



{ testing } angular

a.guide.to.
robust.angular.applications

+ mathias schäfer



Table of Contents

1. [Introduction](#)
2. [Target audience](#)
3. [Terminology](#)
4. [Testing principles](#)
 1. [What makes a good test](#)
 2. [What testing can achieve](#)
 3. [Tailoring your testing approach](#)
 4. [The right amount of testing](#)
 5. [Levels of testing](#)
 1. [End-to-end tests](#)
 2. [Unit tests](#)
 3. [Integration tests](#)
 6. [Distribution of testing efforts](#)
 7. [Black box vs. white box testing](#)
5. [Example applications](#)
 1. [The counter Component](#)
 2. [The Flickr photo search](#)
6. [Angular testing principles](#)
 1. [Testability](#)
 2. [Dependency injection and faking](#)
 3. [Testing tools](#)

4. [Testing conventions](#)
5. [Running the unit and integration tests](#)
6. [Configuring Karma and Jasmine](#)
7. [Test suites with Jasmine](#)
 1. [Creating a test suite](#)
 2. [Specifications](#)
 3. [Structure of a test](#)
 4. [Expectations](#)
 5. [Efficient test suites](#)
8. [Faking dependencies](#)
 1. [Equivalence of fake and original](#)
 2. [Effective faking](#)
 3. [Faking functions with Jasmine spies](#)
 4. [Spying on existing methods](#)
9. [Debugging tests](#)
 1. [Test focus](#)
 2. [Developer tools](#)
 3. [Debug output and the JavaScript debugger](#)
 4. [Inspect the DOM](#)
 5. [Jasmine debug runner](#)
10. [Testing Components](#)
 1. [Unit test for the counter Component](#)
 2. [TestBed](#)

3. [Configuring the testing Module](#)
 4. [Rendering the Component](#)
 5. [TestBed and Jasmine](#)
 6. [ComponentFixture and DebugElement](#)
 7. [Writing the first Component spec](#)
 8. [Querying the DOM with test ids](#)
 9. [Triggering event handlers](#)
 10. [Expecting text output](#)
 11. [Testing helpers](#)
 12. [Filling out forms](#)
 13. [Testing Inputs](#)
 14. [Testing Outputs](#)
 15. [Repetitive Component specs](#)
 16. [Black vs. white box Component testing](#)
-
11. [Testing Components with children](#)
 1. [Shallow vs. deep rendering](#)
 2. [Unit test](#)
 3. [Faking a child Component](#)
 4. [Faking a child Component with ng-mocks](#)
 12. [Testing Components depending on Services](#)
 1. [Service dependency integration test](#)
 2. [Faking Service dependencies](#)
 3. [Fake Service with minimal logic](#)
 4. [Faking Services: Summary.](#)

13. [Testing complex forms](#)

1. [Sign-up form Component](#)
2. [Form validation and errors](#)
3. [Test plan](#)
4. [Test setup](#)
5. [Successful form submission](#)
6. [Invalid form](#)
7. [Form submission failure](#)
8. [Required fields](#)
9. [Asynchronous validators](#)
10. [Dynamic field relations](#)
11. [Password type toggle](#)
12. [Testing form accessibility](#)
 1. [pa11y](#)
 2. [pa11y-ci](#)
 3. [Start server and run pa11y-ci](#)

13. [Form accessibility: Summary](#)

14. [Testing Components with Spectator](#)

1. [Component with an Input](#)
2. [Component with children and Service dependency](#)
3. [Event handling with Spectator](#)
4. [Spectator: Summary](#)

15. [Testing Services](#)

1. [Testing a Service with internal state](#)
 2. [Testing a Service that sends HTTP requests](#)
 1. [Call the method under test](#)
 2. [Find pending requests](#)
 3. [Respond with fake data](#)
 4. [Check the result of the method call](#)
 5. [Verify that all requests have been answered](#)
 6. [Testing the error case](#)
 7. [Alternatives for finding pending requests](#)
 3. [Testing Services: Summary](#)
-
16. [Testing Pipes](#)
 1. [GreetPipe](#)
 2. [GreetPipe test](#)
 3. [Testing Pipes with dependencies](#)
 1. [TranslateService](#)
 2. [TranslatePipe](#)
 3. [TranslatePipe test](#)
-
17. [Testing Directives](#)
 1. [Testing Attribute Directives](#)
 1. [ThresholdWarningDirective](#)
 2. [ThresholdWarningDirective test](#)
 2. [Testing Structural Directives](#)
 1. [PaginateDirective](#)

2. [PaginateDirective test](#)

18. [Testing Modules](#)

19. [Measuring code coverage](#)

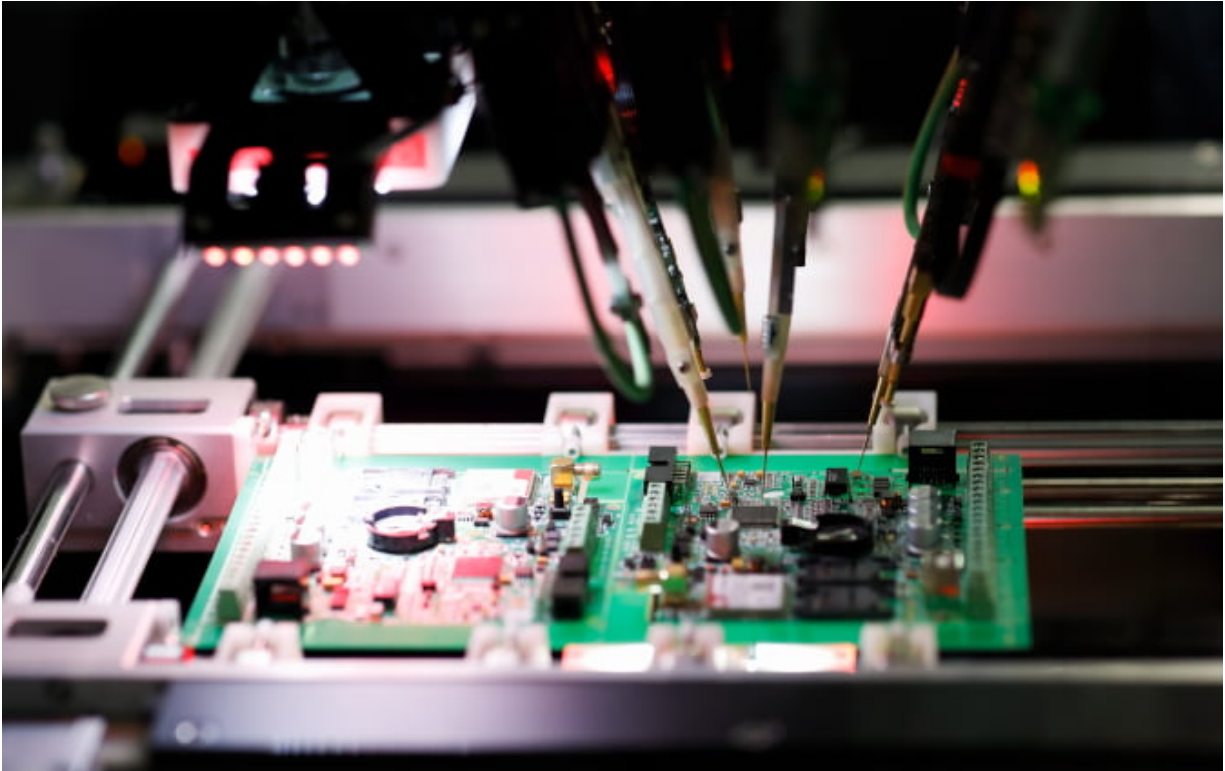
1. [Coverage report](#)
2. [How to use the coverage report](#)

20. [End-to-end testing](#)

1. [Strengths of end-to-end tests](#)
2. [Deployment for end-to-end tests](#)
3. [How end-to-end tests work](#)
4. [End-to-end testing frameworks](#)
5. [Introducing Cypress](#)
6. [Installing Cypress](#)
7. [Writing an end-to-end test with Cypress](#)
8. [Testing the counter Component](#)
9. [Running the Cypress tests](#)
10. [Asynchronous tests](#)
11. [Automatic retries and waiting](#)
12. [Testing the counter increment](#)
13. [Finding elements](#)
14. [Interacting with elements](#)
15. [Custom Cypress commands](#)
16. [Testing the Flickr search](#)
 1. [Testing the search form](#)
 2. [Testing the full photo](#)

- 17. [Page objects](#)
- 18. [Faking the Flickr API](#)
- 19. [End-to-end testing: Summary](#)

- 21. [Summary](#)
- 22. [Index of example applications](#)
- 23. [References](#)
- 24. [Acknowledgements](#)
- 25. [About](#)
- 26. [License](#)



Flying probes testing a printed circuit board. Photo by genkur from iStock.

Testing Angular

A Guide to Robust Angular Applications

THE ANGULAR FRAMEWORK is a mature and comprehensive solution for enterprise-ready applications based on web technologies. At Angular's core lies the ability to test all application parts in an automated way. How do we take advantage of Angular's testability?

This guide explains the principles of automated testing as well as the practice of testing Angular web applications. It empowers you and your team to write effective tests on a daily basis.

In this book, you learn how to set up your own testing conventions, write your own testing helpers and apply proven testing libraries. The book covers different tests that complement each other: unit tests, integration tests and end-to-end tests.

With a strong focus on testing Angular Components, this guide teaches you to write realistic specs that describe and verify a Component's behavior. It demonstrates how to properly mock dependencies like Services to test a Component in isolation. The guide introduces the Spectator and ng-mocks libraries, two powerful testing libraries.

Apart from Components, the book illustrates how to test the other application parts: Services, Pipes, Directives as well as Modules.

Last but not least, it covers end-to-end tests with Cypress.

Introduction

LEARNING OBJECTIVES

- The benefits of automated software testing
- Why testing is difficult to learn
- The difference between implementation and testing
- Establishing a steady testing practice

Most web developers come across automated tests in their career. They fancy the idea of writing code to scrutinize a web application and put it to an acid test. As web developers, as business people, we want to know whether the site works for the user, our customers.

Does the site allow the user to complete their tasks? Is the site still functional after new features have been introduced or internals have been refactored? How does the site react to usage errors or system failure? Testing gives answers to these questions.

I believe the benefits of automated testing are easy to grasp. Developers want to sleep well and be confident that their application works correctly. Moreover, testing helps developers to write better software. Software that is more robust, better to understand and easier to maintain.

In stark contrast, I have met only few web developers with a steady testing practice. Only few find it *easy*, let alone *enjoy* writing tests. This task is seen as a chore or nuisance.

Often individual developers are blamed for the lack of tests. The claim that developers are just too ignorant or lazy to write tests is simplistic and downright toxic. If testing has an indisputable value, we need to examine why developers avoid it while being convinced of the benefits. Testing should be easy, straightforward and commonplace.

If you are struggling with writing tests, it is not your fault or deficit. We are all struggling because testing software is inherently complicated and difficult.

First, writing automated tests requires a different mindset than writing the implementation code. Implementing a feature means building a structure – testing means trying to knock it over.

You try to find weaknesses and loopholes in your own work. You think through all possible cases and pester your code with “What if?” questions. What seems frustrating at first sight is an invaluable strategy to improve your code.

Second, testing has a steep learning curve. If testing can be seen as a tool, it is not like a screwdriver or power drill. Rather, it compares to a tractor or excavator. It takes training to operate

these machines. And it takes experience to apply them accurately and safely.

This is meant to encourage you. Getting started with testing is hard, but it gets easier and easier with more practice. The goal of this guide is to empower you to write tests on a daily basis that cover the important features of your Angular application.

Target audience

LEARNING OBJECTIVES

- Angular topics you need to know when reading this book
- Where to learn about Angular core concepts
- Picking chapters you are interested in

The target audience of this guide are intermediate Angular developers. You should be familiar with Angular's core concepts.

TESTING, NOT IMPLEMENTATION

This guide teaches you how to test Angular application parts like Components and Services. It assumes you know how to implement them, but not how to test them properly. If you have questions regarding Angular's core concepts, please refer to the [official Angular documentation](#).

If you have not used individual concepts yet, like Directives, that is fine. You can simply skip the related chapters and pick chapters you are interested in.

JAVASCRIPT & TYPESCRIPT PROFICIENCY

Furthermore, this guide is not an introduction to JavaScript or TypeScript. It assumes you have enough JavaScript and TypeScript knowledge to write the implementation and test code you need.

Of course, this guide will explain special idioms commonly used for testing.

The official Angular documentation offers a comprehensive [guide on testing](#). It is a recommended reading, but this guide does not assume you have read it.

Terminology

LEARNING OBJECTIVES

- Terms used in this book
- Notation conventions

Before we dive in, a quick note regarding the technical terms.

Some words have a special meaning in the context of Angular. In the broader JavaScript context, they have plenty other meanings. This guide tries to distinguish these meanings by using a different letter case.

When referring to core Angular concepts, this guide uses **upper case**:

Module, Component, Service, Input, Output, Directive, Pipe, etc.

When using these terms in the general sense, this guide uses **lower case**:

module, component, service, input, output, etc.

Testing principles

LEARNING OBJECTIVES

- The goals of testing and how to achieve them
- Establishing regular testing practice in your team
- Classify tests by their proximity to the code
- Classify tests by their knowledge of code internals

There is a gap between practical introductions – how to test a feature – and fundamental discussions on the core concepts – what does testing achieve, which types of tests are beneficial, etc. Before we dive into the tutorial, we need to reflect on a few basics about testing.

What makes a good test

When writing tests, you need to keep the goals of testing in mind. You need to judge whether a test is valuable with regard to these goals.

Automated testing has several technical, economical and organizational benefits. Let us pick a few that are useful to judge a test:

1. **Testing saves time and money.** Testing tries to nip software problems in the bud. Tests prevent bugs before they cause

real damage, when they are still manageable and under control.

Of course, quality assurance takes time and costs money itself. But it takes less time and is cheaper than letting bugs slip through into the software release.

When a faulty application ships to the customer, when users run into a bug, when data is lost or corrupted, your whole business might be at stake. After an incident, it is expensive to analyze and fix the bug in order to regain the user's trust.

COST-EFFECTIVE

A valuable test is cost-effective. The test prevents bugs that could ultimately render the application unusable. The test is cheap to write compared to the potential damage it prevents.

- 2. Testing formalizes and documents the requirements.** A test suite is a formal, human- and machine-readable description of how the code should behave. It helps the original developers to understand the requirements they have to implement. It helps fellow developers to understand the challenges they had to deal with.

DESCRIPTIVE

A valuable test clearly describes how the implementation code should behave. The test uses a proper language to talk to developers and convey the requirements. The test lists known cases the implementation has to deal with.

3. **Testing ensures that the code implements the requirements and does not exhibit bugs.** Testing taps every part of the code in order to find flaws.

SUCCESS AND ERROR CASES

A valuable test covers the important scenarios – both correct and incorrect input, expected cases as well as exceptional cases.

4. **Testing makes change safe by preventing regressions.** Tests not only verify that the current implementation meets the requirements. They also verify that the code still works as expected after changes. With proper automated tests in place, accidentally breakage is less likely. Implementing new features and code refactoring is safer.

PREVENT BREAKAGE

A valuable test fails when essential code is changed or deleted. Design the test to fail if dependent behavior is changed. It should still pass if unrelated code is changed.

What testing can achieve

Automated testing is a tool with a specific purpose. A basic concept is that testing helps to build an application that functions according to its requirements. That is true, but there are certain subtleties.

The [International Software Testing Qualifications Board \(ISTQB\)](#) came up with **Seven Testing Principles** that shed light on what testing can achieve and what not. Without discussing every principle, let us consider the main ideas.

DISCOVER BUGS

The purpose of a test is to **discover bugs**. If the test fails, it proves the presence of a bug (or the test is set up incorrectly). If the test passes, it proves that *this particular test setup* did not trigger a bug. It does not prove that the code is correct and free of bugs.

TEST HIGH-RISK CASES


So should you write automated tests for all possible cases to ensure correctness? No, say the ISTQB principles: “**Exhaustive testing is impossible**”. It is neither technically feasible nor worthwhile to write tests for all possible inputs and conditions. Instead, you should *assess the risks* of a certain case and write tests for high-risk cases first.

Even if it was viable to cover all cases, it would give you a false sense of security. No software is without errors, and a fully tested software may still be a usability nightmare that does not satisfy its users.

ADAPT TESTING APPROACH

Another core idea is that **testing depends on its context** and that it needs to be adapted again and again to provide meaning. The specific context in this guide are single-page web applications written in JavaScript, made with Angular. Such applications need specific testing methods and tools we will get to know.

Once you have learned and applied these tools, you should not stop. A fixed tool chain will only discover certain types of bugs. You need to try different approaches to find new classes of bugs. Likewise, an existing test suite needs to be updated regularly so that it still finds regressions.

 [International Software Testing Qualifications Board: Certified Tester Foundation Level Syllabus, Version 2018 V3.1, Page 16: Seven Testing Principles](#)

Tailoring your testing approach

There is not one correct approach to testing. In fact there are several competing schools of thoughts and methodologies. Learn

from other's experience, but develop a testing approach that suits your application, your team, your project or business.

EXAMINE YOUR APPLICATION

Before you start setting up tests, you should examine the current situation of your application:

- What are the **critical features**? For example, logging in, searching for a record and editing a form.
- What are the frequently reported **technical problems and obstacles**? For example, your application may lack error handling or cross-browser compatibility.
- What are the **technical requirements**? For example, your application needs to consume structured data from a given back-end API. In turn, it needs to expose certain URL routes.

DEVELOPMENT PROCESS

This technical assessment is as important as an inquiry of your development team:

- What is the overall **attitude on testing**? For example, some developers value testing while others find it ineffective to avoid bugs.

