Testing Angular Applications

Jesse Palmer
Corinna Cohn
Mike Giambalvo
Craig Nishina

Foreword by
Brad Green, Google

Sample Chapter

MANNING

Anatomy of a Basic Component Unit Test

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { BasicComponent } from './basic.component';

describe('BasicComponent', () => {
  let component: BasicComponent;
  let fixture: ComponentFixture<BasicComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ BasicComponent ]
     })
   .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(BasicComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create an instance of BasicComponent', () => {
    expect(component).toBeTruthy();
  });

  afterEach(() => {
    fixture = null;
    component = null;
  });
});
```

Setup—The setup part of your tests usually will involve three parts: declaring variables, a beforeEach to configure TestBed, and a beforeEach to initialize the variables.

Actual test

Teardown—Use afterEach to destroy variable references.

Testing Angular Applications

by Jesse Palmer
Corinna Cohn
Michael Giambalvo
Craig Nishina

Chapter 9

9 Understanding timeouts

This chapter covers

- Understanding and avoiding the causes of timeout errors in Protractor
- Waiting for specific changes in your application, rather than relying on `browser.sleep()`
- Understanding flakiness and eliminating it with Protractor

Now that you know how to make basic end-to-end tests for Angular apps, let's talk about one of the most frequent issues you might run into. Timeout errors are the most common problems people encounter when using Protractor for the first time. Understanding what causes them and how to fix them requires a clear understanding of how browser tests run. You'll also need to know what Protractor is doing behind the scenes to make tests more reliable by waiting for Angular to be stable while running a test.

In this chapter, we'll explore how to avoid the common timeout-related pitfalls that new Protractor users stumble into. On the way, you'll learn how Angular's

change detection works and how Protractor integrates with it. You'll also learn some advanced techniques for making your own waiting logic. You can find the example code from this chapter at https://www.manning.com/books/testing-angular-applications and http://mng.bz/6k1S.

9.1 Kinds of timeouts

Protractor tests involve many different pieces working together, so different kinds of timeouts are possible. For example, Jasmine will mark your test as failed if it takes too long to complete, and WebDriver will throw an error if a browser command takes too long. For this chapter, we're only concerned about one kind of timeout: the timeout that occurs if Protractor waits too long for Angular to be stable.

What is flakiness?

According to Dictionary.com, *flaky* is slang for eccentric or crazy. When we say a test is flaky, what we mean is that it's nondeterministic—it might fail even though there's nothing wrong with your app. You want to avoid flakiness—if the tests can fail when nothing has changed in the app, then they become less useful.

One potential cause of flakiness is having a test read the DOM of a page while Angular is in the middle of updating it. You could avoid this issue by adding sleep commands after every step in your tests that might cause Angular to update the page, but that would slow down your test runs, and it's not guaranteed to work. Protractor takes a more efficient route and syncs your tests with Angular to help prevent flaky test failures.

Waiting for Angular to be stable prevents your test from interacting with the page while Angular is in the middle of an update, which makes your tests less flaky. But it can cause problems, particularly when you need to test a page that isn't part of an Angular app.

9.2 Testing pages without Angular

Remember from chapter 8 that Protractor intercepts the commands your test sends to WebDriver and automatically waits for Angular to be ready. Figure 9.1 shows this process in detail. This mechanism is one of the biggest advantages of using Protractor, but sometimes it gets in the way.

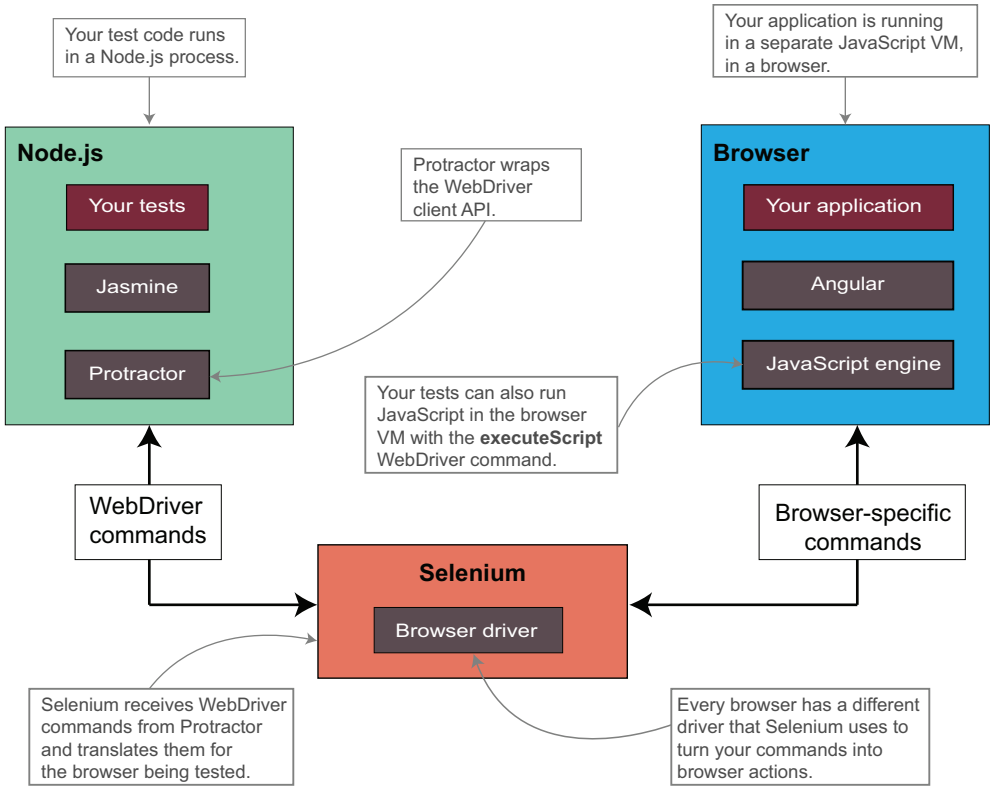

Figure 9.1 How Protractor interacts with WebDriver

One of the first problems new users of Protractor face is writing a test that logs in to their application. If the authentication page is static HTML and not part of the Angular app, then Protractor will throw an `Angular could not be found on the page` error. Protractor expects to see a page that's part of an Angular app and can't tell the difference between one that's not supposed to have Angular and one that's broken.

The example Contacts app doesn't have authentication, but pretend for a minute that it does and that you need to log in before you can run your tests. We've added a fake login page at /assets/login.html. This simple HTML file, which doesn't do anything, is bundled with the Contacts app.

9.2.1 Disabling waitForAngular

Make a test that navigates to the login page using `browser.get('/assets/login.html')`. Running the test produces this error:

