source.map(square);

          scheduler.expectObservable(r).toBe(expected,
                 { 'a': 1, 'b': 4, 'c': 9, 'd': 16, 'e': 25,
                   'f': 36, 'g':49, 'h': 64, 'i': 81});

          scheduler.flush();
      });
});
```

Helper function that uses Chai to perform a deep.equal assertion of its arguments

Creates an instance of the TestScheduler and passes the comparison function to use

Creates a cold observable from the ASCII diagram

Creates the assertion value placeholders

Source stream with square operation

Uses the scheduler to wire expectations

Flushes the stream, which causes the cold observable to emit its values

In this example, you use two marble diagrams to set up your test case. The first is used to create a source input that behaves like a cold observable. Like the normal diagrams that you saw earlier in the book, each number indicates an event, and each dash indicates a single unit of time. What a single unit of time means for your application is something you'll need to determine. Again, this comes down to how you dilate time in a stream, whether a dash means 1 ms or 1 minute. These marble diagrams carry a lot more meaning than just lines and letters. It turns out that each line segment "-" represents 10 frames of a time period. So, "-----" is a total of 50 frames of the unit of time (typically, each frame represents 10 ms).

The second stream is the expected stream. In order to clarify what's happening, you use a simple associative array that maps the expected values for each notification emitted through the stream.

The test scheduler is extremely powerful because it allows you to test your streams visually. In addition, you're able to test the entire range of observable behaviors, from the construction of the stream, to the emission of events, all the way to the teardown of the stream on completion.

But, admittedly, there are easier ways to test `map()` using a plain Mocha test because it's a synchronous operation and doesn't use time for anything. Remember,

time is what makes asynchronous programming difficult, and that's the problem you're trying to solve.

> **MARBLE SYNTAX** You can find the meaning of all the ASCII symbols of the marble language here: https://github.com/ReactiveX/RxJS/blob/master/doc/writing-marble-tests.md.

These frames are meaningful for operations that are based on time. Let's use the virtual scheduler to test a stream with `debounceTime()`, which would otherwise be complicated and brittle to test because you'd have to rely on adding your own timestamps to emitted notification objects. Let RxJS do this for you.

Listing 9.10 Testing the `debounceTime` operator

```
describe('Marble test with debounceTime', function () {
   it('Should delay all element by the specified time', function () {
      let scheduler = new Rx.TestScheduler(assertDeepEqual);

      let source = scheduler.createHotObservable(
          '-a--------b------c----|');

      let expected = '------a--------b------(s|)';

      let r = source.debounceTime(50, scheduler);
      scheduler.expectObservable(r).toBe(expected);
      scheduler.flush();
   });
});
```

Creates a stream with the first element on the second frame

You debounce with 50 ms (5 frames), the first input after the fifth frame.

Passes in the virtual scheduler into debounceTime()

Running this test creates a stream that simulates (fakes) the effect of `debounceTime()` with a behavior that matches the expected number of frames. As you can see from the diagram, the first notification as a result of emitting a should appear after the fifth frame in `debounceTime(50)`. Now that you know how to fake time, you can speed up that long-running unit test based on `interval()`.

9.6.2 Fake it 'til you make it

Removing time from the stream means that you shift to using the virtual timer's internal clock, which you can wind up by using the time units "-" in the marble diagrams. The `interval(1000)` operator emits consecutive integers every second and is an example of code you might use in production. So in order to simulate your one-second interval, you'll use a 10 ms mocked interval. Now, you know that a scheduler is what's controlling this behavior behind the scenes, so let's take advantage of it to create the mock source as well as the correct expectation.

Listing 9.11 Speeding up `runInterval()` with the virtual time scheduler

```
it('Should square and add even numbers', function () {
   let scheduler = new Rx.TestScheduler(assertDeepEqual);

   let source = scheduler.createColdObservable(
```

Creates an observable that emits values every unit of time (10 ms)