- What is the current **testing practice**? For example, developers sometimes write tests, but not as a daily routine.
- What is the **experience on writing tests**? For example, some developers have written tests for several environments, while others understand the basic concepts but have not yet gotten into practice.
- What are the **obstacles** that impede a good testing routine? For example, developers have not been trained on the testing tools.
- Are tests **well-integrated** into your development workflow? For example, a continuous integration server automatically runs the test suite on every change set.

Once you have answered these questions, you should set up a testing goal and implement steps to achieve it.

RETURN ON INVESTMENT

A good start is to think economically. What is the return on investment of writing a test? Pick the low-hanging fruits. Find business-critical features and make sure they are covered by tests. Write tests that require little effort but cover large parts of the code.

NORMALIZE TESTING

Simultaneously, integrate testing into your team's workflow:

- Make sure everyone shares the same basic expertise.
- Offer formal training workshops and pair experienced programmers with team members less familiar with testing.
- Appoint maintainers and contact persons for test quality and testing infrastructure.
- Hire dedicated software testers, if applicable.

Writing automated tests should be **easy and fun** for your team members. Remove any obstacles that make testing difficult or inefficient.

The right amount of testing

A fierce debate revolves around the right amount of testing. Too little testing is a problem: Features are not properly specified, bugs go unnoticed, regressions happen. But too much testing consumes development time, yields no additional profit and slows down development in the long run.

So we need to reach a sweet spot. If your testing practice deteriorates from this spot, you run into problems. If you add

more tests, you observe little benefit.

MEANINGFUL TESTS

Tests differ in their value and quality. Some tests are more meaningful than others. If they fail, your application is actually unusable. This means **the quality of tests is more important than their quantity**.

A common metric of testing is **code coverage**. It counts the lines in your code that are called by your tests. It tells you which parts of your code are executed at all.

This metric on testing is **useful but also deeply flawed** because the value of a test cannot be quantified automatically. Code coverage tells you whether a piece of code was called, regardless of its importance.

FIND UNCOVERED CODE

The coverage report may point to important behavior that is not yet covered by tests, but should be. It does not tell whether the existing tests are meaningful and make the right expectations. You can merely infer that the code does not throw exceptions under test conditions.

It is controversial whether one should strive for 100% code coverage. While it is feasible to cover 100% of certain business-

critical code, it requires immense efforts to cover all parts of an application written in Angular and TypeScript.

COVER MAIN FEATURES

If you write tests for the main features of your app from a user's perspective, you can achieve a code coverage of 60-70%. Every extra percent gain takes more and more time and bears weird and twisted tests that do not reflect the actual usage of your application.

We are going to discuss the [practical use of code coverage tools](#) later.

 [Angular guide: Code Coverage](#)

Levels of testing

We can distinguish automated tests by their perspective and proximity to the code.

End-to-end tests

SIMULATE REAL USAGE

Some tests have a *high-level, bird's-eye view* on the application. They simulate a user interacting with the application: Navigating to an address, reading text, clicking on a link or button, filling out a form, moving the mouse or typing on the keyboard. These tests make expectations about what the user sees and reads in the browser.

From the user's perspective, it does not matter that your application is implemented in Angular. Technical details like the inner structure of your code are not relevant. There is no distinction between front-end and back-end, between parts of your code. The full experience is tested.

END-TO-END TESTS

These tests are called **end-to-end (E2E) tests** since they integrate all parts of the application from one end (the user) to the other end (the darkest corners of the back-end). End-to-end tests also form the automated part of **acceptance tests** since they tell whether the application works for the user.

Unit tests

Other tests have a *low-level, worm's-eye view* on the application. They pick a small piece of code and put it through its paces. From this perspective, implementation details matter. The developer

needs to set up an appropriate testing environment to trigger all relevant cases.

ISOLATE ONE PIECE

The shortsighted worm only sees what is directly in front. This perspective tries to cut off the ties of the code under test with its dependencies. It tries to *isolate* the code in order to examine it.

UNIT TESTS

These tests are called **unit tests**. A unit is a small piece of code that is reasonable to test.

Integration tests

COHESIVE GROUPS

Between these two extreme perspectives, there are tests that operate on specific parts of the code, but test *cohesive groups*. They prescind from implementation details and try to take the user's perspective.

INTEGRATION TESTS

These tests are called **integration tests** since they test how well the parts *integrate* into the group. For example, all parts of one feature may be tested together. An integration test proves that the parts work together properly.

Distribution of testing efforts

All levels of testing are necessary and valuable. Different types of tests need to be combined to create a thorough test suite.

But how should we divide our attention? On which level should we spend most of the time? Should we focus on end-to-end tests since they mimic how the user interacts with the application? Again, this is a controversial issue among testing experts.

SPEED

What is indisputable is that high-level tests like end-to-end tests are expensive and slow, while lower-level tests like integration and unit tests are cheaper and faster.

RELIABILITY

Because of their inherent complexity, end-to-end tests tend to be unreliable. They often fail even though the software is without fault. Sometimes they fail for no apparent reason. When you run the same tests again, they suddenly pass. Even if the test correctly fails, it is hard to find the root cause of the problem. You need to wander through the full stack to locate the bug.

SETUP COSTS

End-to-end tests use a real browser and run against the full software stack. Therefore the testing setup is immense. You need

to deploy front-end, back-end, databases, caches, etc. to testing machines and then have machines to run the end-to-end tests.

In comparison, integration tests are simpler and unit tests even more so. Since they have less moving parts and fewer dependencies, they run faster and the results are reproducible. The setup is relatively simple. Integration and unit tests typically run on one machine against a build of the code under test.

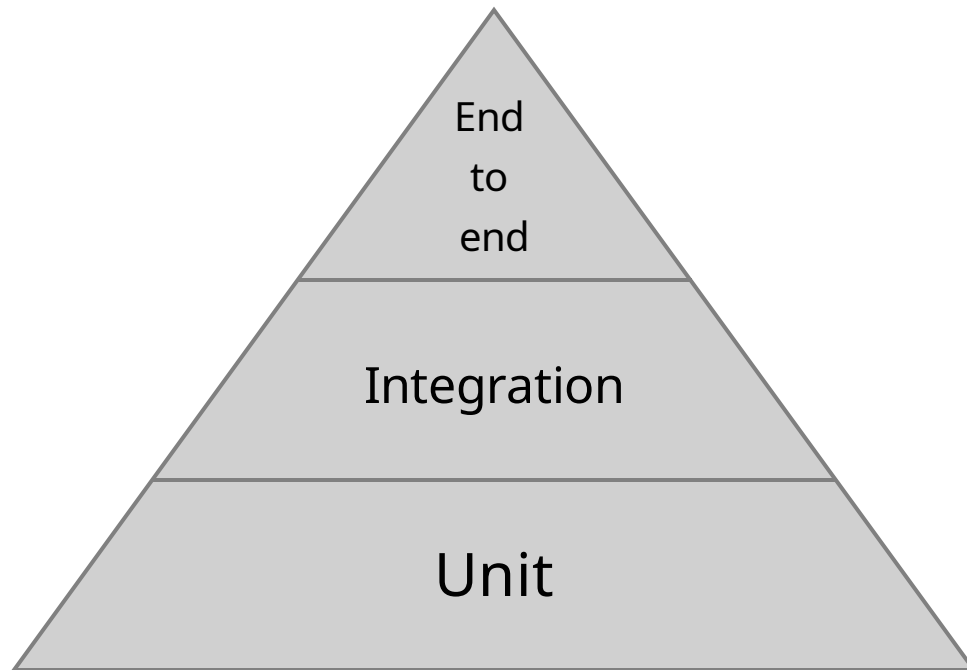
COST VS. BENEFIT

The crucial question for dividing your testing efforts is: Which tests yield the most return on investment? How much work is it to maintain a test in relation to its benefit?

In theory, the benefit of end-to-end tests is the highest, since they indicate whether the application works for the user. In practice, they are unreliable, imprecise and hard to debug. The business value of integration and unit tests is estimated higher.

DISTRIBUTION

For this reason, some experts argue you should write few end-to-end test, a fair amount of integration tests and many unit tests. If this distribution is visualized, it looks like a pyramid:



These proportions are known as the **Testing Pyramid**. They are widely recognized in software testing across domains, platforms and programming languages.

However, this common distribution also drew criticism. In particular, experts disagree on the value of unit tests.

DESIGN GUIDE

On the one hand, unit tests are precise and cheap. They are ideal to specify all tiny details of a shared module. They help developers to design small, composable modules that “do one thing and do it well”. This level of testing forces developers to reconsider how the module interacts with other modules.

CONFIDENCE

On the other hand, unit tests are too low-level to check whether a certain feature works for the user. They give you little confidence that your application works. In addition, unit tests might increase the cost of every code change.

Unit tests run the risk of mirroring or even duplicating implementation details. These details change frequently because of new requirements elsewhere or during internal refactoring. If you change a line of code somewhere, some distant unit test suddenly fails.

This makes sense if you have touched shared types or shared logic, but it may just be a false alarm. You have to fix this failing test for technical reasons, not because something broke.

MIDDLE GROUND

Integration tests provide a better trade-off. These mid-level tests prescind from implementation details, cover a group of code units and provide more confidence. They are less likely to fail if you refactor code inside of the group.

That is why some experts deem integration tests more valuable and recommend that you spend most of your testing efforts on this level.

In Angular, the difference between unit and integration tests is sometimes subtle. A unit test typically focusses on a single

Angular Component, Directive, Service, Pipe, etc. Dependencies are replaced with fakes. An integration test spans one Component together with its children and possibly connected Services as well. It is also possible to write a test that integrates all parts of an Angular Module.

Comparison of software testing levels

Level	End-to-End	Integration	Unit
Coverage	full	large	small
Performance	slow	fast	fastest
Reliability	least reliable	reliable	most reliable
Isolate Failures	hard	fair	easy
Simulate the Real User	yes	no	no

(Table adapted from a [Google Testing Blog article](#) by Mike Wacker.)

 [Martin Fowler: Test Pyramid](#)

 [Google Testing Blog: Just Say No to More End-to-End Tests](#)

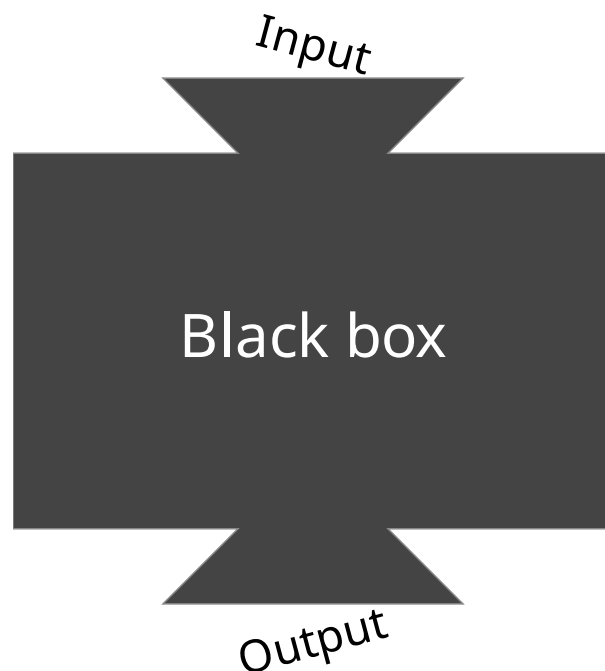
 [Kent C. Dodds: Write tests. Not too many. Mostly integration.](#)

Black box vs. white box testing

Once you have identified a piece of code you would like to test, you have to decide how to test it properly. One important distinction is whether a test treats the implementation as a closed, unlit box – a **black box** – or an open, lit box – a **white box**. In this metaphor, the code under test is a machine in a box with holes for inputs and outputs.

OUTSIDE

Black box testing does not assume anything about the internal structure. It puts certain values into the box and expects certain output values. The test talks to the publicly exposed, documented API. The inner state and workings are not examined.



INSIDE

White box testing opens the box, sheds light on the internals and takes measurements by reaching into the box. For example, a white box test may call methods that are not part of the public API, but still technically tangible. Then it checks the internal state and expects that it has changed accordingly.

IRRELEVANT INTERNALS

While both approaches have their value, this guide recommends to **write black box tests whenever possible**. You should check what the code does for the user and for other parts of the code. For this purpose, it is not relevant how the code looks internally. Tests that make assumptions about internals are likely to break in the future when the implementation slightly changes.

RELEVANT BEHAVIOR

More importantly, white box tests run the risk of forgetting to check the real output. They reach into the box, spin some wheel, flip some switch and check a particular state. They just assume the output without actually checking it. So they fail to cover important code behavior.

PUBLIC API

For an Angular Component, Directive, Service, Pipe, etc., a black box test passes a certain input and expects a proper output or measures side effects. The test only calls methods that are

marked with `public` in the TypeScript code. Internal methods should be marked with `private`.

Example applications

LEARNING OBJECTIVES

- Example Angular applications used in the book
- Features demonstrated by the examples

In this guide, we will explore the different aspects of testing Angular applications by looking at two examples.

The counter Component

 [Counter Component: Source code](#)

 [Counter Component: Run the app](#)

The counter is a reusable Component that increments, decrements and resets a number using buttons and input fields.

CHALLENGING TO TEST

For intermediate Angular developers, this might look trivial. That is intentional. This guide assumes that you know Angular basics and that you are able to build a counter Component, but struggle testing the ins and outs.

The goals of this example are:

- **Simplicity:** Quickly grasp what the Component is supposed to do.
- **Cover core Angular features:** Reusable Components with state, Inputs, Outputs, templates, event handling.
- **Scalability:** Starting point for more complex application architectures.

STATE MANAGEMENT

The counter comes in three flavors with different state management solutions:

1. An independent, self-sufficient counter Component that holds its own state.
2. A counter that is connected to a Service using dependency injection. It shares its state with other counters and changes it by calling Service methods.
3. A counter that is connected to a central NgRx Store. (NgRx is a popular state management library.) The counter changes the state indirectly by dispatching NgRx Actions.

While the counter seems easy to implement, it already offers valuable challenges from a testing perspective.

The Flickr photo search

 [Flickr photo search: Source code](#)

 [Flickr photo search: Run the app](#)

This application allows you to search for photos on Flickr, the popular photo hosting site.

TYPICAL APPLICATION FLOW

First, you enter a search term and start the search. The Flickr search API is queried. Second, the search results with thumbnails are rendered. Third, you can select a search result to see the photo details.

This application is straight-forward and relatively simple to implement. Still it raises important questions:

- **App structure:** How to split responsibilities into Components and how to model dependencies.
- **API communication:** How to fetch data by making HTTP requests and update the user interface.
- **State management:** Where to hold the state, how to pass it down in the Component tree, how to alter it.

The Flickr search comes in two flavors using different state management solutions:

STATE MANAGEMENT

1. The state is managed in the top-level Component, passed down in the Component tree and changed using Outputs.
2. The state is managed by an NgRx Store. Components are connected to the store to pull state and dispatch Actions. The state is changed in a Reducer. The side effects of an Action are handled by NgRx Effects.

Once you are able to write automatic tests for this example application, you will be able to test most features of a typical Angular application.

Angular testing principles

LEARNING OBJECTIVES

- Angular's architectural principles that facilitate testing
- Standard and alternative testing tools
- Running and configuring unit and integration tests with Karma and Jasmine

Testability

In contrast to other popular front-end JavaScript libraries, Angular is an opinionated, comprehensive framework that covers all important aspects of developing a JavaScript web application. Angular provides high-level structure, low-level building blocks and means to bundle everything together into a usable application.

TESTABLE ARCHITECTURE

The complexity of Angular cannot be understood without considering automated testing. Why is an Angular application structured into Components, Services, Modules, etc.? Why are the parts intertwined the way they are? Why do all parts of an Angular application apply the same patterns?

An important reason is **testability**. Angular's architecture guarantees that all application parts can be tested easily in a

similar way.

WELL-STRUCTURED CODE

We know from experience that code that is easy to test is also simpler, better structured, easier to read and easier to understand. The main technique of writing testable code is to break code into smaller chunks that “do one thing and do it well”. Then couple the chunks loosely.

Dependency injection and faking

A major design pattern for loose coupling is **dependency injection** and the underlying **inversion of control**. Instead of creating a dependency itself, an application part merely declares the dependency. The tedious task of creating and providing the dependency is delegated to an *injector* that sits on top.

This division of work decouples an application part from its dependencies: One part does not need to know how to set up a dependency, let alone the dependency’s dependencies and so forth.

LOOSE COUPLING

Dependency injection turns tight coupling into loose coupling. A certain application part no longer depends on a specific class,

function, object or other value. It rather depends on an abstract **token** that can be traded in for a concrete implementation. The injector takes the token and exchanges it for a real value.

ORIGINAL OR FAKE

This is of immense importance for automated testing. In our test, we can decide how to deal with a dependency:

- We can either provide an **original**, fully-functional implementation. In this case, we are writing an [integration test](#) that includes direct and indirect dependencies.
- Or we provide a **fake** implementation that does not have side effects. In this case, we are writing a [unit test](#) that tries to test the application part in *isolation*.

A large portion of the time spent while writing tests is spent on decoupling an application part from its dependencies. This guide will teach you how to set up the test environment, isolate an application part and reconnect it with equivalent fake objects.

 [Angular guide: Dependency injection](#)

Testing tools

Angular provides solid testing tools out of the box. When you create an Angular project using the command line interface, it comes with a fully-working testing setup for unit, integration and end-to-end tests.

BALANCED DEFAULTS

The Angular team already made decisions for you: [Jasmine](#) as testing framework and [Karma](#) as test runner. Implementation and test code is bundled with [Webpack](#). Application parts are typically tested inside Angular's [TestBed](#).