```
1) the contact list should find the title
  - Failed: Angular could not be found on the page http://localhost:4200/
    assets/login.html. If this is not an Angular application, you may need
    to turn off waiting for Angular.
```

What does the error mean? As you saw earlier, Protractor automatically intercepts the commands your tests send to WebDriver and inserts commands that communicate with Angular and wait for your application to be ready for testing. When you navigate to a page that isn't an Angular app, Protractor throws an error, because it can't find Angular. To fix the error, you need to tell Protractor not to wait for Angular:

```
it('should be able to log in', () => {
  browser.waitForAngularEnabled(false);
  browser.get('/assets/login.html');
  element(by.css('input.user')).sendKeys('username');
  element(by.css('input.password')).sendKeys('password');
  element(by.id('login')).click();
);
```

Now, suppose you want to test whether clicking the login button redirects to the contact list page. You might add the following before the last line in your test:

```
  const list = element(by.css('app-contact-list tr'));
  expect(list.getText()).toContain('Jeff Pipe');
});
```

You're almost there, but now the test is failing for a different reason. You've turned off waiting for Angular, which means Protractor now has no way to know what the application is doing.

9.2.2 Automatically waiting for Angular

You'll see this error if you run the test from the previous section:

```
1) the contact list should find the title
  - Failed: No element found using locator: By(css selector, app-contact-list
     tr)
```

The problem is that when you're looking for the contact list, your app is still loading. Because you've told Protractor not to wait for Angular, it goes right ahead and looks for the contact list, then fails when it doesn't find it. The fix for this is simple—tell Protractor to start waiting for Angular again after you click the login button. The next listing shows the full, working test.

Listing 9.1 Testing a login page

Disables automatically waiting for Angular

Tests the login page

Re-enables waiting so you can test the application

```
it('should be able to login', () => {
  browser.waitForAngularEnabled(false);
  browser.get('/assets/login.html');
  element(by.css('input.user')).sendKeys('username');
  element(by.css('input.password')).sendKeys('password');
  element(by.id('login')).click();

  browser.waitForAngularEnabled(true);
  const list = element(by.css('app-contact-list tr'));
  expect(list.getText()).toContain('Jeff Pipe');
});
```

Now your test won't wait for Angular on the login page but will go back to waiting for it when you return to the app. Why do you need to explicitly disable waiting? Why couldn't Protractor detect whether an Angular app was on the page and skip waiting if it didn't find one? Protractor has no way of telling the difference between a page that isn't an Angular app and an app that's loading slowly, so you need to let it know that you're intentionally sending it to a non-Angular page. Being explicit about your intentions prevents issues with tests that might be hard to debug, especially when you're relying on Protractor to automatically wait for Angular to finish updating the page. It's important that you know right away if that mechanism isn't working when you expect it to be.

9.2.3 When to use browser.waitForAngularEnabled()

Knowing how and when to enable and disable waiting for Angular, even on pages that are part of an Angular app, is an important part of writing tests using Protractor. But turning off waiting for Angular can have side effects. For example, your tests won't know when Angular is done updating the page, so you might have to use other synchronization methods. One such method is `ExpectedConditions`.

9.3 Waiting with ExpectedConditions

When you tell Protractor not to wait for Angular, you might start seeing test failures if Angular updates the page while your test is running. It's tempting to make the tests pass by sprinkling in `browser.sleep()` commands, but that is a bad idea for a couple of reasons. First, the right amount of time to sleep is arbitrary and hard to know ahead of time. It also slows down your tests, because you end up waiting a fixed amount of time, even if the condition you're waiting for has already occurred. Instead, you can use `browser.wait()` and `ExpectedConditions` to wait for specific conditions in your application to be true, like so:

```
let EC = browser.ExpectedConditions;
browser.wait(EC.visibilityOf($('.popup-title')), 2000,
  'Wait for popup title to be visible.');
```

This will pause your test and repeatedly check whether the given condition is true, up to some specified timeout (two seconds, in this case).

> **NOTE** You should always specify a timeout and an error message when using `browser.wait()`. If you don't specify a timeout, it will keep waiting until your per-test timeout is hit, and having an error message makes timeouts much easier to debug.

Table 9.1 lists all the expected conditions built in to Protractor. You also can combine any number of conditions with `and()`, `or()`, and `not()`, like this:

```
let EC = browser.ExpectedConditions;
let titleCondition =
EC.and(EC.titleContains('foo'),
    EC.not(EC.titleContains('bar')));
browser.wait(titleCondition, 5000,
  'Waiting for title to contain foo and not bar');
```

Table 9.1 Types of expected conditions

Name	When it's true
`alertIsPresent`	An alert dialog is open.
`elementToBeClickable`	The given element is visible and enabled.
`textToBePresentInElement`	The element contains the given string (case-sensitive).
`textToBePresentInElementValue`	The element's `value` attribute contains the given string (case sensitive).
`titleContains`	`document.title` contains the given string (case sensitive).
`titleIs`	`document.title` exactly matches the given string.
`urlContains`	The current URL contains the given string (case sensitive).
`urlIs`	The current URL exactly matches the given string.
`presenceOf`	The element is present in the current page (but may or may not be visible).
`stalenessOf`	The element is no longer part of the page's DOM (the opposite of `presenceOf`).
`visibilityOf`	The element is present in the page, is visible, and has a height and width greater than 0.
`invisibilityOf`	The element is either not present in the DOM or is not visible (opposite of `visibilityOf`).
`elementToBeSelected`	The element is currently selected (if the element is an `<option>` or an `<input>` with a `checkbox` or `radio` type).

Note that you can combine expected conditions and assign them to variables so you can reuse them.

> **WARNING** The `ExpectedConditions` object holds a reference to the `browser` object. Be careful using it in tests that restart or create multiple browsers, and use `browser.ExpectedConditions` instead of `protractor.ExpectedConditions`. Get the reference to `ExpectedConditions` after you restart or fork the browser.

9.3.1 Waiting for the contact list to load

Now that you know the basics of expected conditions, you have another way to make the test from listing 9.1 pass, as shown in the next listing.

Listing 9.2 Using `ExpectedConditions` instead of `waitForAngular`

```
it('should be able to login', () => {
  let EC = browser.ExpectedConditions;
  browser.waitForAngularEnabled(false);
```

```
  browser.get('/assets/login.html');
  element(by.css('input.user')).sendKeys('username');
  element(by.css('input.password')).sendKeys('password');
  element(by.id('login')).click();

  const list = element(by.css('app-contact-list'));
  const listReady = EC.not(
    EC.textToBePresentInElement(list, 'Loading contacts'));
  browser.wait(listReady, 5000, 'Wait for list to load');
  expect(list.getText()).toContain('Jeff Pipe');
});
```

Builds the expected condition

Waits up to five seconds for 'Loading contacts' to go away

As shown, instead of turning on `waitForAngular`, you can wait for the `'Loading contacts'` text to go away. There's no single right answer here—use whichever method is more readable and maintainable for your tests. But expected conditions are a helpful tool to have, especially when dealing with animations, as you'll see in the next section.

9.3.2 Testing a dialog

Another good time to use an expected condition is when you need to wait, like when you're opening a dialog. Figure 9.2 shows a dialog from the Contacts app. On the detail page for a contact, a button (circled in red) opens a dialog that shows a feed of that contact's social media updates.

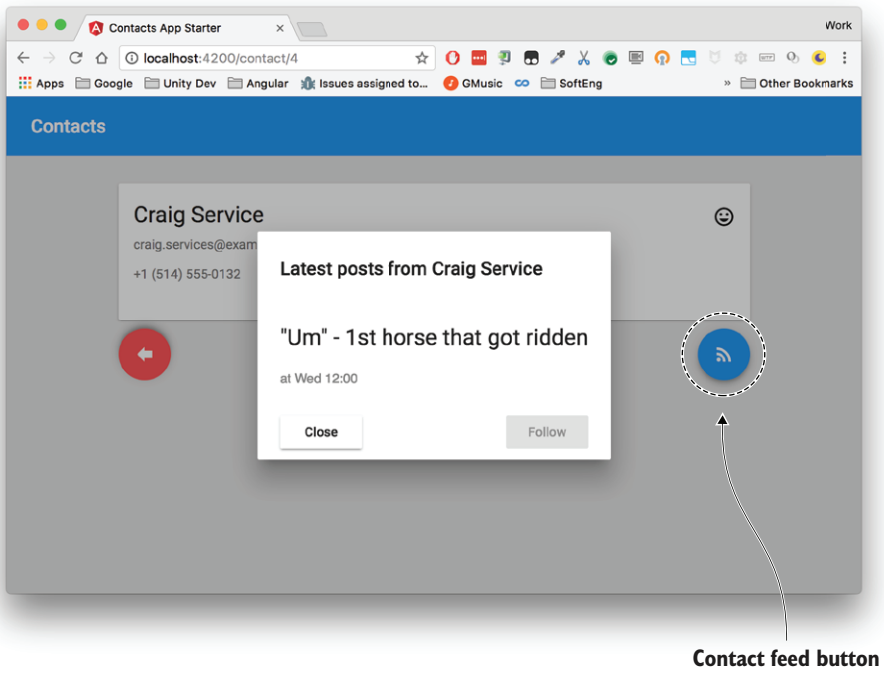

Figure 9.2 The social media feed dialog of a contact

When you click the feed button, the dialog animates opening. When you click Close, it animates fading away briefly before closing. These animations can be problematic when you try to test the dialog, as the following listing shows.

Listing 9.3 Testing the feed dialog with `waitForAngular`