```
        '-1-2-3-4-5-6-7-8-9-|');
    let expected = '-------------------(s-|';
    let r = runInterval(source);
    scheduler.expectObservable(r).toBe(expected, {'s': 120});
    scheduler.flush();
});
```

The expected output is a stream with a single result at the end, given by the reduce operation.

Asserts the end value to be 120

Certainly, refactoring the `runInterval()` stream to make it more testable paid off. You were able to easily inject a virtual cold observable as the producer of events, and everything worked exactly as expected.

9.6.3 Refactoring your search stream for testability

As you've seen in this chapter, RxJS's notion of time is much more sophisticated than a simple callback, and your test cases must reflect that. The simple fact that you can incorporate delays or debouncing into a stream means that the test cases must also understand how time flows and, perhaps more important, must be able to manipulate it when necessary.

Let's finally circle back to the example of your search component, which used a `debounceTime()` operation to prevent flooding the Wikipedia servers with unnecessary search queries. This stream is a bit longer and more complex, but now you have everything you need to properly test it.

If you used a realistic time of 250–500 ms to handle this scenario, it would mean that your test case would likely need to run for at least a second. Although that may not seem like a lot, as we mentioned previously, in a large test suite with several hundred test cases, that could mean minutes to run, which throws continuous integration right out the window. You definitely want to do better than this for your tests if you plan to test as you develop. Now, let's apply what you learned and refactor your existing search stream with an eye for testability.

Testing this in its original state is somewhat difficult. Thus, one of the benefits of plugging this into the RxJS tests is that you can refactor the code based on best practices. So how would you test this?

As before, the focus should be on decoupling the producer, the pipeline, and the subscription so that you can test that your functions are working correctly as integrated into the stream without worrying about how the DOM emits events (producer) and gets updated (observer). You're interested in testing the actual business logic and not the interaction with any other technology.

Just like before, refactoring your stream into a function changes the stream from the hardcoded

```
const search$ = Rx.Observable.fromEvent(inputText, 'keyup')
  .pluck('target','value')
  .debounceTime(500)
  .filter(notEmpty)
  .do(term => console.log(`Searching with term ${term}`))
```

```
  .map(query => URL + query)
  .switchMap(query =>
      Rx.Observable.fromPromise(ajax(query)).pluck('query',
           'search').defaultIfEmpty([]))
  .subscribe(arr => {
    count.innerHTML = `${result.length} results`;
    if(arr.length === 0) {
      clearResults(results);
    }
    else {
      appendResults(results, arr);
    }
  });
```

to a more modular stream composed of a `source$` to which you can pass a virtual observable stream and a search stream `fetchResult$` in charge of making the AJAX call to fetch results from Wikipedia (which you already tested in listing 9.3). By mocking both of these parameters, you can execute the entire stream without worrying about asynchronous callbacks, how the data is produced, and how it's affected by `debounceTime()`. Here's the refactored `search$` function, as implemented in application code:

```
const search$ = (source$, fetchResult$, url = '', scheduler = null) =>
    source$
      .debounceTime(500, scheduler)
      .filter(notEmpty)
      .do(term => console.log(`Searching with term ${term}`))
      .map(query => url + query)
      .switchMap(fetchResult$);
```

This way of encapsulating an observable sequence into its own function is known as an *epic*. Epics will become important in chapter 10, because they will allow you to easily embed RxJS into an overall reactive architecture.

To use the reactive architecture, just call the function with the source and AJAX streams:

```
search$(

    Rx.Observable.fromEvent(inputText, 'keyup')
      .pluck('target','value'),

    query =>
      Rx.Observable.fromPromise(ajax(query))
      .pluck('query', 'search')
      .defaultIfEmpty([])

).subscribe(arr => {
    if(arr.length === 0) {
      clearResults(results);
    }
    else {
      appendResults(results, arr);
    }
  });
```

Furthermore, parameterizing the dependent streams keeps your tests from making outbound calls to the Wikipedia APIs. This is desirable because you don't want your unit test to be compromised by a third-party dependency. In other words, in place of `fetchResults$`, you'll provide an observable with a compatible return type.

This second version doesn't look as fluent as the original, but it's now a lot easier to test, as shown in the next listing. Using the virtual scheduler, you're also able to test how the debouncing works in the stream. Because your debouncing extends to half a second, you use a simple function `frame()` to easily inline any number of time units into your marble diagrams.

Listing 9.12 Unit test main search logic