This setup is a trade-off with strengths and weaknesses. Since it is just one possible way to test Angular applications, you can compile your own testing tool chain.

ALTERNATIVES

For example, some Angular developers use [Jest](#) instead of Jasmine and Karma. Some use [Spectator](#) or the [Angular Testing Library](#) instead of using [TestBed](#) directly.

These alternatives are not better or worse, they simply make different trade-offs. This guide uses Jasmine and Karma for unit and integration tests. Later, you will learn about Spectator.

Once you have reached the limits of a particular setup, you should investigate whether alternatives make testing your application easier, faster and more reliable.

Testing conventions

Angular offers some tools and conventions on testing. By design, they are flexible enough to support different ways of testing. So you need to decide how to apply them.

MAKING CHOICES

This freedom of choice benefits experts, but confuses beginners. In your project, there should be one preferable way how to test a specific application part. You should make choices and set up project-wide conventions and patterns.

CAST CONVENTIONS INTO CODE

The testing tools that ship with Angular are low-level. They merely provide the basic operations. If you use these tools directly, your tests become messy, repetitive and hard to maintain.

Therefore, you should create **high-level testing tools** that cast your conventions into code in order to write short, readable and understandable tests.

This guide values strong conventions and introduces helper functions that codify these conventions. Again, your mileage may vary. You are free to adapt these tools to your needs or build other testing helpers.

Running the unit and integration tests

The Angular command line interface (CLI) allows you to run the unit, integration and end-to-end tests. If you have not installed the CLI yet or need to update to the latest version, run this command on your shell:

```
npm install -g @angular/cli
```

This installs Angular CLI globally so the `ng` command can be used everywhere. `ng` itself does nothing but exposing a couple of Angular-specific commands.

For example, `ng new` creates a new Angular project directory with a ready-to-use application scaffold. `ng serve` starts a development server, and `ng build` makes a build.

The command for starting the unit and integration tests is:

```
ng test
```

First, this command finds all files in the directory tree that match the pattern `.spec.ts`. Using Webpack, it compiles them into a JavaScript bundle, together with its dependencies. The bundle code also initializes the Angular testing environment – the `TestBed`.

Typically, an Angular application loads and starts an `AppModule`. This startup is called bootstrapping. The `AppModule` then imports other Modules, Components, Services, etc. This way, the bundler finds all parts of the application.

The test bundle works differently. It does not start with one Module in order to walk through its dependencies. It merely imports all files whose name ends with `.spec.ts`.

`.SPEC.TS`

Each `.spec.ts` file represents a test. Typically, one `.spec.ts` file contains at least one Jasmine test suite (more on that in the next chapter). The `.spec.ts` files are located in the same directory as the implementation code.

In our example application, the `CounterComponent` is located in [src/app/components/counter/counter.component.ts](#). The corresponding test file sits in [src/app/components/counter/counter.component.spec.ts](#). This is an Angular convention, not a technical necessity, and we are going to stick to it.

KARMA

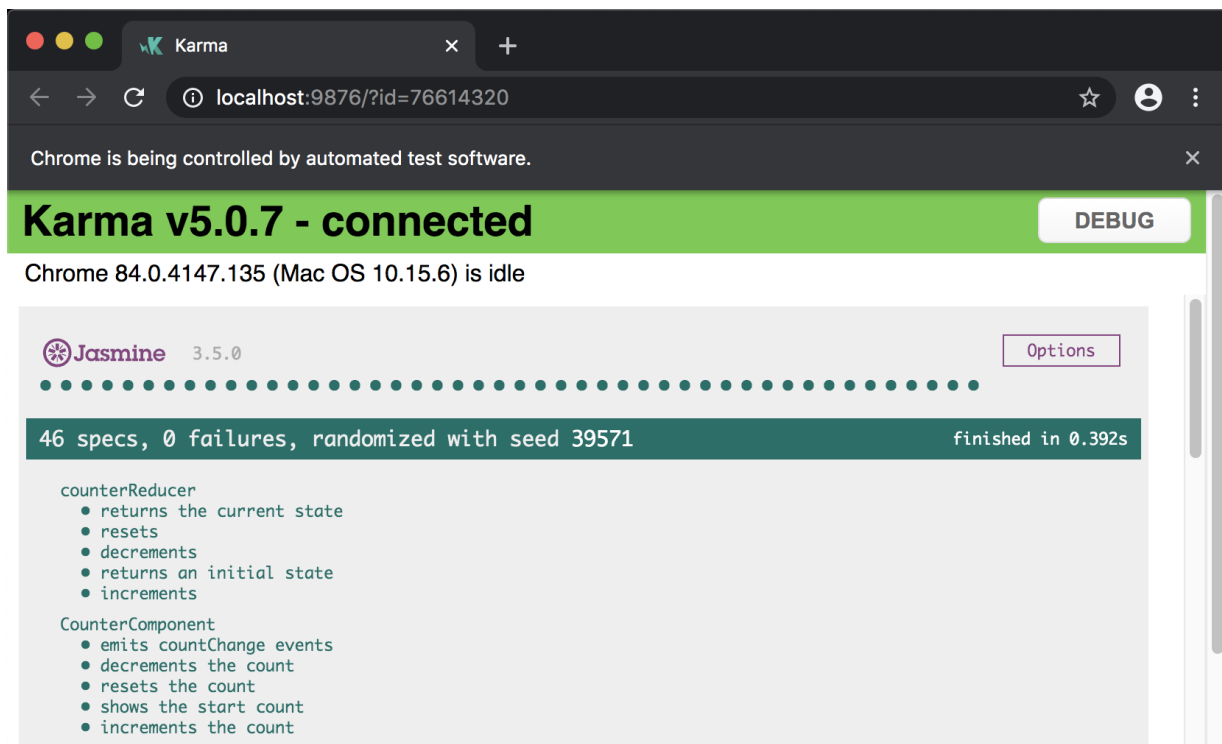
Second, `ng test` launches Karma, the test runner. Karma starts a development server at <http://localhost:9876/> that serves the JavaScript bundles compiled by Webpack.

Karma then launches one or more browsers. The idea of Karma is to run the same tests in different browsers to ensure cross-browser interoperability. All widely used browsers are supported: Chrome, Internet Explorer, Edge, Firefox and Safari. Per default, Karma starts Chrome.

TEST RUNNER

The launched browser navigates to <http://localhost:9876/>. As mentioned, this site serves the test runner and the test bundle. The tests start immediately. You can track the progress and read the results in the browser and on the shell.

When running the tests in the [counter project](#), the browser output looks like this:



This is the shell output:

```
INFO [karma-server]: Karma v5.0.7 server started at http://0.0.0.0:9876/  
INFO [launcher]: Launching browsers Chrome with concurrency unlimited  
INFO [launcher]: Starting browser Chrome  
WARN [karma]: No captured browser, open http://localhost:9876/  
INFO [Chrome 84.0.4147.135 (Mac OS 10.15.6)]: Connected on socket yH0-  
wtoVtfIRWMoWAAAA with id 76614320  
Chrome 84.0.4147.135 (Mac OS 10.15.6): Executed 46 of 46 SUCCESS (0.394 secs  
/ 0.329 secs)  
TOTAL: 46 SUCCESS
```

Webpack watches changes on the `.spec.ts` files and files imported by them. When you change the implementation code, `counter.component.ts` for example, or the test code, `counter.component.spec.ts` for example, Webpack automatically re-compiles the bundle and pushes it to the open browsers. All tests will be restarted.

RED-GREEN CYCLE

This feedback cycle allows you to work on the implementation and test code side-by-side. This is important for test-driven development. You change the implementation and expect the test to fail – the test is “red”. You adapt the test so it passes again – the test is “green”. Or you write a failing test first, then adapt the implementation until the test passes.

Test-driven development means letting the red-green cycle guide your development.

 [Angular CLI reference: ng test](#)

Configuring Karma and Jasmine

Karma and Jasmine are configured in the file `karma.conf.js` in the project's root directory. Since Angular 15, the Angular CLI does not create this file per default. If it does not exist, you can create it using this shell command:

```
ng generate config karma
```

There are many configuration options and plenty of plugins, so we will only look at a few.

LAUNCHERS

As mentioned, the standard configuration runs the tests in the Chrome browser. To run the tests in other browsers, we need to install different **launchers**.

Each launcher needs to be loaded in the `plugins` array:

```
plugins: [  
  require('karma-jasmine'),  
  require('karma-chrome-launcher'),  
  require('karma-jasmine-html-reporter'),  
  require('karma-coverage'),  
  require('@angular-devkit/build-angular/plugins/karma')  
],
```

There is already one launcher, `karma-chrome-launcher`. This is an npm package.

To install other launchers, we first need to install the respective npm package. Let us install the Firefox launcher. Run this shell command:

```
npm install --save-dev karma-firefox-launcher
```

Then we require the package in `karma.conf.js`:

```
plugins: [  
  require('karma-jasmine'),  
  require('karma-chrome-launcher'),  
  require('karma-firefox-launcher'),  
  require('karma-jasmine-html-reporter'),  
  require('karma-coverage'),  
  require('@angular-devkit/build-angular/plugins/karma'),  
],
```

To run the tests in Firefox as well, we need to add the Firefox to the browsers list: `browsers: ['Chrome']` becomes `browsers: ['Chrome', 'Firefox']`.

Karma will now start two browsers to run the tests in parallel.

REPORTERS

Another important concept of Karma are **reporters**. They format and output the test results. In the default configuration, three reporters are active:

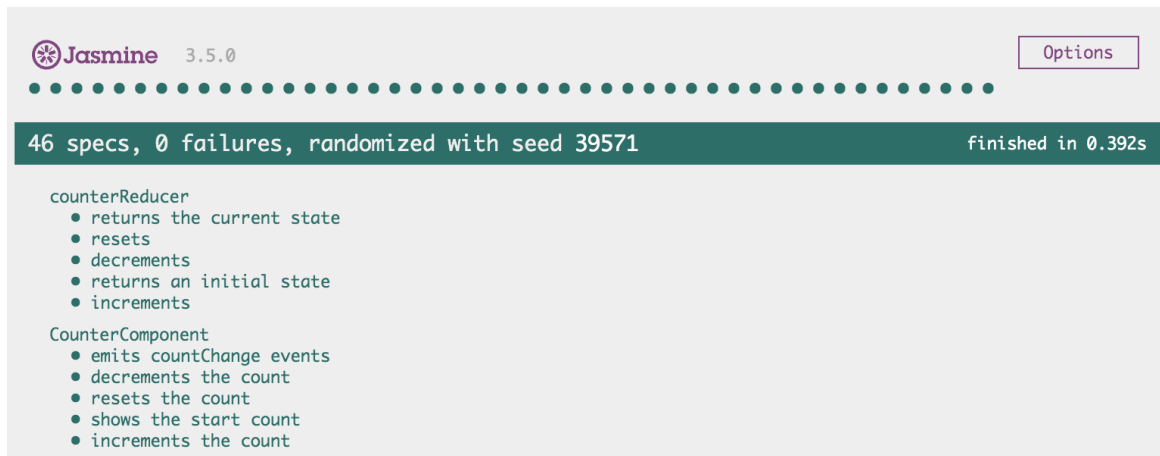
1. The built-in [progress](#) reporter outputs text on the shell. While the tests are running, it outputs the progress:

```
Chrome 84.0.4147.135 (Mac OS 10.15.6): Executed 9 of 46 SUCCESS (0.278 secs / 0.219 secs)
```

And finally:

```
Chrome 84.0.4147.135 (Mac OS 10.15.6): Executed 46 of 46 SUCCESS (0.394 secs / 0.329 secs)
TOTAL: 46 SUCCESS
```

2. The standard HTML reporter [kjhtml](#) (npm package: [karma-jasmine-html-reporter](#)) renders the results in the browser.



3. The coverage reporter (npm package: [karma-coverage](#)) creates the test coverage report. See [measuring code coverage](#).

By editing the `reporters` array, you can add reporters or replace the existing ones:

```
reporters: ['progress', 'kjhtml'],
```

For example, to add a reporter that creates JUnit XML reports, first install the npm package:

```
npm install --save-dev karma-junit-reporter
```

Next, require it as a plugin:

```
plugins: [  
  require('karma-jasmine'),  
  require('karma-chrome-launcher'),  
  require('karma-jasmine-html-reporter'),  
  require('karma-coverage'),  
  require('karma-junit-reporter'),  
  require('@angular-devkit/build-angular/plugins/karma'),  
],
```

Finally, add the reporter:

```
reporters: ['progress', 'kjhtml', 'junit'],
```

After running the tests with `ng test`, you will find an XML report file in the project directory.

JASMINE CONFIGURATION

The configuration for the Jasmine adapter is located in `jasmine` object inside the `client` object:

```
client: {
  jasmine: {
    // you can add configuration options for Jasmine here
    // the possible options are listed at
    https://jasmine.github.io/api/edge/Configuration.html
    // for example, you can disable the random execution with `random:
false`
    // or set a specific seed with `seed: 4321`
  },
  clearContext: false // leave Jasmine Spec Runner output visible in browser
},
```

This guide recommends to activate one useful Jasmine configuration option: `failSpecWithNoExpectations` lets the test fail if it does not contain at least one expectation. (More on [expectations](#) later.) In almost all cases, specs without expectations stem from an error in the test code.

```
client: {
  jasmine: {
    failSpecWithNoExpectations: true,
  },
  clearContext: false // leave Jasmine Spec Runner output visible in browser
},
```

 [Karma documentation: Configuration File](#)

 [Karma documentation: Plugins](#)

 [npm: List of Karma plugins](#)

 [Jasmine reference: Configuration options](#)

Test suites with Jasmine

LEARNING OBJECTIVES

- Introducing the Jasmine testing framework
- Writing test suites, specs and assertions
- Structuring a spec with Arrange, Act, Assert
- Efficient test suites with setup and teardown logic

Angular ships with Jasmine, a JavaScript framework that enables you to write and execute unit and integration tests. Jasmine consists of three important parts:

1. A library with classes and functions for constructing tests.
2. A test execution engine.
3. A reporting engine that outputs test results in different formats.

If you are new to Jasmine, it is recommended to read the [official Jasmine tutorial](#). This guide provides a short introduction to Jasmine, exploring the basic structure and terminology that will be used throughout this guide.

Creating a test suite

In terms of Jasmine, a test consists of one or more **suites**. A suite is declared with a `describe` block:

```
describe('Suite description', () => {  
  /* ... */  
});
```

Each suite *describes* a piece of code, the *code under test*.

DESCRIBE: SUITE

`describe` is a function that takes two parameters.

1. A string with a human-readable name. Typically the name of the function or class under test. For example, `describe('CounterComponent', /* ... */)` is the suite that tests the `CounterComponent` class.
2. A function containing the suite definition.

A `describe` block groups related specs that we will learn about in the next chapter.

NESTING DESCRIBE

`describe` blocks can be nested to structure big suites and divide them into logical sections:

```
describe('Suite description', () => {  
  describe('One aspect', () => {  
    /* ... */  
  })  
});
```

```
});  
describe('Another aspect', () => {  
  /* ... */  
});  
});
```

Nested `describe` blocks add a human-readable description to a group of specs. They can also host their own setup and teardown logic.

Specifications

IT: SPEC

Each suit consists of one or more *specifications*, or short, **specs**. A spec is declared with an `it` block:

```
describe('Suite description', () => {  
  it('Spec description', () => {  
    /* ... */  
  });  
  /* ... more specs ... */  
});
```

Again, `it` is a function that takes two parameters. The first parameter is a string with a human-readable description. The second parameter is a function containing the spec code.

READABLE SENTENCE

The pronoun `it` refers to the code under test. `it` should be the subject of a human-readable sentence that asserts the behavior of the code under test. The spec code then proves this assertion. This style of writing specs originates from the concept of Behavior-Driven Development (BDD).

One goal of BDD is to describe software behavior in a natural language – in this case, English. Every stakeholder should be able to read the `it` sentences and understand how the code is supposed to behave. Team members without JavaScript knowledge should be able to add more requirements by forming `it does something` sentences.

Ask yourself, what does the code under test do? For example, in case of a `CounterComponent`, *it* increments the counter value. And *it* resets the counter to a specific value. So you could write:

```
it('increments the count', () => {  
  /* ... */  
});  
it('resets the count', () => {  
  /* ... */  
});
```

After `it`, typically a verb follows, like `increments` and `resets` in the example.

NO “SHOULD”

Some people prefer to write `it('should increment the count', /* ... */)`, but `should` bears no additional meaning. The nature of a spec is to state what the code under test *should* do. The word “should” is redundant and just makes the sentence longer. This guide recommends to simply state what the code does.

 [Jasmine tutorial: Your first suite](#)

Structure of a test

Inside the `it` block lies the actual testing code. Irrespective of the testing framework, the testing code typically consists of three phases: **Arrange, Act and Assert**.

ARRANGE, ACT, ASSERT

1. **Arrange** is the preparation and setup phase. For example, the class under test is instantiated. Dependencies are set up. Spies and fakes are created.
2. **Act** is the phase where interaction with the code under test happens. For example, a method is called or an HTML element in the DOM is clicked.

3. **Assert** is the phase where the code behavior is checked and verified. For example, the actual output is compared to the expected output.

How could the structure of the spec `it('resets the count', /* ... */) for the CounterComponent look like?`

1. **Arrange:**

- Create an instance of `CounterComponent`.
- Render the Component into the document.

2. **Act:**

- Find and focus the reset input field.
- Enter the text "5".
- Find and click the "Reset" button.

3. **Assert:**

- Expect that the displayed count now reads "5".