```
describe('feed dialog', () => {
  beforeEach(() => {
    browser.get('/contact/4')
  });

  it('should open the dialog', () => {
    browser.waitForAngularEnabled(true);
    let feedButton = element(by.css('button.feed-button'));
    feedButton.click();

    let dialogTitle = element(
      by.css('app-contact-feed h2.mat-dialog-title'));
    expect(dialogTitle.getText())
      .toContain('Latest posts from Craig Service');

    let closeButton = element(by.css('button[mat-dialog-close]'))
    closeButton.click();

    expect(dialogTitle.isDisplayed()).toBeFalsy();
  });
});
```

Make sure that waitForAngular is turned on

You should see the title of the feed dialog.

Closes the dialog

The title should go away when the dialog closes.

This is a simple test—it clicks the button to open the feed dialog, verifies that the expected title of the dialog is visible, and clicks the Close button. Unfortunately, when you run this test, you see this error:

```
1) contact detail feed dialog should open the dialog
  - Expected true to be falsy.
```

The last expectation fails, because after you click the Close button, you can still see the dialog title while the closing animation runs. Although Protractor is waiting for Angular to finish updating the page, it won't wait for the closing animation to end. This is the kind of situation where you might need to use expected conditions.

9.3.3 Waiting for elements to become stale

Let's fix this test by using expected conditions instead of relying on `waitForAngular`.

Listing 9.4 Testing the feed dialog with expected conditions

```
describe('feed dialog', () => {
  let EC;

  beforeEach(() => {
    browser.get('/contact/4');
    EC = browser.ExpectedConditions;
  });

  it('should open the dialog with expected conditions', () => {
    browser.waitForAngularEnabled(false);

    let feedButton = element(by.css('button.feed-button'));
    browser.wait(EC.elementToBeClickable(feedButton),
      3000, 'waiting for feed button to be clickable');
    feedButton.click();

    let dialogTitle = element(
      by.css('app-contact-feed h2.mat-dialog-title'))
    browser.wait(EC.visibilityOf(dialogTitle),
      1000, 'waiting for the dialog title to appear');
    expect(dialogTitle.getText())
      .toContain('Latest posts from Craig Service');

    let closeButton = element(by.css('button[mat-dialog-close]'))
    closeButton.click();
    browser.wait(EC.stalenessOf(dialogTitle),
      3000, 'wait for dialog to close');
    expect(dialogTitle.isPresent()).toBeFalsy();
  });
});
```

Waits for the feed button to be clickable

Waits for the dialog title to be visible

Waits for the dialog title to be removed from the page

This test passes because you wait for the dialog title to become stale before the last expectation. In WebDriver tests, a stale element is one that you may have a reference to, but that was removed from the page. In this case, the title of the dialog box becomes stale because the closing animation has finished and the dialog has been removed from the page. Elements that you remove from the page with `*ngFor` or `*ngIf` also would become stale.

This test disables `waitForAngular` and relies on expected conditions entirely, but you also can combine the two techniques. For example, you could have made the test from listing 9.3 pass by adding `browser.wait(EC.stalenessOf(dialogTitle))` before the expectation and leaving `waitForAngular` enabled. Either way is fine—the important thing is that your tests reliably do the same thing each time they run.

9.4 Creating custom conditions

Expected conditions are powerful, but sometimes they aren't enough for your needs. Instead of waiting for an element to be present or text to be visible, you might want to wait for a more complicated condition to be true. For example, you might need to wait until a certain number of elements match a CSS selector. Or, perhaps a single selector can't describe the set of elements you're waiting for. In cases that are too hard to express with expected conditions, you can use `browser.wait` with a custom condition.

9.4.1 Using browser.wait

The feed dialog from figure 9.2 will update automatically with new posts from the contact. For testing purposes, it shows a new update after a random delay, on average every five seconds. Say you add a feature that the Follow button will be enabled only when the contact has made two or more posts. In the following listing, you can see a test that verifies that feature.

Listing 9.5 Using `browser.wait` with a custom condition

```
describe('feed dialog', () => {
  beforeEach(() => {
    browser.get('/contact/4');
  });

  it('should enable the follow button with more than two posts', () => {
    let feedButton = element(by.css('button.feed-button'));
    feedButton.click();

    let followButton = element(by.css('button.follow'))
    expect(followButton.isEnabled()).toBeFalsy();
    let moreThanOnePost = () => {
      return element.all(by.css('app-contact-feed mat-list-item')).count()
        .then((count) => {
          return count >= 2;
        })
    };
    browser.wait(moreThanOnePost, 20000, 'Waiting for two posts');

    expect(followButton.isEnabled()).toBeTruthy();
  });
});
```

Verifies that the Follow button is initially disabled

Counts the number of mat-list-items and returns true if there are two or more

Waits until contact makes two or more posts

Verifies that the Follow button is enabled

The first argument to `browser.wait` is a function that will run repeatedly until either it returns true or the timeout is elapsed. In the example, a function looks for all elements that match the `'app-contact-feed mat-list-item'` selector, which will match each

post in the feed. Because Protractor needs to send a request to the browser driver to inspect the page, the result of `element.all(...).count()` is a promise that's resolved with the number of elements that match the selector instead of a number. You then chain this promise with a `.then()` block that returns a Boolean, which is true if the count is greater than or equal to two.

This is similar to how expected conditions work. The expected conditions built in to Protractor (from table 9.1) are functions that inspect the page and return promises that are true when the condition is met. Listing 9.5 is an example of how you can create your own conditions if you need to.

9.4.2 Getting elements from the browser

WebDriver converts DOM elements returned from the browser via a `browser.executeScript` call into instances of WebElement classes that you can use in your tests. So, instead of using element finders, you can write custom JavaScript that will run in the browser and return the elements you're looking for. Here's the test from listing 9.5 but using a custom element finder.

Listing 9.6 Retrieving elements with a custom finder

Function that runs in the browser and returns elements