```
function frames(n = 1, unit = '-') {
    return (n === 1) ? unit :
       unit + frames(n - 1, unit);
}

describe('Search component', function () {
    const results_1 = [
       'rxmarbles.com',
       'https://www.manning.com/books/rxjs-in-action'
    ];

    const results_2 =
         ['https://www.manning.com/books/rxjs-in-action'
    ];

    const searchFn = term => {
       let r = [];
       if(term.toLowerCase() === 'rx') {
          r = results_1;
       }
       else if (term.toLowerCase() === 'rxjs') {
          r =  results_2;
       }
       return Rx.Observable.of(r);
    };

    it('Should test the search stream with debouncing', function () {

       let searchTerms = {
          a: 'r',
          b: 'rx',
          c: 'rxjs',
       };

       let scheduler = new Rx.TestScheduler(assertDeepEqual);
       let source = scheduler.createHotObservable(
            '-(ab)-' + frames(50) +'-c|', searchTerms);

       let r = search$(source, searchFn, '', scheduler);

        let expected = frames(50) + '-f------(s|)';

        scheduler.expectObservable(r).toBe(expected,
           {
```

Helper function to embed any number of time units "-"into a marble diagram

Dummy data for first search action

Dummy data for second search action

Stub search stream that will be projected onto the source observable as part of the search

User input into search stream

Observable that describes the debounce effect. Helper function frames(50) is used to emulate a debounceTime of 500 ms.

Invokes the search stream with all necessary pieces

Creates expectations for the first and second result sets

```
                'f': results_1,
                's': results_2
            });

        scheduler.flush();
    });
});
```

This unit test attempts to simulate a user entering the letters *rx* quickly, producing two results. The stream gets debounced with 500 ms, and finally the third and fourth letters are entered to make *rxjs*. At this moment, the dummy AJAX observable returns only one result to simulate the result set being filtered down. Finally, you've thoroughly unit tested the entire search component.

As Einstein postulated in the early 1900s, all time is relative to the observer. In RxJS, we can transpile this expression to "all time is relative to the scheduler used." In this chapter, we explored how to use the tools provided by RxJS to test reactive applications. In doing so, we also unpacked some concepts surrounding time and its relationship with streams and the RxJS internal notification publishing mechanism. These concepts are important to support the future maintainability of your code. In the next chapter, we'll take reactive programming to new heights. We'll put everything together to create a simple web application that mixes the power of RxJS with a UI component library known as React.

9.7 Summary

- Functional programs are easy to test, given that all functions are pure and have clear signatures.
- Testing asynchronous code can be challenging, and you need to leverage async-aware unit-testing frameworks like Mocha.
- You can combine Mocha with powerful assertion interfaces like Chai.js to create elegant and fluent tests.
- Testing synchronous observables follows the same procedures as testing any pure function.
- Testing asynchronous behavior as well as streams that bend time can be done effectively using the virtual scheduler.
- It's best to make your streams testable and modular. Attempt to keep your business logic separate, as a set of functions, and to decouple a stream from its producer and observer; this will allow you to manipulate its test boundaries to suit the different use cases you want to test.

JAVASCRIPT/WEB DEVELOPMENT

RxJS IN ACTION

Daniels • Atencio

On the web, events and messages flow constantly between UI and server components. With RxJS, you can filter, merge, and transform these streams directly, opening the world of data flow programming to browser-based apps. This JavaScript implementation of the ReactiveX spec is perfect for on-the-fly tasks like autocomplete. Its asynchronous communication model makes concurrency much, much easier.

RxJS in Action is your guide to building a reactive web UI using RxJS. You'll begin with an intro to stream-based programming as you explore the power of RxJS through practical examples. With the core concepts in hand, you'll tackle production techniques like error handling, unit testing, and interacting with frameworks like React and Redux. And because RxJS builds on ideas from the world of functional programming, you'll even pick up some key FP concepts along the way.

What's Inside

- Building clean, declarative, fault-tolerant applications
- Transforming and composing streams
- Taming asynchronous processes
- Integrating streams with third-party libraries
- Covers RxJS 5

This book is suitable for readers comfortable with JavaScript and standard web application architectures.

Paul P. Daniels is a professional software engineer with experience in .NET, Java, and JavaScript. **Luis Atencio** is a software engineer working daily with Java, PHP, and JavaScript platforms, and author of Manning's *Functional Programming in JavaScript*.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/rxjs-in-action

"Important information you need to know in order to become an effective reactive programmer."
—From the Foreword by Ben Lesh
Project lead, RxJS 5

"Covers the subject thoroughly and with great accessibility."
—Corinna Cohn, Fusion Alliance

"All you need to really understand streaming!"
—Carlos Corutto, Globant

"Learn to leverage the power of RxJS to build a reactive and resilient foundation for your applications."
—Thomas Peklak, Emakina CEE

ISBN-13: 978-1-61729-341-2
ISBN-10: 1-61729-341-5
54999
9 781617 293412

$49.99 / Can $65.99 [INCLUDING eBOOK]