STRUCTURE A TEST

This structure makes it easier to come up with a test and also to implement it. Ask yourself:

- What is the necessary setup? Which dependencies do I need to provide? How do they behave? (*Arrange*)
- What is the user input or API call that triggers the behavior I would like to test? (*Act*)
- What is the expected behavior? How do I prove that the behavior is correct? (*Assert*)

GIVEN, WHEN, THEN

In Behavior-Driven Development (BDD), the three phases of a test are fundamentally the same. But they are called **Given, When and Then**. These plain English words try to avoid technical jargon and pose a natural way to think of a test's structure: "*Given* these conditions, *when* the user interacts with the application, *then* it behaves in a certain way."

Expectations

In the *Assert* phase, the test compares the actual output or return value to the expected output or return value. If they are the same, the test passes. If they differ, the test fails.

Let us examine a simple contrived example, an `add` function:

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

A primitive test without any testing tools could look like this:

```
const expectedValue = 5;
const actualValue = add(2, 3);
if (expectedValue !== actualValue) {
  throw new Error(
    `Wrong return value: ${actualValue}. Expected: ${expectedValue}`
  );
}
```

EXPECT

We could write that code in a Jasmine spec, but Jasmine allows us to create expectations in an easier and more concise manner: The `expect` function together with a **matcher**.

```
const expectedValue = 5;
const actualValue = add(2, 3);
expect(actualValue).toBe(expectedValue);
```

First, we pass the actual value to the `expect` function. It returns an expectation object with methods for checking the actual value. We would like to compare the actual value to the expected value, so we use the `toBe` matcher.

MATCHERS

`toBe` is the simplest matcher that applies to all possible JavaScript values. Internally, it uses JavaScript's strict equality operator `===`. `expect(actualValue).toBe(expectedValue)` essentially runs `actualValue === expectedValue`.

`toBe` is useful to compare primitive values like strings, numbers and booleans. For objects, `toBe` matches only if the actual and the expected value are the very same object. `toBe` fails if two objects are not identical, even if they happen to have the same properties and values.

For checking the deep equality of two objects, Jasmine offers the `toEqual` matcher. This example illustrates the difference:

```
// Fails, the two objects are not identical
expect({ name: 'Linda' }).toBe({ name: 'Linda' });

// Passes, the two objects are not identical but deeply equal
expect({ name: 'Linda' }).toEqual({ name: 'Linda' });
```

Jasmine has numerous useful matchers built-in, `toBe` and `toEqual` being the most common. You can add custom matchers to hide a complex check behind a short name.

READABLE SENTENCE

The pattern `expect(actualValue).toEqual(expectedValue)` originates from Behavior-Driven Development (BDD) again. The `expect` function call and the matcher methods form a human-readable sentence: “Expect the actual value to equal the expected value.” The goal is to write a specification that is as readable as a plain text but can be verified automatically.

 [Jasmine documentation: Built-in matchers](#)

Efficient test suites

When writing multiple specs in one suite, you quickly realize that the *Arrange* phase is similar or even identical across these specs. For example, when testing the [CounterComponent](#), the *Arrange* phase always consists of creating a Component instance and rendering it into the document.

REPETITIVE SETUP

This setup is repeated over and over, so it should be defined once in a central place. You could write a [setup](#) function and call it at the beginning of each spec. But using Jasmine, you can declare code that is called before and after each spec, or before and after all specs.

For this purpose, Jasmine provides four functions: [beforeEach](#), [afterEach](#), [beforeAll](#) and [afterAll](#). They are called inside of a [describe](#) block, just like [it](#). They expect one parameter, a function that is called at the given stages.

```
describe('Suite description', () => {  
  beforeAll(() => {  
    console.log('Called before all specs are run');  
  });  
  afterAll(() => {
```

```
    console.log('Called after all specs are run');
  });

  beforeEach(() => {
    console.log('Called before each spec is run');
  });
  afterEach(() => {
    console.log('Called after each spec is run');
  });

  it('Spec 1', () => {
    console.log('Spec 1');
  });
  it('Spec 2', () => {
    console.log('Spec 2');
  });
});
```

This suite has two specs and defines shared setup and teardown code. The output is:

```
Called before all specs are run
Called before each spec is run
Spec 1
Called after each spec is run
Called before each spec is run
Spec 2
Called after each spec is run
Called after all specs are run
```

Most tests we are going to write will have a `beforeEach` block to host the *Arrange* code.

Faking dependencies

LEARNING OBJECTIVES

- Testing a code unit in isolation
- Replacing dependencies with fakes
- Rules for creating fakes to avoid pitfalls
- Using Jasmine spies to fake functions and methods

When testing a piece of code, you need to decide between an [integration test](#) and a [unit test](#). To recap, the integration test includes (“integrates”) the dependencies. In contrast, the unit test replaces the dependencies with fakes in order to isolate the code under test.

ALSO KNOWN AS MOCKING

These replacements are also called *test doubles*, *stubs* or *mocks*. Replacing a dependency is called *stubbing* or *mocking*.

Since these terms are used inconsistently and their difference is subtle, **this guide uses the term “fake” and “faking”** for any dependency substitution.

FAKING SAFELY

Creating and injecting fake dependencies is essential for unit tests. This technique is double-edged – powerful and dangerous at the same time. Since we will create many fakes throughout this

guide, we need to set up **rules for faking dependencies** to apply the technique safely.

Equivalence of fake and original

A fake implementation must have the same shape of the original. If the dependency is a function, the fake must have the same signature, meaning the same parameters and the same return value. If the dependency is an object, the fake must have the same public API, meaning the same public methods and properties.

REPLACEABILITY

The fake does not need to be complete, but sufficient enough to act as a replacement. The fake needs to be **equivalent to the original** as far as the code under test is concerned, not fully equal to the original.

Imagine a fake building on a movie set. The outer shape needs to be indistinguishable from an original building. But behind the authentic facade, there is only a wooden scaffold. The building is an empty shell.

The biggest danger of creating a fake is that it does not properly mimic the original. Even if the fake resembles the original at the

time of writing the code, it might easily get out of sync later when the original is changed.

When the original dependency changes its public API, dependent code needs to be adapted. Also, the fake needs to be aligned.

When the fake is outdated, the unit test becomes a fantasy world where everything magically works. The test passes but in fact the code under test is broken.

KEEP FAKE IN SYNC

How can we ensure that the fake is up-to-date with the original?

How can we ensure the equivalence of original and fake in the long run and prevent any possible divergence?

We can use TypeScript to **enforce that the fake has a matching type**. The fake needs to be strictly typed. The fake's type needs to be a subset of the original's type.

TYPE EQUIVALENCE

Then, TypeScript assures the equivalence. The compiler reminds us to update the implementation and the fake. The TypeScript code simply does not compile if we forget that. We will learn how to declare matching types in the upcoming examples.

Effective faking

The original dependency code has side effects that need to be suppressed during testing. The fake needs to *effectively* prevent the original code from being executed. Strange errors may happen if a mix of fake and original code is executed.

DO NOT MIX FAKE AND ORIGINAL

In some faking approaches, the fake inherits from the original. Only those properties and methods are overwritten that are currently used by the code under test.

This is dangerous since we may forget to overwrite methods. When the code under test changes, the test may accidentally call original methods of the dependency.

This guide will present thorough faking techniques that do not allow a slip. They imitate the original code while shielding the original from calls.

Faking functions with Jasmine spies

Jasmine provides simple yet powerful patterns to create fake implementations. The most basic pattern is the **Jasmine spy** for replacing a function dependency.

CALL RECORD

In its simplest form, a spy is a function that records its calls. For each call, it records the function parameters. Using this record, we later assert that the spy has been called with particular input values.

For example, we declare in a spec: “Expect that the spy has been called two times with the values `mickey` and `minnie`, respectively.”

Like every other function, a spy can have a meaningful return value. In the simple case, this is a fixed value. The spy will always return the same value, regardless of the input parameters. In a more complex case, the return value originates from an underlying fake function.

CREATESPY

A standalone spy is created by calling `jasmine.createSpy`:

```
const spy = jasmine.createSpy('name');
```

`createSpy` expects one parameter, an optional name. It is recommended to pass a name that describes the original. The name will be used in error messages when you make expectations against the spy.

Assume we have class `TodoService` responsible for fetching a to-do list from the server. The class uses the [Fetch API](#) to make an

HTTP request. (This is a plain TypeScript example. It is uncommon to use `fetch` directly in an Angular app.)

```
class TodoService {
  constructor(
    // Bind `fetch` to `window` to ensure that `window` is the `this`
    context
    private fetch = window.fetch.bind(window)
  ) {}

  public async getTodos(): Promise<string[]> {
    const response = await this.fetch('/todos');
    if (!response.ok) {
      throw new Error(
        `HTTP error: ${response.status} ${response.statusText}`
      );
    }
    return await response.json();
  }
}
```

INJECT FAKE

The `TodoService` uses the **constructor injection** pattern. The `fetch` dependency can be injected via an optional constructor parameter. In production code, this parameter is empty and defaults to the original `window.fetch`. In the test, a fake dependency is passed to the constructor.

The `fetch` parameter, whether original or fake, is saved as an instance property `this.fetch`. Eventually, the public method

`getTodos` uses it to make an HTTP request.

In our unit test, we do not want the Service to make any HTTP requests. We pass in a Jasmine spy as replacement for `window.fetch`.

```
// Fake todos and response object
const todos = [
  'shop groceries',
  'mow the lawn',
  'take the cat to the vet'
];
const okResponse = new Response(JSON.stringify(todos), {
  status: 200,
  statusText: 'OK',
});

describe('TodoService', () => {
  it('gets the to-dos', async () => {
    // Arrange
    const fetchSpy = jasmine.createSpy('fetch')
      .and.returnValue(okResponse);
    const todoService = new TodoService(fetchSpy);

    // Act
    const actualTodos = await todoService.getTodos();

    // Assert
    expect(actualTodos).toEqual(todos);
    expect(fetchSpy).toHaveBeenCalledWith('/todos');
  });
});
```

There is a lot to unpack in this example. Let us start with the fake data before the `describe` block:

```
const todos = [  
  'shop groceries',  
  'mow the lawn',  
  'take the cat to the vet'  
];  
const okResponse = new Response(JSON.stringify(todos), {  
  status: 200,  
  statusText: 'OK',  
});
```

First, we define the fake data we want the `fetch` spy to return. Essentially, this is an array of strings.

FAKE RESPONSE

The original `fetch` function returns a `Response` object. We create one using the built-in `Response` constructor. The original server response is a string before it is parsed as JSON. So we need to serialize the array into a string before passing it to the `Response` constructor. (These `fetch` details are not relevant to grasp the spy example.)

Then, we declare a test suite using `describe`:

```
describe('TodoService', () => {  
  /* ... */  
});
```

The suite contains one spec that tests the `getTodos` method:

```
it('gets the to-dos', async () => {  
  /* ... */  
});
```

The spec starts with *Arrange* code:

```
// Arrange  
const fetchSpy = jasmine.createSpy('fetch')  
  .and.returnValue(okResponse);  
const todoService = new TodoService(fetchSpy);
```

Here, we create a spy. With `.and.returnValue(...)`, we set a fixed return value: the successful response.

INJECT SPY

We also create an instance of `TodoService`, the class under test. We pass the spy into the constructor. This is a form of manual dependency injection.

In the *Act* phase, we call the method under test:

```
const actualTodos = await todoService.getTodos();
```

`getTodos` returns a Promise. We use an `async` function together with `await` to access the return value easily. Jasmine deals with async functions just fine and waits for them to complete.

In the *Assert* phase, we create two expectations:

```
expect(actualTodos).toEqual(todos);  
expect(fetchSpy).toHaveBeenCalledWith('/todos');
```

DATA PROCESSING

First, we verify the return value. We compare the actual data (`actualTodos`) with the fake data the spy returns (`todos`). If they are equal, we have proven that `getTodos` parsed the response as JSON and returned the result. (Since there is no other way `getTodos` could access the fake data, we can deduce that the spy has been called.)

VERIFY CALL RECORD

Second, we verify that the `fetch` spy has been called *with the correct parameter*, the API endpoint URL. Jasmine offers several matchers for making expectations on spies. The example uses `toHaveBeenCalledWith` to assert that the spy has been called with the parameter `'/todos'`.

Both expectations are necessary to guarantee that `getTodos` works correctly.

HAPPY AND UNHAPPY PATHS

After having written the first spec for `getTodos`, we need to ask ourselves: Does the test fully cover its behavior? We have tested the success case, also called *happy path*, but the error case, also

called *unhappy path*, is yet to be tested. In particular, this error handling code:

```
if (!response.ok) {  
  throw new Error(  
    `HTTP error: ${response.status} ${response.statusText}`  
  );  
}
```

When the server response is not “ok”, we throw an error. “Ok” means the HTTP response status code is 200-299. Examples of “not ok” are “403 Forbidden”, “404 Not Found” and “500 Internal Server Error”. Throwing an error rejects the Promise so the caller of `getTodos` knows that fetching the to-dos failed.

The fake `okResponse` mimics the success case. For the error case, we need to define another fake `Response`. Let us call it `errorResponse` with the notorious HTTP status 404 Not Found:

```
const errorResponse = new Response('Not Found', {  
  status: 404,  
  statusText: 'Not Found',  
});
```

Assuming the server does not return JSON in the error case, the response body is simply the string `'Not Found'`.

Now we add a second spec for the error case:

```
describe('TodoService', () => {  
  /* ... */  
})
```

```

it('handles an HTTP error when getting the to-dos', async () => {
  // Arrange
  const fetchSpy = jasmine.createSpy('fetch')
    .and.returnValue(errorResponse);
  const todoService = new TodoService(fetchSpy);

  // Act
  let error;
  try {
    await todoService.getTodos();
  } catch (e) {
    error = e;
  }

  // Assert
  expect(error).toEqual(new Error('HTTP error: 404 Not Found'));
  expect(fetchSpy).toHaveBeenCalledWith('/todos');
});
});

```

In the *Arrange* phase, we inject a spy that returns the error response.

CATCHING ERRORS

In the *Act* phase, we call the method under test but anticipate that it throws an error. In Jasmine, there are several ways to test whether a Promise has been rejected with an error. The example above wraps the `getTodos` call in a `try/catch` statement and saves the error. Most likely, this is how implementation code would handle the error.

In the *Assert* phase, we make two expectations again. Instead of verifying the return value, we make sure the caught error is an `Error` instance with a useful error message. Finally, we verify that the spy has been called with the right value, just like in the spec for the success case.

Again, this is a plain TypeScript example to illustrate the usage of spies. Usually, an Angular Service does not use `fetch` directly but uses `HttpClient` instead. We will get to know testing this later (see [Testing a Service that sends HTTP requests](#)).

 [TodoService: Implementation and test code](#)

 [Jasmine reference: Spies](#)

Spying on existing methods

We have used `jasmine.createSpy('name')` to create a standalone spy and have injected it into the constructor. Explicit constructor injection is straight-forward and used extensively in Angular code.

SPY ON OBJECT METHODS

Sometimes, there is already an object whose method we need to spy on. This is especially helpful if the code uses global methods

from the browser environment, like `window.fetch` in the example above.

For this purpose, we can use the `spyOn` method:

```
spyOn(window, 'fetch');
```

OVERWRITE AND RESTORE

This installs a spy on the global `fetch` method. Under the hood, Jasmine saves the original `window.fetch` function for later and overwrites `window.fetch` with a spy. Once the spec is completed, Jasmine automatically restores the original function.

`spyOn` returns the created spy, enabling us to set a return value, like we have learned above.

```
spyOn(window, 'fetch')  
  .and.returnValue(okResponse);
```

We can create a version of `TodoService` that does not rely on construction injection, but uses `fetch` directly:

```
class TodoService {  
  public async getTodos(): Promise<string[]> {  
    const response = await fetch('/todos');  
    if (!response.ok) {  
      throw new Error(  
        `HTTP error: ${response.status} ${response.statusText}`  
      );  
    }  
    return await response.json();  
  }  
}
```

```
    }  
  }  
}
```

The test suite then uses `spyOn` to catch all calls to `window.fetch`:

```
// Fake todos and response object  
const todos = [  
  'shop groceries',  
  'mow the lawn',  
  'take the cat to the vet'  
];  
const okResponse = new Response(JSON.stringify(todos), {  
  status: 200,  
  statusText: 'OK',  
});  
  
describe('TodoService', () => {  
  it('gets the to-dos', async () => {  
    // Arrange  
    spyOn(window, 'fetch')  
      .and.returnValue(okResponse);  
    const todoService = new TodoService();  
  
    // Act  
    const actualTodos = await todoService.getTodos();  
  
    // Assert  
    expect(actualTodos).toEqual(todos);  
    expect(window.fetch).toHaveBeenCalledWith('/todos');  
  });  
});
```

Not much has changed here. We spy on `fetch` and make it return `okResponse`. Since `window.fetch` is overwritten with a spy, we make the expectation against it to verify that it has been called.

Creating standalone spies and spying on existing methods are not mutually exclusive. Both will be used frequently when testing Angular applications, and both work well with dependencies injected into the constructor.

 [Jasmine reference: spyOn](#)

Debugging tests

LEARNING OBJECTIVES

- Fixing problems in your test
- Finding the cause of a failing test
- Applying familiar debugging techniques to tests
- Using Jasmine debugging features

Writing tests is as arduous as writing implementation code. You will be stuck quite often and ask yourself why the test fails – and sometimes why the test passes when it should rather fail.

The good news is that you can apply familiar debugging techniques to tests as well.

Test focus

Some tests require an extensive *Arrange* phase, the *Act* phase calls several methods or simulates complex user input. These tests are hard to debug.

ISOLATE THE PROBLEM

When locating an error, narrow down the scope gradually:
Execute only one test, one suite, one spec, one expectation.