```
it('should enable the follow button (custom finder)', () => {
    let feedButton = element(by.css('button.feed-button'));
    feedButton.click();

    let followButton = element(by.css('button.follow'))
    expect(followButton.isEnabled()).toBeFalsy();

    function findAllPosts() {
      return document.querySelectorAll('app-contact-feed mat-list-item')
    }

    browser.wait(() => {
      return browser.driver.executeScript(findAllPosts)
        .then((posts: WebElement[]) => {
          return posts.length >= 2;
        })
    }, 20000, 'Waiting for two posts');

    expect(followButton.isEnabled()).toBeTruthy();
  });
```

Using the custom element finder in browser.wait

Waits at most 20 seconds before timing out

The test in listing 9.6 is the same as the one in 9.5. However, instead of using an element finder, it uses a JavaScript function that runs in the browser and returns an array of WebElements. Although this example may seem trivial, it shows how you can write

custom JavaScript that extracts an arbitrary collection of DOM elements from the page. Remember that your tests are running in Node.js, but they can still execute JavaScript in the browser. The `findAllPosts()` function runs in the browser, but you can use the result it returns in your Node.js-based Protractor tests.

9.5 Handling long-running tasks

The feed dialog in the contact detail page (figure 9.3) continuously updates with new posts. For demonstration purposes, the example app does this with an observable that produces an infinite stream of random posts.

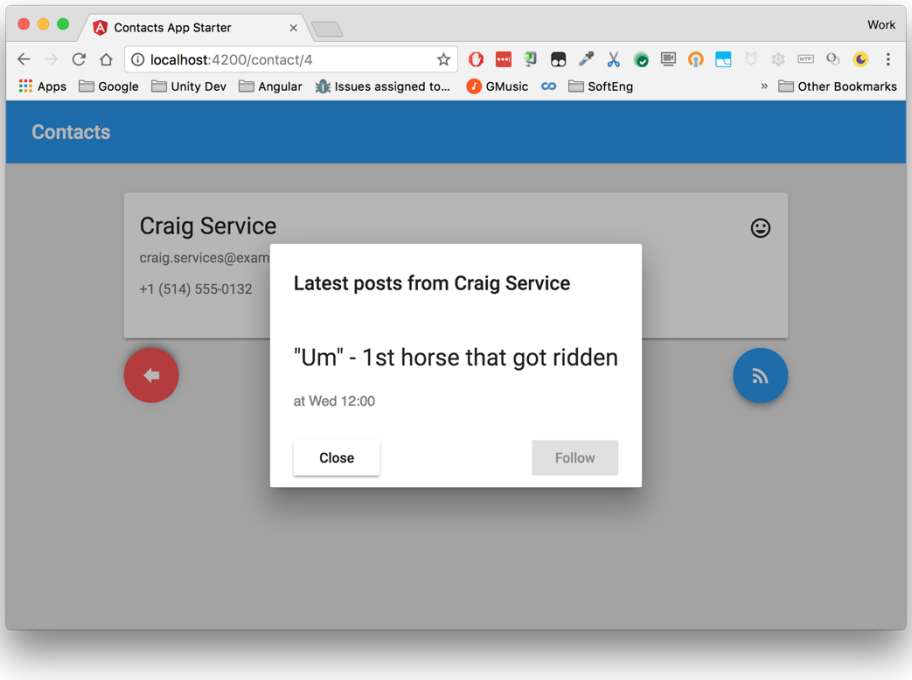

Figure 9.3 The social media feed dialog of a contact.

The following listing is the implementation of the service that the feed dialog uses. In a real application, you would make an HTTP call to get the posts. But the real service could easily have the same interface and return an observable stream of posts.

Listing 9.7 A service that creates a random stream of posts

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Rx';
import { FEED_UPDATES } from './mock-updates';

@Injectable()
export class ContactFeedService {
  constructor() { }
```

```
  public getFeed() {
    const updateId = Math.floor(Math.random() * FEED_UPDATES.length));
    return Observable.interval(500)
      .map((x) => Math.random() * 2 + 2)
      .concatMap((x) => Observable.of(x).delay(x * 1000))
      .map((x) => FEED_UPDATES[updateId]);
  }
}
```

Transforms the observable stream into a stream of randomly delayed events

Randomly picks a string from FEED_UPDATES and puts it in the stream

`FEED_UPDATES` is an array of strings. The feed dialog component subscribes to this observable, as you can see in the following listing.

Listing 9.8 Testing the contact feed dialog

```
import {Component, OnInit, OnDestroy, Optional, Inject} from '@angular/core';
import {MdDialogRef, MD_DIALOG_DATA} from '@angular/material';
import {ContactFeedService} from '../shared/services/contact-feed.service';
import {Subscription} from 'rxjs/Subscription';

@Component({
  selector: 'app-contact-feed',
  templateUrl: './contact-feed.component.html',
  styleUrls: ['./contact-feed.component.css']
})
export class ContactFeedDialogComponent implements OnInit, OnDestroy {
  sub: Subscription;
  updates: string[] = [];
  name: string;
  closeDisabled = true;

  constructor(public dialogRef: MdDialogRef<ContactFeedDialogComponent>,
      private feed: ContactFeedService,
      @Optional() @Inject(MD_DIALOG_DATA) data: any) {
    this.name = data.name;
  }

  ngOnInit() {
    this.closeDisabled = false;

    this.sub = this.feed.getFeed().subscribe((x) => {
      this.updates.push(x);
      if (this.updates.length >= 4) {
        this.updates.shift();
      }
    });
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }
}
```

Subscribes to feed updates and pushes them into the updates property

Cleans up the subscription when the component is destroyed

Unfortunately, the tests from listings 9.5 and 9.6 (which have `waitForAngular` enabled) will time out when trying to test this dialog:

```
1) feed dialog should enable the follow button with more than two posts using
     executeScript
  - Error: Timeout - Async callback was not invoked within timeout specified
     by jasmine.DEFAULT_TIMEOUT_INTERVAL.
```

Protractor's `waitForAngular` hooks into the same method that Angular uses to run change detection and update template bindings. That's how Protractor knows that Angular is done updating the page, but it means that by default, Protractor will wait until all asynchronous tasks that could update the page have finished. The contact feed dialog in the example polls forever, so Angular times out, because there's always a pending task that can update the page.

9.5.1 Using expected conditions

You could disable waiting for Angular. But if you want to avoid flaky tests, you'll need to use expected conditions to wait after every action that can cause the page to update. That's what the test from listing 9.4 did. Here it is again as a reminder.

Listing 9.9 Testing the feed dialog with expected conditions

Disables waitForAngular

You need to wait for the initial page load.

```
it('should open the dialog with expected conditions', () => {
  browser.waitForAngularEnabled(false);
  let feedButton = element(by.css('button.feed-button'));
  browser.wait(EC.elementToBeClickable(feedButton),
    3000, 'waiting for feed button to be clickable');
  feedButton.click();

  let dialogTitle =
      element(by.css('app-contact-feed h2.mat-dialog-title'));
  browser.wait(EC.visibilityOf(dialogTitle),
    1000, 'waiting for the dialog title to appear');
  expect(dialogTitle.getText())
      .toContain('Latest posts from Craig Service');

  let closeButton = element(by.css('button[mat-dialog-close]'))
  closeButton.click();
  browser.wait(EC.stalenessOf(dialogTitle), 3000,
      'wait for dialog to close');
  expect(dialogTitle.isPresent()).toBeFalsy();
});
```

When you click the feed button, you'll need to wait for the dialog to show.

Clicking the close button also requires a wait.

Waiting after every action that could cause a page update can be a drag, and it makes the test harder to read. It would be better if you could write a test that used Protractor's automatic `waitForAngular` behavior, but doing so will require understanding how browsers run asynchronous code, and how Angular knows when to update the page.

To get there, you'll need to know more about zones and how JavaScript operates. The first step to learning about zones is to understand how the browser event loop works.

9.5.2 The browser event loop

JavaScript is single-threaded, meaning it does one thing at a time. Somewhere inside your browser is an event loop that looks something like this:

```
while(true) {
  event = waitForNextEvent()
  doJavaScriptThings(event);
  doBrowserThings(event);
}
```

In this example, `doBrowserThings()` refers to the work the browser does outside of your app's JavaScript—rendering the page, doing I/O, and so on. An event can be something like a timer firing, a mouse click event, or an XHR request changing in status. These events create tasks in the JavaScript VM, and three kinds of tasks are possible: microtasks, macrotasks, and eventtasks (see table 9.2).

Table 9.2 Types of tasks

Task type	When it runs
microtask	Run immediately, before the browser does any rendering or I/O. `Promise.resolve()` will schedule a microtask.
macrotask	Guaranteed to run at least once and in the same order that they're scheduled. Macrotasks run interleaved with browser rendering and I/O and are scheduled by `setTimeout` or `setInterval`. After a macrotask finishes, all microtasks are run before control passes back to the browser.
eventtask	Run in response to events (for example, `addEventListener('click', eventCallback)` or XHR state change). Unlike macrotasks, eventtasks might never run.

All microtasks run before control of the event loop goes back to the browser. Running a microtask might add more microtasks to the queue (for example, by making a `Promise.resolve()` call). Once the microtask queue is empty, control passes back to the browser so it can render the page, perform I/O, and wait for the next eventtask or macrotask to occur. The situation is a little more complicated than the simple `while(true)` loop we covered earlier. Now that you know a bit more about how browsers work, let's consider how this relates to Angular.

9.5.3 What happened to $timeout?

If you've used AngularJS, you might remember the `$timeout` service. When doing asynchronous work in AngularJS, instead of using `window.setTimeout()` or `XMLHttpRequest()` directly, you needed to use the special AngularJS services `$timeout` and `$http`. These services were wrappers around the native browser calls that would make sure change detection ran after the asynchronous task was done, so the content of your page would update if your model changed.

You don't need these special services in Angular. Instead, Angular uses a library called Zone.js to run your application's asynchronous tasks in a context called the Angular zone. Zone.js does this by patching all the browser APIs that create async calls with hooks that track which zone that task is running in. That's how Angular knows when an async callback started by your app occurs and is able to run change detection after it, which removes the need for `$timeout`.

DEFINITION A *zone* is an execution context that persists across async tasks—sort of like thread-local storage in Java, but for async tasks.

9.5.4 Highway to the Angular zone

Protractor knows about the Angular zone; it knows when tasks are pending that might cause a change detection. When you enable `waitForAngular`, Protractor will cause all of your WebDriver commands to wait until there are no more tasks pending in the Angular zone. Let's look at a simple example of running asynchronous tasks in a browser.

Assume that the code in figure 9.4 is running in the Angular zone. In this example, Protractor would wait forever, because the `pollForever()` function (in the lower-left red box) is constantly creating a task in the Angular zone.