Per default, Karma and Jasmine compile and run all specs again with every code change. This leads to a slow feedback cycle when you work on a particular spec. After a code change, it may take 10-20 seconds before you see the test result. Also one spec might interfere with another spec.

The easiest way to narrow down the scope is to set a **focus** on a suite or spec. Let us assume you have a test suite with two specs:

```
describe('Example spec', () => {  
  it('one spec', () => { /* ... */ });  
  it('another spec', () => { /* ... */ });  
});
```

FDESCRIBE

If you want Jasmine to run only this test suite and skip all others, change `describe` to `fdescribe`:

```
fdescribe('Example spec', () => {  
  it('one spec', () => { /* ... */ });  
  it('another spec', () => { /* ... */ });  
});
```

FIT

If you want Jasmine to run only one spec, change `it` to `fit`:

```
describe('Example spec', () => {  
  fit('one spec', () => { /* ... */ });  
  it('another spec', () => { /* ... */ });  
});
```

This improves the developing experience tremendously.

The Webpack module bundler still re-emits the whole bundle even if you have only changed one line of code and even if there is a test focus on one suite.

BUNDLE ONE FILE

In this case, you can instruct `ng test` to consider only the file you are currently working on. Webpack then includes all its dependencies, like the Angular framework, but not more.

For example, to include only tests called `counter.component.spec.ts`, we call `ng test` with the `--include` option.

```
ng test --include **/counter.component.spec.ts
```

`**/counter.component.spec.ts` means all files called `counter.component.spec.ts` in any subdirectory.

The bundling is now fast and the feedback is almost instant when we change implementation or test code.

Keep in mind to remove the test focus before committing your code. There are several tools that prevent `fdescribe` and `fit` from being committed.

 [Jasmine API reference: fdescribe](#)

 [Jasmine API reference: fit](#)

- 🔗 [Angular CLI reference: ng test](#)
- 🔗 [Tim Deschryver: Don't commit focused tests](#)

Developer tools

The Jasmine test runner is just another web page made with HTML, CSS and JavaScript. This means you can debug it in the browser using the developer tools.

FAMILIAR DEBUGGING TOOLS

Focus the browser window and open the developer tools. In Chrome, Firefox and Edge, you can use the F12 key.

You can use the developer tools to:

- Write debug output to the console using `console.log`, `console.debug` and friends.
- Use the JavaScript debugger. You can either set breakpoints in the developer tools or place a `debugger` statement.
- Inspect the DOM of rendered Components.

Debug output and the JavaScript debugger

The most primitive tool, `console.log`, is in fact invaluable when debugging tests. You can place debug output both in the test code and the implementation code.

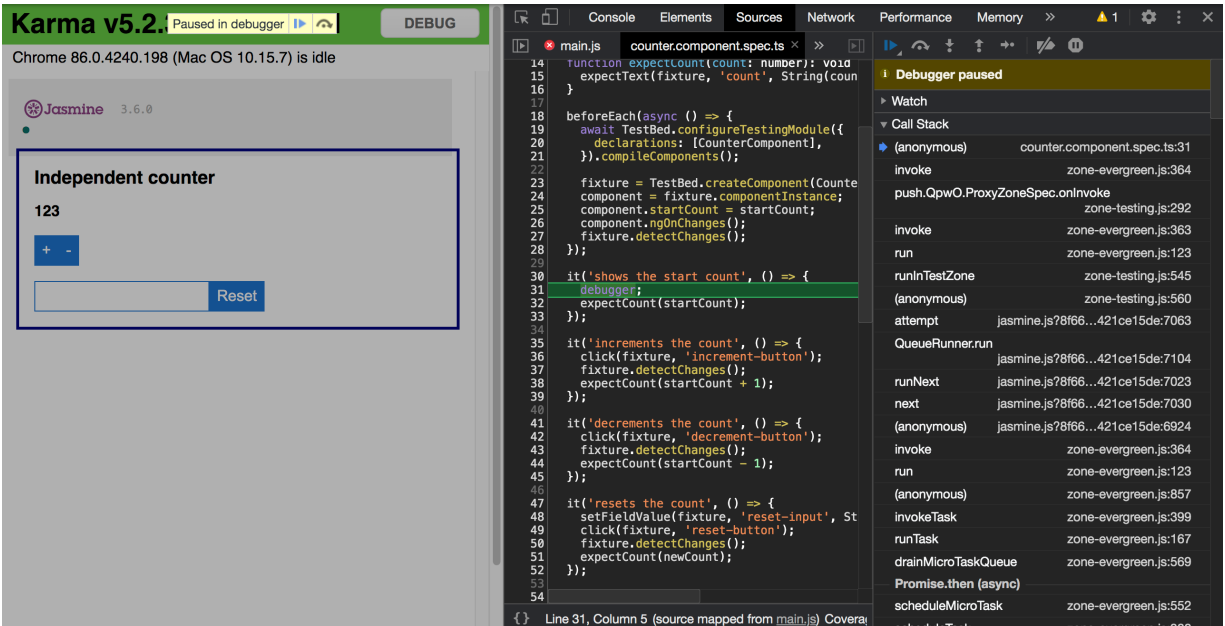
VERSATILE `console.log`

Use debug output to answer these questions:

- Is the test, suite, spec run at all?
- Does the test execution reach the log command?
- Did the test call the class, method, function under test correctly?
- Are callbacks called correctly? Do Promises complete or fail? Do Observables emit, complete or error?
- For Component tests:
 - Is Input data passed correctly?
 - Are the lifecycle methods called correctly?

DEBUGGER

Some people prefer to use **debugger** instead of console output.



While the debugger certainly gives you more control, it halts the JavaScript execution. It may disturb the processing of asynchronous JavaScript tasks and the order of execution.

ASYNCHRONOUS LOGGING

The **console** methods have their own pitfalls. For performance reasons, browsers do not write the output to the console synchronously, but asynchronously.

If you output a complex object with **console.log(object)**, most browsers render an interactive representation of the object on the console. You can click on the object to inspect its properties.

```
const exampleObject = { name: 'Usagi Tsukino' };  
console.log(exampleObject);
```

It is important to know that the rendering happens asynchronously. If you change the object shortly after, you might see the changed object, not the object at the time of the `console.log` call.

```
const exampleObject = { name: 'Usagi Tsukino' };  
console.log(exampleObject);  
exampleObject.name = 'Sailor Moon';
```

On the console, the object representation may show `name: 'Sailor Moon'` instead of `name: 'Usagi Tsukino'`.

One way to prevent this confusion is to create a snapshot of the object. You convert the object to a JSON string:

```
const exampleObject = { name: 'Usagi Tsukino' };  
console.log(JSON.stringify(exampleObject, null, '  '));  
exampleObject.name = 'Sailor Moon';
```

LOG A SNAPSHOT

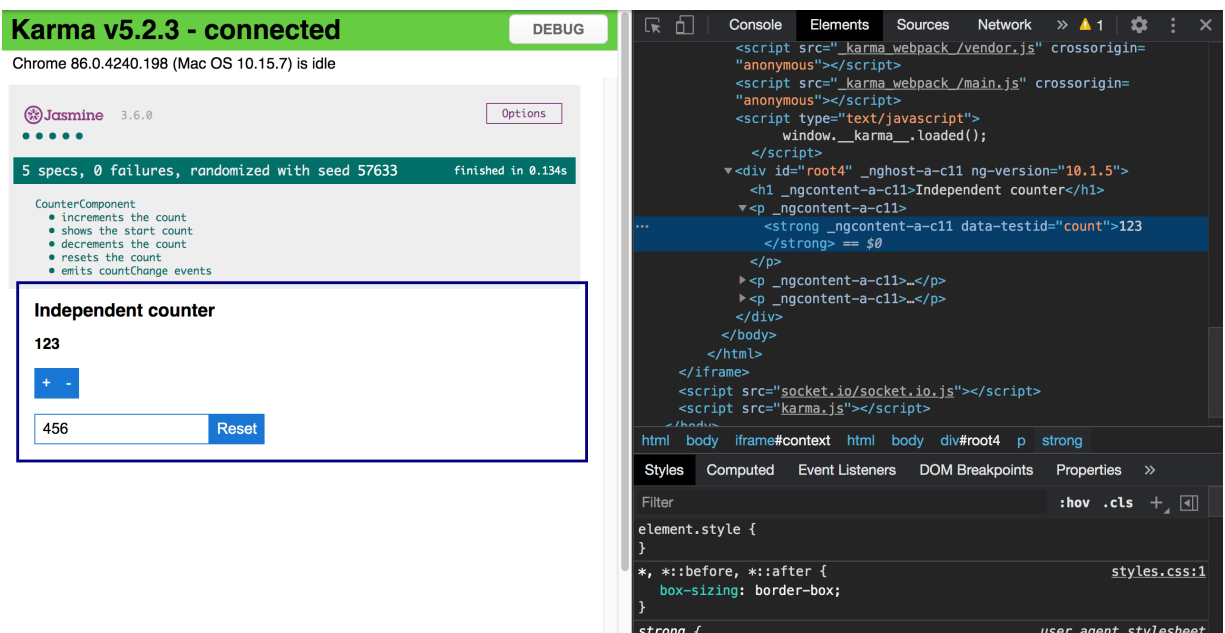
If you want an interactive representation on the console, create a copy of the object with `JSON.stringify` followed by `JSON.parse`:

```
const exampleObject = { name: 'Usagi Tsukino' };  
console.log(JSON.parse(JSON.stringify(exampleObject)));  
exampleObject.name = 'Sailor Moon';
```

Obviously, this only works for objects that can be serialized as JSON.

Inspect the DOM

In the next chapter, we will learn how to test Components. These tests will render the Component into the DOM of the Jasmine test runner page. This means you can briefly see the states of the rendered Component in the browser.



In the screenshot above, you see the rendered Component on the left side and the inspected DOM on the right side.

ROOT ELEMENT

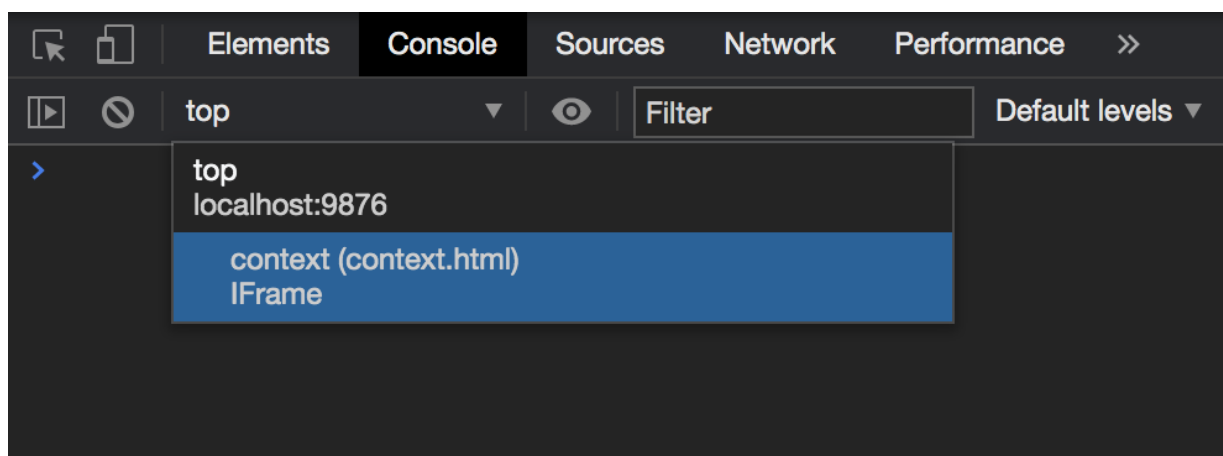
The Component's root element is rendered into the last element in the document, below the Jasmine reporter output. Make sure to set a focus on a single spec to see the rendered Component.

The rendered Component is interactive. For example, you can click on buttons and the click handlers will be called. But as we will learn later, there is no automatic change detection in the testing environment. So you might not see the effect of the interaction.

Jasmine debug runner

The Karma page at <http://localhost:9876> loads an iframe with the actual Jasmine instance, <http://localhost:9876/context.html>. This iframe complicates debugging because the developer tools operate on the topmost document per default.

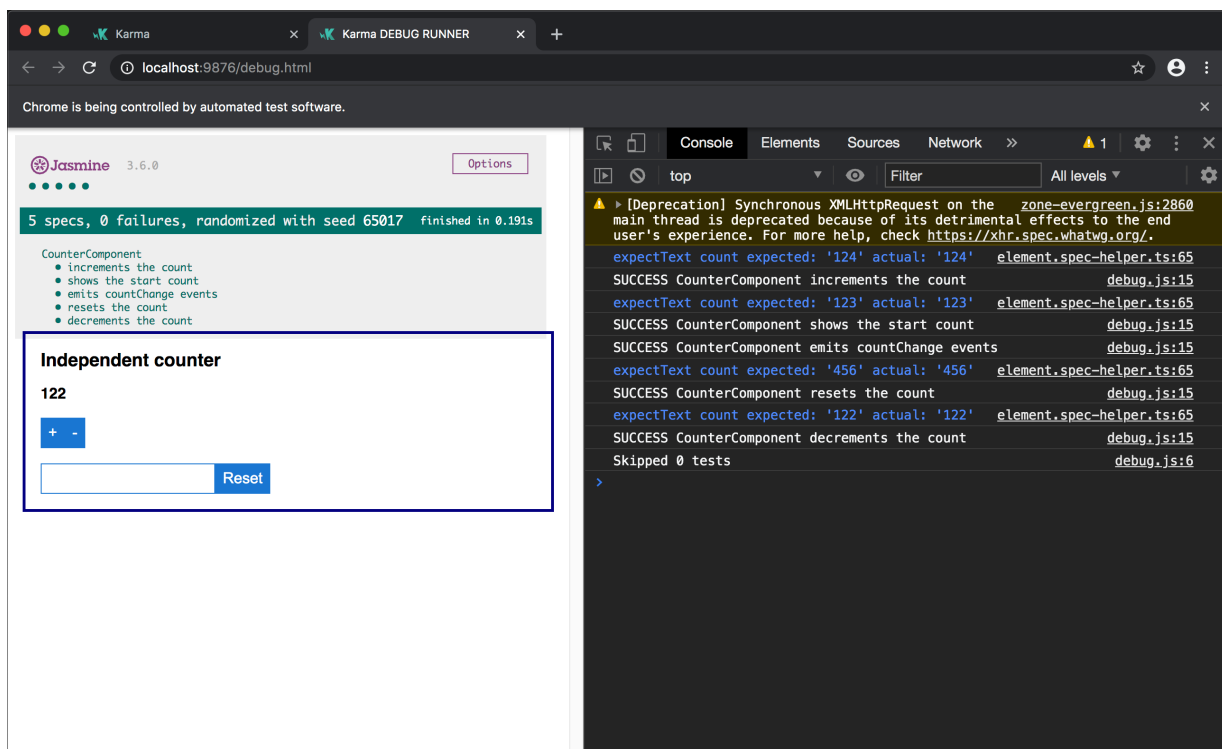
In the developer tools, you can select the iframe window context (Chrome is pictured):



This way you can access global objects and the DOM of the document where the tests run.

DEBUG RUNNER WITHOUT IFRAME

Another helpful feature is Karma's debug test runner. Click on the big "DEBUG" button on the top-right. Then a new tab opens with <http://localhost:9876/debug.html>.



The debug test runner does not have an iframe, it loads Jasmine directly. Also it automatically logs spec runs on the shell.

If you change the test or implementation code, the debug runner does not re-run the tests. You have to reload the page manually.

Testing Components

LEARNING OBJECTIVES

- Setting up a Component test using Angular's testing Module
- Getting familiar with Angular's Component testing abstractions
- Accessing the rendered DOM and checking text content
- Simulating user input like clicks and form field input
- Testing Component Input and Outputs
- Using helpers functions for common Component testing tasks

Components are the power houses of an Angular application.
Components are composed to form the user interface.

A Component deals with several concerns, among others:

- It renders the template into the HTML DOM.
- It accepts data from parent Components using Input properties.
- It emits data to parent Components using Outputs.
- It reacts to user input by registering event handlers.
- It renders the content (`ng-content`) and templates (`ng-template`) that are passed.
- It binds data to form controls and allows the user to edit the data.

- It talks to Services or other state managers.
- It uses routing information like the current URL and URL parameters.

All these tasks need to be tested properly.

Unit test for the counter Component

As a first example, we are going to test the [CounterComponent](#).

When designing a Component test, the guiding questions are:
What does the Component do, what needs to be tested? How do I test this behavior?

COUNTER FEATURES

We will test the following features of the [CounterComponent](#):

- It displays the current count. The initial value is 0 and can be set by an Input.
- When the user activates the "+" button, the count increments.
- When the user activates the "-" button, the count decrements.
- When the user enters a number into the reset input field and activates the reset button, the count is set to the given value.

- When the user changes the count, an Output emits the new count.

Writing down what the Component does already helps to structure the unit test. The features above roughly translate to specs in a test suite.

 [CounterComponent: full code](#)

TestBed

Several chores are necessary to render a Component in Angular, even the simple counter Component. If you look into the [main.ts](#) and the [AppModule](#) of a typical Angular application, you find that a “platform” is created, a Module is declared and this Module is bootstrapped.

The Angular compiler translates the templates into JavaScript code. To prepare the rendering, an instance of the Component is created, dependencies are resolved and injected, inputs are set.

Finally, the template is rendered into the DOM. For testing, you could do all that manually, but you would need to dive deeply into Angular internals.

TESTBED

Instead, the Angular team provides the **TestBed** to ease unit testing. The **TestBed** creates and configures an Angular environment so you can test particular application parts like Components and Services safely and easily.