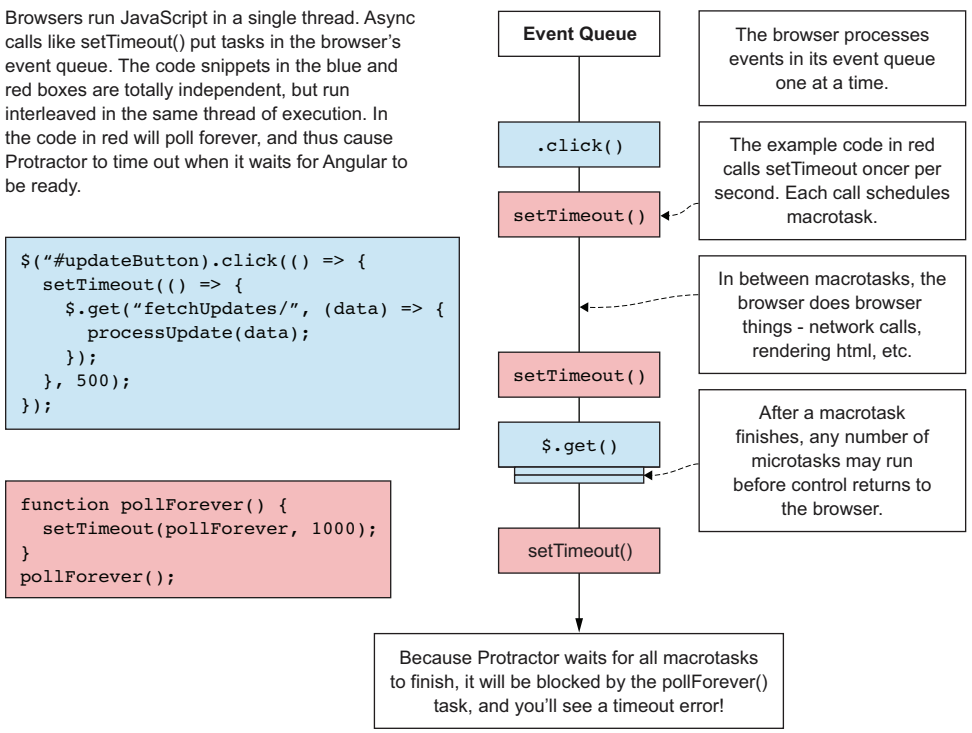

Figure 9.4 Async tasks running in a browser

NOTE A color version of this image is available in the electronic versions of this book, available free to purchasers at www.manning.com.

If you want to avoid this fate of waiting forever, you can move that code inside a call to `NgZone.runOutsideAngular()`, as in figure 9.5. Then `pollForever()` won't trigger a change detection when it runs, and Protractor won't wait for it to finish.

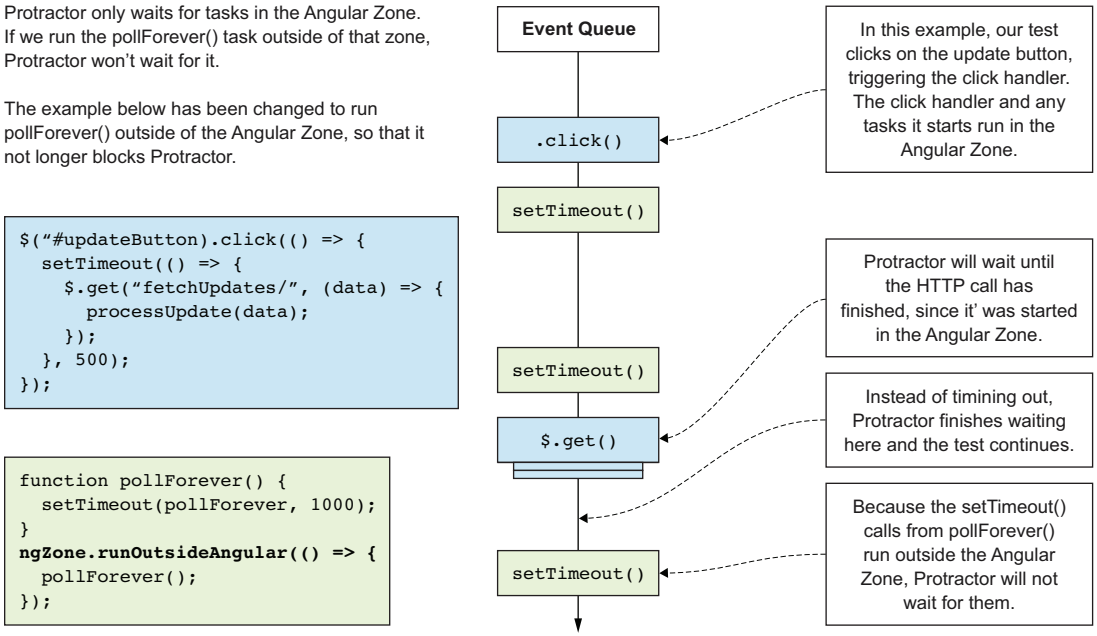

Figure 9.5 Running a polling task outside the Angular zone

This long-running task now runs in a way that won't cause Protractor to wait until the test times out. Next you'll apply this same technique to the Contacts app example and use it to fix the test.