🔗 [Angular API reference: TestBed](#)

🔗 [Testing Utility APIs: TestBed](#)

Configuring the testing Module

The **TestBed** comes with a testing Module that is configured like normal Modules in your application: You can declare Components, Directives and Pipes, provide Services and other Injectables as well as import other Modules. **TestBed** has a static method **configureTestingModule** that accepts a Module definition:

```
TestBed.configureTestingModule({  
  imports: [ /*... */ ],  
  declarations: [ /*... */ ],  
  providers: [ /*... */ ],  
});
```

DECLARE WHAT IS NECESSARY

In a unit test, add those parts to the Module that are strictly necessary: the code under test, mandatory dependencies and fakes. For example, when writing a unit test for

`CounterComponent`, we need to declare that Component class. Since the Component does not have dependencies, does not render other Components, Directives or Pipes, we are done.

```
TestBed.configureTestingModule({
  declarations: [CounterComponent],
});
```

Our Component under test is now part of a Module. We are ready to render it, right? Not yet. First we need to compile all declared Components, Directives and Pipes:

```
TestBed.compileComponents();
```

This instructs the Angular compiler to translate the template files into JavaScript code.

CONFIGURE AND COMPILE

Since `configureTestingModule` returns the `TestBed` again, we can chain those two calls:

```
TestBed
  .configureTestingModule({
    declarations: [CounterComponent],
  })
  .compileComponents();
```

You will see this pattern in most Angular tests that rely on the `TestBed`.

Rendering the Component

Now we have a fully-configured testing Module with compiled Components. Finally, we can render the Component under test using `createComponent`:

```
const fixture = TestBed.createComponent(CounterComponent);
```

`createComponent` returns a `ComponentFixture`, essentially a wrapper around the Component with useful testing tools. We will learn more about the `ComponentFixture` later.

`createComponent` renders the Component into a `div` container element in the HTML DOM. Alas, something is missing. The Component is not fully rendered. All the static HTML is present, but the dynamic HTML is missing. The template bindings, like `{{ count }}` in the example, are not evaluated.

MANUAL CHANGE DETECTION

In our testing environment, there is **no automatic change detection**. Even with the default change detection strategy, a Component is not automatically rendered and re-rendered on updates.

In testing code, we have to **trigger the change detection manually**. This might be a nuisance, but it is actually a feature. It

allows us to test asynchronous behavior in a synchronous manner, which is much simpler.

So the last thing we need to do is to trigger change detection:

```
fixture.detectChanges();
```

 [Angular API reference: ComponentFixture](#)

TestBed and Jasmine

The code for rendering a Component using the `TestBed` is now complete. Let us wrap the code in a Jasmine test suite.

```
describe('CounterComponent', () => {  
  let fixture: ComponentFixture<CounterComponent>;  
  
  beforeEach(async () => {  
    await TestBed.configureTestingModule({  
      declarations: [CounterComponent],  
    }).compileComponents();  
  
    fixture = TestBed.createComponent(CounterComponent);  
    fixture.detectChanges();  
  });  
  
  it('...', () => {  
    /* ... */  
  });  
});
```

Using `describe`, we define a test suite for the `CounterComponent`. It contains a `beforeEach` block that configures the `TestBed` and renders the Component.

ASYNC COMPILATION

You might wonder why the function passed to `beforeEach` is marked as an `async` function. It is because `compileComponents` is an asynchronous operation. To compile the Components, Angular needs to fetch external the template files referenced by `templateUrl`.

If you are using the Angular CLI, which is most likely, the template files are already included in the test bundle. So they are available instantly. If you are not using the CLI, the files have to be loaded asynchronously.

This is an implementation detail that might change in the future. The safe way is wait for `compileComponents` to complete.

ASYNC AND AWAIT

Per default, Jasmine expects that your testing code is synchronous. The functions you pass to `it` but also `beforeEach`, `beforeAll`, `afterEach`, `afterAll` need to finish in a certain amount of time, also known as timeout. Jasmine also supports asynchronous specs. If you pass an `async` function, Jasmine waits for it to finish.

ComponentFixture and DebugElement

`TestBed.createComponent(CounterComponent)` returns a fixture, an instance of `ComponentFixture`. What is the fixture and what does it provide?

The term fixture is borrowed from real-world testing of mechanical parts or electronic devices. A fixture is a standardized frame into which the test object is mounted. The fixture holds the object and connects to electrical contacts in order to provide power and to take measurements.

COMPONENTFIXTURE

In the context of Angular, the `ComponentFixture` holds the Component and provides a convenient interface to both the Component instance and the rendered DOM.

The fixture references the Component instance via the `componentInstance` property. In our example, it contains a `CounterComponent` instance.

```
const component = fixture.componentInstance;
```

The Component instance is mainly used to set Inputs and subscribe to Outputs, for example:

```
// This is a ComponentFixture<CounterComponent>
const component = fixture.componentInstance;
// Set Input
component.startCount = 10;
// Subscribe to Output
component.countChange.subscribe((count) => {
  /* ... */
});
```

We will learn more on testing Inputs and Outputs later.

DEBUGELEMENT

For accessing elements in the DOM, Angular has another abstraction: The `DebugElement` wraps the native DOM element. The fixture's `debugElement` property returns the Component's host element. For the `CounterComponent`, this is the `app-counter` element.

```
const { debugElement } = fixture;
```

The `DebugElement` offers handy properties like `properties`, `attributes`, `classes` and `styles` to examine the DOM element itself. The properties `parent`, `children` and `childNodes` help navigating in the DOM tree. They return `DebugElements` as well.

NATIVEELEMENT

Often it is necessary to unwrap the `DebugElement` to access the native DOM element inside. Every `DebugElement` has a `nativeElement` property:

```
const { debugElement } = fixture;
const { nativeElement } = debugElement;
console.log(nativeElement.tagName);
console.log(nativeElement.textContent);
console.log(nativeElement.innerHTML);
```

`nativeElement` is typed as `any` because Angular does not know the exact type of the wrapped DOM element. Most of the time, it is a subclass of `HTMLElement`.

When you use `nativeElement`, you need to learn about the DOM interface of the specific element. For example, a `button` element is represented as `HTMLButtonElement` in the DOM.

 [Angular API reference: ComponentFixture](#)

 [Angular API reference: DebugElement](#)

Writing the first Component spec

We have compiled a test suite that renders the `CounterComponent`. We have met Angular's primary testing abstractions: `TestBed`, `ComponentFixture` and `DebugElement`.

Now let us roll up our sleeves and write the first spec! The main feature of our little counter is the ability to increment the count. Hence the spec:

```
it('increments the count', () => {  
  /* ... */  
});
```

The **Arrange, Act and Assert** phases help us to structure the spec:

- We have already covered the *Arrange* phase in the `beforeEach` block that renders the Component.
- In the *Act* phase, we click on the increment button.
- In the *Assert* phase, we check that the displayed count has incremented.

```
it('increments the count', () => {  
  // Act: Click on the increment button  
  // Assert: Expect that the displayed count now reads "1".  
});
```

To click on the increment button, two actions are necessary:

1. Find the increment button element in the DOM.
2. Fire a click event on it.

Let us learn about finding elements in the DOM first.

Querying the DOM with test ids

Every `DebugElement` features the methods `query` and `queryAll` for finding descendant elements (children, grandchildren and so forth).

QUERY AND QUERYALL

- `query` returns the first descendant element that meets a condition.
- `queryAll` returns an array of all matching elements.

Both methods expect a predicate, that is a function judging every element and returning `true` or `false`.

BY.CSS

Angular ships with predefined predicate functions to query the DOM using familiar CSS selectors. For this purpose, pass `By.css('...')` with a CSS selector to `query` and `queryAll`.

```
const { debugElement } = fixture;
// Find the first h1 element
const h1 = debugElement.query(By.css('h1'));
// Find all elements with the class .user
const userElements = debugElement.queryAll(By.css('.user'));
```

The return value of `query` is a `DebugElement` again, that of `queryAll` is an array of `DebugElements` (`DebugElement[]`) in

TypeScript notation).

In the example above, we have used a type selector (`h1`) and a class selector (`.user`) to find elements in the DOM. For everyone familiar with CSS, this is familiar as well.

While these selectors are fine when styling Components, using them in a test needs to be challenged.

AVOID TIGHT COUPLING

Type and class selectors introduce a *tight coupling* between the test and the template. HTML elements are picked for semantic reasons. Classes are picked mostly for styling. Both change frequently when the Component template is refactored. Should the test fail if the element type or class changes?

Sometimes the element type and the class are crucial for the feature under test. But most of the time, they are not relevant for the feature. The test should better find the element by a feature that never changes and that bears no additional meaning: test ids.

TEST IDS

A **test id** is an identifier given to an element just for the purpose of finding it in a test. The test will still find the element if the element type or unrelated attributes change.

The preferred way to mark an HTML element is a [data attribute](#). In contrast to element types, `class` or `id` attributes, data attributes do not come with any predefined meaning. Data attributes never clash with each other.

DATA-TESTID

For the purpose of this guide, we use the **data-testid** attribute. For example, we mark the increment button in the `CounterComponent` with `data-testid="increment-button"`:

```
<button (click)="increment()" data-testid="increment-button">+</button>
```

In the test, we use the corresponding attribute selector:

```
const incrementButton = debugElement.query(
  By.css('[data-testid="increment-button"]')
);
```

ESTABLISH A CONVENTION

There is a nuanced discussion around the best way to find elements during testing. Certainly, there are several valid and elaborate approaches. This guide will only present one possible approach that is simple and approachable.

The Angular testing tools are neutral when it comes to DOM querying. They tolerate different approaches. After consideration, you should opt for one specific solution, document it as a [testing convention](#) and apply it consistently across all tests.

Triggering event handlers

Now that we have marked and got hold of the increment button, we need to click on it.

It is a common task in tests to simulate user input like clicking, typing in text, moving pointers and pressing keys. From an Angular perspective, user input causes DOM events.

The Component template registers event handlers using the schema `(event)="handler($event)"`. In the test, we need to simulate an event to call these handlers.

TRIGGER EVENT HANDLER

`DebugElement` has a useful method for firing events: `triggerEventHandler`. This method calls all event handlers for a given event type like `click`. As a second parameter, it expects a fake event object that is passed to the handlers:

```
incrementButton.triggerEventHandler('click', {  
  /* ... Event properties ... */  
});
```

This example fires a `click` event on the increment button. Since the template contains `(click)="increment()"`, the `increment`

method of `CounterComponent` will be called.

EVENT OBJECT

The `increment` method does not access the event object. The call is simply `increment()`, not `increment($event)`. Therefore, we do not need to pass a fake event object, we can simply pass `null`:

```
incrementButton.triggerEventHandler('click', null);
```

It is worth noting that `triggerEventHandler` does not dispatch a synthetic DOM event. The effect stays on the `DebugElement` abstraction level and does not touch the native DOM.

NO BUBBLING

This is fine as long as the event handler is registered on the element itself. If the event handler is registered on a parent and relies on event bubbling, you need to call `triggerEventHandler` directly on that parent. `triggerEventHandler` does not simulate event bubbling or any other effect a real event might have.

Expecting text output

We have completed the *Act* phase in which the test clicks on the increment button. In the *Assert* phase, we need to expect that the displayed count changes from “0” to “1”.

In the template, the count is rendered into a `strong` element:

```
<strong>{{ count }}</strong>
```

FIND BY TEST ID

In our test, we need to find this element and read its text content. For this purpose, we add a test id:

```
<strong data-testid="count">{{ count }}</strong>
```

We can now find the element as usual:

```
const countOutput = debugElement.query(  
  By.css('[data-testid="count"]')  
);
```

TEXT CONTENT

The next step is to read the element's content. In the DOM, the count is a text node that is a child of `strong`.

Unfortunately, the `DebugElement` does not have a method or property for reading the text content. We need to access the native DOM element that has a convenient `textContent` property.

```
countOutput.nativeElement.textContent
```

Finally, we expect that this string is `"1"` using Jasmine's `expect`:

```
expect(countOutput.nativeElement.textContent).toBe('1');
```

The `counter.component.spec.ts` now looks like this:

```

/* Incomplete! */
describe('CounterComponent', () => {
  let fixture: ComponentFixture<CounterComponent>;
  let debugElement: DebugElement;

  // Arrange
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [CounterComponent],
    }).compileComponents();

    fixture = TestBed.createComponent(CounterComponent);
    fixture.detectChanges();
    debugElement = fixture.debugElement;
  });

  it('increments the count', () => {
    // Act
    const incrementButton = debugElement.query(
      By.css('[data-testid="increment-button"]')
    );
    incrementButton.triggerEventHandler('click', null);

    // Assert
    const countOutput = debugElement.query(
      By.css('[data-testid="count"]')
    );
    expect(countOutput.nativeElement.textContent).toBe('1');
  });
});

```

When we run that suite, the spec fails:

CounterComponent increments the count FAILED

Error: Expected '0' to be '1'.

What is wrong here? Is the implementation faulty? No, the test just missed something important.

MANUAL CHANGE DETECTION

We have mentioned that in the testing environment, Angular does not automatically detect changes in order to update the DOM. Clicking the increment button changes the `count` property of the Component instance. To update the template binding `{{ count }}`, we need to *trigger the change detection manually*.

```
fixture.detectChanges();
```

The full test suite now looks like this:

```
describe('CounterComponent', () => {
  let fixture: ComponentFixture<CounterComponent>;
  let debugElement: DebugElement;

  // Arrange
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [CounterComponent],
    }).compileComponents();

    fixture = TestBed.createComponent(CounterComponent);
    fixture.detectChanges();
    debugElement = fixture.debugElement;
  });
```

```
it('increments the count', () => {  
  // Act  
  const incrementButton = debugElement.query(  
    By.css('[data-testid="increment-button"]')  
  );  
  incrementButton.triggerEventHandler('click', null);  
  // Re-render the Component  
  fixture.detectChanges();  
  
  // Assert  
  const countOutput = debugElement.query(  
    By.css('[data-testid="count"]')  
  );  
  expect(countOutput.nativeElement.textContent).toBe('1');  
});
```

Congratulations! We have written our first Component test. It is not complete yet, but it already features a typical workflow. We will make small improvements to the existing code with each spec we add.

 [CounterComponent: test code](#)

Testing helpers

The next `CounterComponent` feature we need to test is the decrement button. It is very similar to the increment button, so the spec looks almost the same.

First, we add a test id to the decrement button:

```
<button (click)="decrement()" data-testid="decrement-button">-</button>
```

Then we write the spec:

```
it('decrements the count', () => {  
  // Act  
  const decrementButton = debugElement.query(  
    By.css('[data-testid="decrement-button"]')  
  );  
  decrementButton.triggerEventHandler('click', null);  
  // Re-render the Component  
  fixture.detectChanges();  
  
  // Assert  
  const countOutput = debugElement.query(  
    By.css('[data-testid="count"]')  
  );  
  expect(countOutput.nativeElement.textContent).toBe('-1');  
});
```

There is nothing new here, only the test id, the variable names and the expected output changed.

REPEATING PATTERNS

Now we have two specs that are almost identical. The code is repetitive and the signal-to-noise ratio is low, meaning there is much code that does little. Let us identify the patterns repeated here:

1. Finding an element by test id
2. Clicking on an element found by test id
3. Expecting a given text content on an element found by test id

These tasks are highly generic and they will appear in almost every Component spec. It is worth writing testing helpers for them.

TESTING HELPERS

A **testing helper** is a piece of code that makes writing tests easier. It makes test code more concise and more meaningful. Since a spec should describe the implementation, a readable spec is better than an obscure, convoluted one.

Your testing helpers should cast your [testing conventions](#) into code. They not only improve the individual test, but make sure all tests use the same patterns and work the same.

A testing helper can be a simple function, but it can also be an abstraction class or a Jasmine extension. For the start, we extract common tasks into plain functions.

FIND BY TEST ID

First, let us write a helper for finding an element by test id. We have used this pattern multiple times:

```
const xyzElement = fixture.debugElement.query(
  By.css('[data-testid="xyz"]')
);
```

We move this code into a reusable function:

```
function findEl<T>(  
  fixture: ComponentFixture<T>,  
  testId: string  
): DebugElement {  
  return fixture.debugElement.query(  
    By.css(`[data-testid="${testId}"]`)  
  );  
}
```

This function is self-contained. We need to pass in the Component fixture explicitly. Since `ComponentFixture<T>` requires a type parameter – the wrapped Component type –, `findEl` also has a type parameter called `T`. TypeScript will infer the Component type automatically when you pass a `ComponentFixture`.

CLICK

Second, we write a testing helper that clicks on an element with a given test id. This helper builds on `findEl`.

```
export function click<T>(  
  fixture: ComponentFixture<T>,  
  testId: string  
): void {  
  const element = findEl(fixture, testId);
```

```
const event = makeClickEvent(element.nativeElement);
element.triggerEventHandler('click', event);
}
```

To create a fake click event object, `click` calls another function, `makeClickEvent`.

```
export function makeClickEvent(
  target: EventTarget
): Partial<MouseEvent> {
  return {
    preventDefault(): void {},
    stopPropagation(): void {},
    stopImmediatePropagation(): void {},
    type: 'click',
    target,
    currentTarget: target,
    bubbles: true,
    cancelable: true,
    button: 0
  };
}
```

This function returns a partial [MouseEvent](#) fake object with the most important methods and properties of real click events. It is suitable for clicks on buttons and links when the pointer position and modifier keys do not matter.

CLICK MEANS ACTIVATE

The `click` testing helper can be used on every element that has a `(click)="..."` event handler. For accessibility, make sure the

element can be focussed and activated. This is already the case for buttons (`button` element) and links (`a` elements).

Historically, the `click` event was only triggered by mouse input. Today, it is a generic “activate” event. It is also triggered by touch input (“tap”), keyboard input or voice input.