9.5.5 *Fixing the test*

Now you know why your test was timing out earlier. The updates from the `ContactFeedService` were scheduling async tasks in the Angular zone, and because it's a continuous stream of tasks, Protractor will wait until the test times out. You could turn off `waitForAngular` and use expected conditions, but you also could fix the test by changing the `ContactFeedDialogComponent`, as in the following listing.

Listing 9.10 Using `runOutsideAngular` in `ContactFeedDialogComponent`

```
import {Component, OnInit, OnDestroy, NgZone, Optional, Inject} from
    ➥'@angular/core';
import {MdDialogRef, MD_DIALOG_DATA} from '@angular/material';
import {ContactFeedService} from '../shared/services/contact-feed.service';
import {Subscription} from 'rxjs/Subscription';
```

```
@Component({
  selector: 'app-contact-feed',
  templateUrl: './contact-feed.component.html',
  styleUrls: ['./contact-feed.component.css']
})
export class ContactFeedDialogComponent implements OnInit, OnDestroy {
  sub: Subscription;
  updates: string[] = [];
  name: string;
  closeDisabled = true;

  constructor(public dialogRef: MdDialogRef<ContactFeedDialogComponent>,
      private feed: ContactFeedService,
      private zone: NgZone,
      @Optional() @Inject(MD_DIALOG_DATA) data: any) {
    this.name = data.name;
  }

  ngOnInit() {
    this.closeDisabled = false;

    this.zone.runOutsideAngular(() => {
      this.sub = this.feed.getFeed().subscribe((x) => {
        this.zone.run(() => {
          this.updates.push(x);
          if (this.updates.length > 4) {
            this.updates.shift();
          }
        });
      });
    });
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }
}
```

Runs the subscription outside the Angular zone, so it won't block Protractor

Adds the update in the Angular zone, so the change propagates to the page

Injects NgZone into the component

Now that the subscription is created outside of the Angular zone, it won't block Protractor. But you need to apply the update within the Angular zone so that Angular will know about the change to your model and will update the page. With only a small change to the component, the test passes! Protractor still will wait while each update is rendered in the dialog, but now it won't time out waiting for the stream of updates to finish.

Summary

- Browser tests consist of three components: your tests running in Node.js, a Selenium WebDriver server, and your application running in a browser. Protractor synchronizes your tests with your application by waiting for Angular to finish updating the page.
- Sometimes you need to disable waiting for Angular in your tests.
- Use expected conditions instead of `browser.sleep`. Tests are more reliable when they wait for a specific condition to be true, rather than pausing for an arbitrary amount of time. Protractor has many expected conditions you can use out of the box.
- If the available expected conditions don't meet your needs, you can make your own using `browser.wait`.
- Angular uses Zone.js to watch for async tasks that might cause the page to change. Protractor also uses Zone.js to wait until every async task that might modify the page has finished.
- Sometimes you'll need to change your application to run certain async tasks outside the Angular zone. If you don't, you might end up blocking Protractor from testing your page.

Anatomy of Basic End-to-End Tests

Sample Page Object File

```
import { browser, by, element } from 'protractor';

export class AppPage {
  navigateTo() {
    return browser.get('/');
  }

  getParagraphText() {
    return element(by.css('app-root h1')).getText();
  }
}
```

You should use the page object file to select elements. This makes your tests cleaner and helps reduce duplicate code.

Sample Test File

```
import { AppPage } from './app.po';

describe('Basic App', () => {
  let page: AppPage;

  beforeEach(() => {
    page = new AppPage();
  });

  it('should display welcome message', () => {
    page.navigateTo();
    const expectedText = 'Welcome to app!';
    const resultText = page.getParagraphText();
    expect(resultText).toEqual(expectedText);
  });

  afterEach(() => {
    page = null;
  });
});
```

Setup—Initializes the page object variable before each test executes.

Actual test that uses the page object for navigation and gets text

Teardown—Use afterEach to destroy the page object variable reference after each test executes.

WEB DEVELOPMENT/TESTING

Testing Angular Applications

Palmer • Cohn • Giambalvo • Nishina

Don't leave the success of your mission-critical Angular apps to chance. Proper testing improves code quality, reduces maintenance costs, and rewards you with happy users. New tools and best practices can streamline and automate all aspects of testing web apps, both in development and in production. This book gets you started.

Testing Angular Applications teaches you how to make testing an essential part of your development and production processes. You'll start by setting up a simple unit testing system as you learn the fundamental practices. Then, you'll fine-tune it as you discover the best tests for Angular components, directives, pipes, services, and routing. Finally, you'll explore end-to-end testing, mastering the Protractor framework, and inserting Angular apps into your continuous integration pipeline.

What's Inside

- Getting to know TypeScript
- Writing and debugging unit tests
- Writing and debugging end-to-end tests with Protractor
- Building continuous integration for your entire test suite

This book is for readers with intermediate JavaScript skills.

Jesse Palmer is a senior engineering manager at Handshake. **Corinna Cohn** is a single-page web application specialist. **Mike Giambalvo** and **Craig Nishina** are engineers at Google.

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

"Provides guidance on the overall strategy for how to think about testing on your projects to get the best return on your investment."

—From the Foreword by Brad Green, Engineering Director for Angular at Google

"A must-have if you want to learn how to test your Angular applications correctly."

—Rafael Avila Martinez
Intersys Consulting

"Essential to development shops delivering products built on the popular Angular framework."

—Jason Pike
Atlas RFID Solutions

"Developers of all levels will benefit from the material covered in this book."

—Jim Schmehil
National Heritage Academies

ISBN-13: 978-1-61729-364-1
ISBN-10: 1-61729-364-4
54499
9 781617 293641

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]