So in your Component, you do not need to listen for touch or keyboard events separately. In the test, a generic `click` event usually suffices.

EXPECT TEXT CONTENT

Third, we write a testing helper that expects a given text content on an element with a given test id.

```
export function expectText<T>(  
  fixture: ComponentFixture<T>,  
  testId: string,  
  text: string,  
) : void {  
  const element = findEl(fixture, testId);  
  const actualText = element.nativeElement.textContent;  
  expect(actualText).toBe(text);  
}
```

Again, this is a simple implementation we will improve later.

Using these helpers, we rewrite our spec:

```
it('decrements the count', () => {  
  // Act  
  click(fixture, 'decrement-button');  
  // Re-render the Component  
  fixture.detectChanges();  
  
  // Assert  
  expectText(fixture, 'count', '-1');  
});
```

That is much better to read and less to write! You can tell what the spec is doing at first glance.

 [CounterComponent: test code](#)

 [Element spec helpers: full code](#)

Filling out forms

We have tested the increment and decrement button successfully. The remaining user-facing feature we need to test is the reset feature.

In the user interface, there is a reset input field and a reset button. The user enters a new number into the field, then clicks on the button. The Component resets the count to the user-provided number.

SET FIELD VALUE

We already know how to click a button, but how do we fill out a form field? Unfortunately, Angular's testing tools do not provide a solution for filling out forms easily.

The answer depends on the field type and value. The generic answer is: Find the native DOM element and set the `value` property to the new value.

For the reset input, this means:

```
const resetInput = debugElement.query(
  By.css('[data-testid="reset-input"]')
);
resetInput.nativeElement.value = '123';
```

With our testing helper:

```
const resetInputEl = findEl(fixture, 'reset-input').nativeElement;
resetInputEl.value = '123';
```

This fills in the value programmatically.

In `CounterComponent`'s template, the reset input has a *template reference variable*, `#resetInput`:

```
<input type="number" #resetInput data-testid="reset-input" />
<button (click)="reset(resetInput.value)" data-testid="reset-button">
  Reset
</button>
```

The click handler uses `resetInput` to access the `input` element, reads the `value` and passes it to the `reset` method.

The example already works because the form is very simple. Setting a field's `value` is not a full simulation of user input and will not work with Template-driven or Reactive Forms yet.

FAKE `INPUT` EVENT

Angular forms cannot observe `value` changes directly. Instead, Angular listens for an `input` event that the browser fires when a field value changes.

For **compatibility with Template-driven and Reactive Forms**, we need to dispatch a fake `input` event. Such events are also called *synthetic events*.

In newer browsers, we create a fake `input` event with `new Event('input')`. To dispatch the event, we use the `dispatchEvent` method of the target element.

```
const resetInputEl = findEl(fixture, 'reset-input').nativeElement;
resetInputEl.value = '123';
resetInputEl.dispatchEvent(new Event('input'));
```

If you need to run your tests in legacy Internet Explorer, a bit more code is necessary. Internet Explorer does not support `new Event('...')`, but the `document.createEvent` method:

```
const event = document.createEvent('Event');
event.initEvent('input', true, false);
resetInputEl.dispatchEvent(event);
```

The full spec for the reset feature then looks like this:

```
it('resets the count', () => {
  const newCount = '123';

  // Act
  const resetInputEl = findEl(fixture, 'reset-input').nativeElement;
  // Set field value
  resetInputEl.value = newCount;
  // Dispatch input event
  const event = document.createEvent('Event');
  event.initEvent('input', true, false);
  resetInputEl.dispatchEvent(event);

  // Click on reset button
  click(fixture, 'reset-button');
  // Re-render the Component
  fixture.detectChanges();

  // Assert
  expectText(fixture, 'count', newCount);
});
```

Filling out forms is a common task in tests, so it makes sense to extract the code and put it into a helper.

HELPER FUNCTIONS

The helper function `setFieldValue` takes a Component fixture, a test id and a string value. It finds the corresponding element using `findEl`. Using another helper, `setFieldElementValue`, it sets the `value` and dispatches an `input` event.

```
export function setFieldValue<T>(  
  fixture: ComponentFixture<T>,  
  testId: string,  
  value: string,  
)< void {  
  setFieldElementValue(  
    findEl(fixture, testId).nativeElement,  
    value  
  );  
}
```

You can find the full source code of the involved helper functions in [element.spec-helper.ts](#).

Using the newly created `setFieldValue` helper, we can simplify the spec:

```
it('resets the count', () => {  
  const newCount = '123';  
  
  // Act  
  setFieldValue(fixture, 'reset-input', newCount);  
  click(fixture, 'reset-button');  
  fixture.detectChanges();  
  
  // Assert
```

```
    expectText(fixture, 'count', newCount);  
  });
```

While the reset feature is simple, this is how to test most form logic. Later, we will learn how to [test complex forms](#).

INVALID INPUT

The `CounterComponent` checks the input value before it resets the count. If the value is not a number, clicking the reset button does nothing.

We need to cover this behavior with another spec:

```
it('does not reset if the value is not a number', () => {  
  const value = 'not a number';  
  
  // Act  
  setFieldValue(fixture, 'reset-input', value);  
  click(fixture, 'reset-button');  
  fixture.detectChanges();  
  
  // Assert  
  expectText(fixture, 'count', startCount);  
});
```

The small difference in this spec is that we set the field value to “not a number”, a string that cannot be parsed as a number, and expect the count to remain unchanged.

This is it! We have tested the reset form with both valid and invalid input.

 [CounterComponent: test code](#)

 [Element spec helpers: full code](#)

Testing Inputs

`CounterComponent` has an Input `startCount` that sets the initial count. We need to test that the counter handles the Input properly.

For example, if we set `startCount` to `123`, the rendered count needs to be `123` as well. If the Input is empty, the rendered count needs to be `0`, the default value.

SET INPUT VALUE

An Input is a special property of the Component instance. We can set this property in the *Arrange* phase.

```
const component = fixture.componentInstance;  
component.startCount = 10;
```

It is a good practice not to change an Input value within a Component. An Input property should always reflect the data passed in by the parent Component.

INPUT VS. COMPONENT STATE

That is why `CounterComponent` has a public Input named `startCount` as well as an internal property named `count`. When

the user clicks the increment or decrement buttons, `count` is changed, but `startCount` remains unchanged.

Whenever the `startCount` Input changes, `count` needs to be set to `startCount`. The safe place to do that is the `ngOnChanges` lifecycle method:

```
public ngOnChanges(): void {  
    this.count = this.startCount;  
}
```

`ngOnChanges` is called whenever a “data-bound property” changes, including Inputs.

Let us write a test for the `startCount` Input. We set the Input in the `beforeEach` block, before calling `detectChanges`. The spec itself checks that the correct count is rendered.

```
/* Incomplete! */  
beforeEach(async () => {  
    /* ... */  
  
    // Set the Input  
    component.startCount = startCount;  
    fixture.detectChanges();  
});  
  
it('shows the start count', () => {  
    expectText(fixture, 'count', String(count));  
});
```

When we run this spec, we find that it fails:

```
CounterComponent > shows the start count  
Expected '0' to be '123'.
```

NGONCHANGES

What is wrong here? Did we forget to call `detectChanges` again?

No, but we forgot to call `ngOnChanges`!

In the testing environment, `ngOnChanges` is not called automatically. We have to call it manually after setting the Input.

Here is the corrected example:

```
describe('CounterComponent', () => {  
  let component: CounterComponent;  
  let fixture: ComponentFixture<CounterComponent>;  
  
  const startCount = 123;  
  
  beforeEach(async () => {  
    await TestBed.configureTestingModule({  
      declarations: [CounterComponent],  
    }).compileComponents();  
  
    fixture = TestBed.createComponent(CounterComponent);  
    component = fixture.componentInstance;  
    component.startCount = startCount;  
    // Call ngOnChanges, then re-render  
    component.ngOnChanges();  
    fixture.detectChanges();  
  });
```



```
/* ... */  
  
it('shows the start count', () => {  
  expectText(fixture, 'count', String(startCount));  
});  
});
```

The `CounterComponent` expects a `number` Input and renders it into the DOM. When reading text from the DOM, we always deal with strings. That is why we pass in a number `123` but expect to find the string `'123'`.

 [CounterComponent: test code](#)

Testing Outputs

While Inputs pass data from parent to child, Outputs send data from child to parent. In combination, a Component can perform a specific operation just with the required data.

For example, a Component may render a form so the user can edit or review the data. Once completed, the Component emits the data as an Output.

Outputs are not a user-facing feature, but a vital part of the public Component API. Technically, Outputs are Component instance

properties. A unit test must inspect the Outputs thoroughly to proof that the Component plays well with other Components.

The `CounterComponent` has an output named `countChange`. Whenever the count changes, the `countChange` Output emits the new value.

```
export class CounterComponent implements OnChanges {  
  /* ... */  
  @Output()  
  public countChange = new EventEmitter<number>();  
  /* ... */  
}
```

SUBSCRIBE TO OBSERVABLE

`EventEmitter` is a subclass of RxJS `Subject`, which itself extends RxJS `Observable`. The Component uses the `emit` method to publish new values. The parent Component uses the `subscribe` method to listen for emitted values. In the testing environment, we will do the same.

Let us write a spec for the `countChange` Output!

```
it('emits countChange events on increment', () => {  
  /* ... */  
});
```

Within the spec, we access the Output via `fixture.componentInstance.countChange`. In the *Arrange*

phase, we subscribe to the `EventEmitter`.

```
it('emits countChange events on increment', () => {
  // Arrange
  component.countChange.subscribe((count) => {
    /* ... */
  });
});
```

We need to verify that the observer function is called with the right value when the increment button is clicked. In the *Act* phase, we click on the button using our helper function:

```
it('emits countChange events on increment', () => {
  // Arrange
  component.countChange.subscribe((count) => {
    /* ... */
  });

  // Act
  click(fixture, 'increment-button');
});
```

CHANGE VARIABLE VALUE

In the *Assert* phase, we expect that `count` has the correct value. The easiest way is to declare a variable in the spec scope. Let us name it `actualCount`. Initially, it is `undefined`. The observer function sets a value – or not, if it is never called.

```
it('emits countChange events on increment', () => {
  // Arrange
```

```

let actualCount: number | undefined;
component.countChange.subscribe((count: number) => {
  actualCount = count;
});

// Act
click(fixture, 'increment-button');

// Assert
expect(actualCount).toBe(1);
});

```

EXPECT CHANGED VALUE

The click on the button emits the count and calls the observer function synchronously. That is why the next line of code can expect that `actualCount` has been changed.

You might wonder why we did not put the `expect` call in the observer function:

```

/* Not recommended! */
it('emits countChange events on increment', () => {
  // Arrange
  component.countChange.subscribe((count: number) => {
    // Assert
    expect(count).toBe(1);
  });

  // Act
  click(fixture, 'increment-button');
});

```

ALWAYS RUN EXPECTATION

This works as well. But if the feature under test is broken and the Output does not emit, `expect` is never called.

Per default, Jasmine warns you that the spec has no expectations but treats the spec as successful (see [Configuring Karma and Jasmine](#)). We want the spec to fail explicitly in this case, so we make sure the expectation is always run.

Now we have verified that `countChange` emits when the increment button is clicked. We also need to proof that the Output emits on decrement and reset. We can achieve that by adding two more specs that copy the existing spec:

```
it('emits countChange events on decrement', () => {  
  // Arrange  
  let actualCount: number | undefined;  
  component.countChange.subscribe((count: number) => {  
    actualCount = count;  
  });  
  
  // Act  
  click(fixture, 'decrement-button');  
  
  // Assert  
  expect(actualCount).toBe(-1);  
});  
  
it('emits countChange events on reset', () => {  
  const newCount = '123';
```

```
// Arrange
let actualCount: number | undefined;
component.countChange.subscribe((count: number) => {
  actualCount = count;
});

// Act
setFieldValue(fixture, 'reset-input', newCount);
click(fixture, 'reset-button');

// Assert
expect(actualCount).toBe(newCount);
});
```

 [CounterComponent: test code](#)

Repetitive Component specs

Testing the `countChange` Output with three specs works fine, but the code is highly repetitive. A testing helper can reduce the repetition. Experts disagree on whether repetitive testing code is a problem at all.

On the one hand, it is hard to grasp the essence of repetitive specs. Testing helpers form a custom language for expressing testing instructions clearly and briefly. For example, if your specs find DOM elements via test ids, a testing helper establishes the convention and hides the implementation details.

On the other hand, abstractions like helper functions make tests more complex and therefore harder to understand. A developer reading the specs needs to get familiar with the testing helpers first. After all, tests should be more readable than the implementation code.

DUPLICATION VS. ABSTRACTION

There is a controversial debate in software development regarding repetition and the value of abstractions. [As Sandi Metz famously stated](#), “duplication is far cheaper than the wrong abstraction”.

This is especially true when writing specs. You should try to eliminate duplication and boilerplate code with [beforeEach/beforeAll](#), simple helper functions and even testing libraries. But do not try to apply your optimization habits and skills to test code.

A test is supposed to reproduce all relevant logical cases. Finding a proper abstraction for all these diverse, sometimes mutually exclusive cases is often futile.

CAREFULLY REDUCE REPETITION

Your mileage may vary on this question. For completeness, let us discuss how to reduce the repetition in the [countChange](#) Output specs.

An Output is an `EventEmitter`, that is a fully-functional RxJS `Observable`. This allows us to transform the `Observable` as we please. Specifically, we can click all three buttons and then expect that the `countChange` Output has emitted three values.

```
it('emits countChange events', () => {  
  // Arrange  
  const newCount = 123;  
  
  // Capture all emitted values in an array  
  let actualCounts: number[] | undefined;  
  
  // Transform the Observable, then subscribe  
  component.countChange.pipe(  
    // Close the Observable after three values  
    take(3),  
    // Collect all values in an array  
    toArray()  
  ).subscribe((counts) => {  
    actualCounts = counts;  
  });  
  
  // Act  
  click(fixture, 'increment-button');  
  click(fixture, 'decrement-button');  
  setFieldValue(fixture, 'reset-input', String(newCount));  
  click(fixture, 'reset-button');  
  
  // Assert  
  expect(actualCounts).toEqual([1, 0, newCount]);  
});
```


This example requires some RxJS knowledge. We are going to encounter RxJS Observables again and again when testing Angular applications. If you do not understand the example above, that is totally fine. It is just an optional way to merge three specs into one.

 [CounterComponent: test code](#)

Black vs. white box Component testing

Component tests are most meaningful if they closely mimic how the user interacts with the Component. The tests we have written apply this principle. We have worked directly with the DOM to read text, click on buttons and fill out form fields because this is what the user does.

These tests are black box tests. We have already talked about [black box vs. white box testing](#) in theory. Both are valid testing methods. As stated, this guide advises to use black box testing first and foremost.

A common technique to enforce black box testing is to mark internal methods as `private` so they cannot be called in the test. The test should only inspect the documented, public API.

INTERNAL YET PUBLIC

In Angular Components, the difference between external and internal properties and methods does not coincide with their TypeScript visibility (`public` vs. `private`). Properties and methods need to be `public` so that the template is able to access them.

This makes sense for Input and Output properties. They need to be read and written from the outside, from your test. However, internal properties and methods exist that are `public` only for the template.

For example, the `CounterComponent` has an Input `startCount` and an Output `countChange`. Both are `public`:

```
@Input()
public startCount = 0;

@Output()
public countChange = new EventEmitter<number>();
```

They form the public API. However, there are several more properties and methods that are `public`:

```
public count = 0;
public increment(): void { /* ... */ }
public decrement(): void { /* ... */ }
public reset(newCount: string): void { /* ... */ }
```

PUBLIC FOR THE TEMPLATE

These properties and methods are internal, they are used only within the Component. Yet they need to be `public` so the template may access them. Angular compiles templates into TypeScript code, and TypeScript ensures that the template code only accesses public properties and methods.

In our `CounterComponent` black box test, we increment the count by clicking on the “+” button. In contrast, many Angular testing tutorials conduct Component white box tests. They call the `increment` method directly:

```
/* Not recommended! */
describe('CounterComponent', () => {
  /* ... */
  it('increments the count', () => {
    component.increment();
    fixture.detectChanges();
    expectText(fixture, 'count', '1');
  });
});
```

This white box test reaches into the Component to access an internal, yet `public` method. This is sometimes valuable, but most of the time it is misused.

INPUTS, OUTPUTS, DOM

As we have learned, a Component test is meaningful if it interacts with the Component via Inputs, Outputs and the rendered DOM. If the Component test calls internal methods or accesses internal

properties instead, it often misses important template logic and event handling.

The white box spec above calls the `increment` method, but does not test the corresponding template code, the increment button:

```
<button (click)="increment()" data-testid="increment-button">+</button>
```

If we remove the increment button from the template entirely, the feature is obviously broken. But the white box test does not fail.

START WITH BLACK BOX TESTS

When applied to Angular Components, black box testing is more intuitive and easier for beginners. When writing a black box test, ask what the Component does for the user and for the parent Component. Then imitate the usage in your test.

A white box test does not examine the Component strictly from the DOM perspective. Thereby, it runs the risk of missing crucial Component behavior. It gives the illusion that all code is tested.

That being said, white box testing is a viable advanced technique. Experienced testers can write efficient white box specs that still test out all Component features and cover all code.

The following table shows which properties and methods of an Angular Component you should access or not in a black box test.

RECOMMENDATION

Black box testing an Angular Component

Class member	Access from test
@Input properties	Yes (write)
@Output properties	Yes (subscribe)
Lifecycle methods	Avoid except for <code>ngOnChanges</code>
Other public methods	Avoid
Private properties and methods	No access

Testing Components with children

LEARNING OBJECTIVES

- Rendering a Component with or without its children
- Checking that the parent and its children are wired up correctly
- Replacing child Components with fakes
- Using the ng-mocks library to fake dependencies

PRESENTATIONAL COMPONENTS

So far, we have tested an independent Component that renders plain HTML elements, but no child Components. Such low-level Components are the workhorses of an Angular application.

- They directly render what the user sees and interacts with.
- They are often highly generic and reusable.
- They are controlled through Inputs and report back using Outputs.
- They have little to none dependencies.
- They are easy to reason about and therefore easy to test.
- The preferred way of testing them is a unit test.

These Components are called **presentational Components** since they directly present a part of the user interface using HTML and CSS. Presentational Components need to be combined and wired to form a working user interface.

CONTAINER COMPONENTS

This is the duty of **container Components**. These high-level Components bring multiple low-level Components together. They pull data from different sources, like Services and state managers, and distribute it to their children.

Container Components have several types of dependencies. They depend on the nested child Components, but also Injectables. These are classes, functions, objects, etc. provided via dependency injection, like Services. These dependencies make testing container Components complicated.

SHALLOW VS. DEEP RENDERING

There are two fundamental ways to test Components with children:

- A unit test using **shallow rendering**. The child Components are not rendered.
- An integration test using **deep rendering**. The child Components are rendered.

Again, both are valid approaches we are going to discuss.

Shallow vs. deep rendering

In the counter example application, the [HomeComponent](#) contains [CounterComponents](#), [ServiceCounterComponents](#) and [NgRxCounterComponents](#).

From the [template](#):

```
<app-counter
  [startCount]="5"
  (countChange)="handleCountChange($event)"
></app-counter>
<!-- ... -->
<app-service-counter></app-service-counter>
<!-- ... -->
<app-ngrx-counter></app-ngrx-counter>
```

These custom `app-*` elements end up in the DOM tree. They become the *host elements* of the child Components.

CHECK WIRING ONLY

A **unit test of [HomeComponent](#)** does not render these children. The host elements are rendered, but they remain empty. You might wonder, what is the point of such a test? What does it do after all?

From `HomeComponent`'s perspective, the inner workings of its children are not relevant. We need to test that the template contains the children. Also, we need to check that `HomeComponent` and its children are wired up correctly using Inputs and Outputs.

In particular, the `HomeComponent` unit test checks that an `app-counter` element is present, that the `startCount` Input is passed correctly and that `HomeComponent` handles the `countChange` event. The same is done for the other children, `app-service-counter` and `app-ngrx-counter`.

RENDER CHILDREN

An **integration test of `HomeComponent`** renders the child Components. The host elements are filled with the output of `CounterComponent`, `ServiceCounterComponent` and `NgRxCounterComponent`, respectively. This integration test is actually testing all four Components.

TEST COOPERATION

We need to decide the level of detail for testing the nested Components. If separate unit tests for them exist, we do not need to click on each respective increment button. After all, the integration test needs to prove that the four Component work together, without going into the child Component details.

 [HomeComponent: implementation and test code](#)

Unit test

Let us write a unit test for `HomeComponent` first. The setup looks familiar to the `CounterComponent` test suite. We are using `TestBed` to configure a testing Module and to render the Component under test.

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

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

    fixture = TestBed.createComponent(HomeComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('renders without errors', () => {
    expect(component).toBeTruthy();
  });
});
```

SMOKE TEST

This suite has one spec that acts as a *smoke test*. It checks the presence of a Component instance. It does not assert anything

specific about the Component behavior yet. It merely proves that the Component renders without errors.

If the smoke test fails, you know that something is wrong with the testing setup.

UNKNOWN CUSTOM ELEMENTS

From Angular 9 on, the spec passes but produces a bunch of warnings on the shell:

`'app-counter' is not a known element:`

1. If `'app-counter'` is an Angular component, then verify that it is part of this module.
2. If `'app-counter'` is a Web Component then add `'CUSTOM_ELEMENTS_SCHEMA'` to the `'@NgModule.schemas'` of this component to suppress this message.

We get the same warning regarding `app-service-counter` and `app-ngrx-counter`. Another warning reads:

`Can't bind to 'startCount' since it isn't a known property of 'app-counter'.`

What do these warnings mean? Angular does not recognize the custom elements `app-counter`, `app-service-counter` and `app-ngrx-counter` because we have not declared Components that match these selectors. The warning points at two solutions:

1. Either declare the child Components in the testing Module.
This turns the test into an integration test.
2. Or tell Angular to ignore the unknown elements. **This turns the test into a unit test.**

IGNORE CHILD ELEMENTS

Since we plan to write a unit test, we opt for the second.

When configuring the testing Module, we can specify `schemas` to tell Angular how to deal with elements that are not handled by Directives or Components.

The warning suggests `CUSTOM_ELEMENTS_SCHEMA`, but the elements in question are not Web Components. We want Angular to simply ignore the elements. Therefore we use the `NO_ERRORS_SCHEMA`, “a schema that allows any property on any element”.

```
await TestBed.configureTestingModule({
  declarations: [HomeComponent],
  schemas: [NO_ERRORS_SCHEMA],
}).compileComponents();
```

With this addition, our smoke test passes.

Now let us write a more meaningful spec! We start with the nested `app-counter`. This is the code we need to cover:

```
<app-counter  
  [startCount]="5"  
  (countChange)="handleCountChange($event)"  
></app-counter>
```

CHILD PRESENCE

First of all, we need to test the presence of `app-counter`, the independent counter. We create a new spec for that purpose:

```
it('renders an independent counter', () => {  
  /* ... */  
});
```

To verify that an `app-counter` element exists in the DOM, we use the familiar `query` method of the topmost `DebugElement`.

```
const { debugElement } = fixture;  
const counter = debugElement.query(By.css('app-counter'));
```

This code uses the `app-counter` type selector to find the element. You might wonder, why not use a test id and the `findEl` helper?

FIND BY ELEMENT TYPE

In this rare occasion, we need to enforce the element `app-counter` because this is `CounterComponent`'s selector.

Using a test id makes the element type arbitrary. This makes tests more robust in other case. When testing the existence of child Components though, it is the element type that invokes the child.

Our spec still lacks an expectation. The query method returns a `DebugElement` or `null`. We simply expect the return value to be truthy:

```
it('renders an independent counter', () => {
  const { debugElement } = fixture;
  const counter = debugElement.query(By.css('app-counter'));
  expect(counter).toBeTruthy();
});
```

Finding a child Component is a common task. Such repeating patterns are good candidates for testing helpers. Not because it is much code, but because the code has a specific meaning we would like to convey.

`debugElement.query(By.css('app-counter'))` is not particularly descriptive. The reader has to think for a moment to realize that the code tries to find a nested Component.

FINDCOMPONENT

So let us introduce a helper function named `findComponent`.

```
export function findComponent<T>({
  fixture: ComponentFixture<T>,
  selector: string,
}): DebugElement {
  return fixture.debugElement.query(By.css(selector));
}
```

Our spec now looks like this:

```
it('renders an independent counter', () => {  
  const counter = findComponent(fixture, 'app-counter');  
  expect(counter).toBeTruthy();  
});
```

CHECK INPUTS

The next feature we need to test is the `startCount` Input. In particular, the property binding `[startCount]="5"` in `HomeComponent`'s template. Let us create a new spec:

```
it('passes a start count', () => {  
  const counter = findComponent(fixture, 'app-counter');  
  /* ... */  
});
```

PROPERTIES

How do we read the Input value? Each `DebugElement` has a `properties` object that contains DOM properties together with its values. In addition, it contains certain property bindings. (The type is `{ [key: string]: any }`).

In a unit test with shallow rendering, `properties` contains the Inputs of a child Component. First, we find `app-counter` to obtain the corresponding `DebugElement`. Then we check the Input value, `properties.startCount`.

```
it('passes a start count', () => {  
  const counter = findComponent(fixture, 'app-counter');
```

```
expect(counter.properties.startCount).toBe(5);
});
```

That was quite easy! Last but not least, we need to test the Output.

OUTPUT EVENT

From `HomeComponent`'s perspective, reacting to the Output is like handling an event on the `app-counter` element. The template uses the familiar `(event)="handler($event)"` syntax:

```
<app-counter
  [startCount]="5"
  (countChange)="handleCountChange($event)"
></app-counter>
```

The `handleCountChange` method is defined in the Component class. It simply calls `console.log` to prove that the child-parent communication worked:

```
export class HomeComponent {
  public handleCountChange(count: number): void {
    console.log('countChange event from CounterComponent', count);
  }
}
```

Let us add a new spec for testing the Output:

```
it('listens for count changes', () => {
  /* ... */
});
```


The spec needs to do two things:

1. *Act*: Find the child Component and let the `countChange` Output emit a value.
2. *Assert*: Check that `console.log` has been called.

From the parent's viewpoint, `countChange` is simply an event. Shallow rendering means there is no `CounterComponent` instance and no `EventEmitter` named `countChange`. Angular only sees an element, `app-counter`, with an event handler, `(countChange)="handleCountChange($event)".`

SIMULATE OUTPUT

In this setup, we can simulate the Output using the known `triggerEventHandler` method.

```
it('listens for count changes', () => {  
  /* ... */  
  const counter = findComponent(fixture, 'app-counter');  
  const count = 5;  
  counter.triggerEventHandler('countChange', 5);  
  /* ... */  
});
```

The spec finds the `app-counter` element and triggers the `countChange` event handler.

The second `triggerEventHandler` parameter, `5`, is not an event object as we know from DOM events like `click`. It is a value that the Output would emit. The `countChange` Output has the type `EventEmitter<number>`, so we use the fixed number `5` for testing purposes.

OUTPUT EFFECT

Under the hood, `triggerEventHandler` runs `handleCountChange($event)` with `$event` being `5`. `handleCountChange` calls `console.log`. This is the observable effect we need to test.

How do we verify that `console.log` has been called? We can [spy on existing methods](#) with Jasmine's `spyOn`.

```
spyOn(console, 'log');
```

This overwrites `console.log` with a spy for the duration of the test run. We need to set up the spy in the *Arrange* phase, at the beginning of our spec.

```
it('listens for count changes', () => {
  spyOn(console, 'log');
  const counter = findComponent(fixture, 'app-counter');
  const count = 5;
  counter.triggerEventHandler('countChange', count);
  /* ... */
});
```

In the *Assert* phase, we expect that the spy has been called with a certain text and the number the Output has emitted.

```
it('listens for count changes', () => {
  spyOn(console, 'log');
  const counter = findComponent(fixture, 'app-counter');
  const count = 5;
  counter.triggerEventHandler('countChange', count);
  expect(console.log).toHaveBeenCalledWith(
    'countChange event from CounterComponent',
    count,
  );
});
```

So much for testing the `CounterComponent` child. The `HomeComponent` also renders a `ServiceCounterComponent` and an `NgRxCounterComponent` like this:

```
<app-service-counter></app-service-counter>
<!-- ... -->
<app-ngrx-counter></app-ngrx-counter>
```

CHILD PRESENCE

Since they do not have Inputs or Outputs, we merely need to test whether they are mentioned in the template. We add two additional specs that check the presence of these `app-service-counter` and `app-ngrx-counter` elements, respectively.

```
it('renders a service counter', () => {
  const serviceCounter = findComponent(fixture, 'app-service-counter');
  expect(serviceCounter).toBeTruthy();
});
```

```
});
```

```
it('renders a NgRx counter', () => {  
  const ngrxCounter = findComponent(fixture, 'app-ngrx-counter');  
  expect(ngrxCounter).toBeTruthy();  
});
```

This is it! We have written a unit test with shallow rendering that proves that `HomeComponent` correctly embeds several child Components.

Note that this is one possible testing method. As always, it has pros and cons. Compared with a full integration test, there is little setup. The specs can use Angular's `DebugElement` abstraction to test presence as well as Inputs and Outputs.

UNIT TEST CONFIDENCE

However, the unit test gives little confidence that `HomeComponent` works in production. We have instructed Angular to ignore the elements `app-counter`, `app-service-counter` and `app-ngrx-counter`.

What if `HomeComponent` uses a wrong element name and the test copies that error? The test would pass incorrectly. We need to render the involved Components together to spot the error.

 [HomeComponent: implementation and test code](#)

 [Element spec helpers: full code](#)

Faking a child Component

There is a middle ground between a naive unit test and an integration test. Instead of working with empty custom elements, we can render *fake* child Components.

A fake Component has the same selector, Inputs and Outputs, but has no dependencies and does not have to render anything. When testing a Component with children, we substitute the children for fake Components.

Let us reduce the [CounterComponent](#) to an empty shell that offers the same public API:

```
@Component({
  selector: 'app-counter',
  template: '',
})
class FakeCounterComponent implements Partial<CounterComponent> {
  @Input()
  public startCount = 0;

  @Output()
  public countChange = new EventEmitter<number>();
}
```

This fake Component lacks a template and any logic, but has the same selector, Input and Output.

SAME PUBLIC API

Remember the [rules for faking dependencies](#)? We need to make sure the fake resembles the original. `FakeCounterComponent implements Partial<CounterComponent>` requires the class to implement a subset of `CounterComponent`. TypeScript enforces that the given properties and methods have the same types as in the original class.

DECLARE FAKE COMPONENT

In our test suite, we place the `FakeCounterComponent` before the `describe` block. The next step is to add the Component to the testing Module:

```
TestBed.configureTestingModule({
  declarations: [HomeComponent, FakeCounterComponent],
  schemas: [NO_ERRORS_SCHEMA],
}).compileComponents();
```

When Angular encounters an `app-counter` element, it instantiates and mounts a `FakeCounterComponent`. The element stays empty since the fake template is empty as well. The `startCount` Input property is set and the parent `HomeComponent` subscribes to the `countChange` Output.

We need to adapt the test suite now that child Component are rendered. Instead of searching for an `app-counter` element and inspecting its properties, we explicitly search for a `FakeCounterComponent` instance.

So far, we have used `DebugElement`'s `query` method to find nested elements. For example:

```
const element = fixture.debugElement.query(By.css('...'));
```

Our helpers `findEl` and `findComponent` are using this pattern as well.

FIND BY DIRECTIVE

Now we want to find a nested Component. We can use `query` together with the `By.directive` predicate function:

```
const counterEl = fixture.debugElement.query(  
  By.directive(FakeCounterComponent)  
);
```

`By.directive` finds all kinds of Directives. A Component is a kind of Directive.

`query` returns a `DebugElement` or `null` in case no match was found. As we have learned, a `DebugElement` always wraps a native DOM element. When we query for `FakeCounterComponent`, we get a `DebugElement` that wraps the `app-counter` element – just as `By.css('app-counter')` would return.

CHILD COMPONENT INSTANCE

The difference is that we can now access the rendered `FakeCounterComponent` via the `componentInstance` property:

```
const counterEl = fixture.debugElement.query(
  By.directive(FakeCounterComponent)
);
const counter: CounterComponent = counterEl.componentInstance;
```

Angular does not know the type of the Component, `componentInstance` has the type `any`. So we add an explicit type annotation.

CHILD PRESENCE

Having access to the child Component instance, we can make expectations against it. First of all, we verify the presence.

```
it('renders an independent counter', () => {
  const counterEl = fixture.debugElement.query(
    By.directive(FakeCounterComponent)
  );
  const counter: CounterComponent = counterEl.componentInstance;

  expect(counter).toBeTruthy();
});
```

This is a smoke test that fails early if no instance of `FakeCounterComponent` was found. `query` would return `null` and `counterEl.componentInstance` would fail with a `TypeError: counterEl is null`.

CHECK INPUTS

The second spec checks the Input. An Input is a property of the Component instance, so `counter.startCount` gives us the value

of the `startCount` Input.

```
it('passes a start count', () => {
  const counterEl = fixture.debugElement.query(
    By.directive(FakeCounterComponent)
  );
  const counter: CounterComponent = counterEl.componentInstance;

  expect(counter.startCount).toBe(5);
});
```

The third spec checks the Output handling: If the counter emits a value, the `HomeComponent` passes it to `console.log`.

EMIT OUTPUT

As mentioned earlier, an Output is an `EventEmitter` property on the Component instance. Previously, we have simulated an Output event using the `triggerEventHandler` abstraction. Now we can access the Output directly and call its `emit` method, just like the code in the child Component does.

```
it('listens for count changes', () => {
  const counterEl = fixture.debugElement.query(
    By.directive(FakeCounterComponent)
  );
  const counter: CounterComponent = counterEl.componentInstance;

  spyOn(console, 'log');
  const count = 5;
  counter.countChange.emit(5);
  expect(console.log).toHaveBeenCalledWith(
```

```

        'countChange event from CounterComponent',
        count,
    );
});

```

We are done! Here is the `HomeComponent` test suite that vets the `CounterComponent` child. To minimize repetition and noise, we move the query part into the `beforeEach` block.

```

@Component({
  selector: 'app-counter',
  template: '',
})
class FakeCounterComponent implements Partial<CounterComponent> {
  @Input()
  public startCount = 0;

  @Output()
  public countChange = new EventEmitter<number>();
}

describe('HomeComponent (faking a child Component)', () => {
  let fixture: ComponentFixture<HomeComponent>;
  let component: HomeComponent;
  let counter: FakeCounterComponent;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [HomeComponent, FakeCounterComponent],
      schemas: [NO_ERRORS_SCHEMA],
    }).compileComponents();

    fixture = TestBed.createComponent(HomeComponent);
  });
});

```

```

    component = fixture.componentInstance;
    fixture.detectChanges();

    const counterEl = fixture.debugElement.query(
      By.directive(FakeCounterComponent)
    );
    counter = counterEl.componentInstance;
  });

  it('renders an independent counter', () => {
    expect(counter).toBeTruthy();
  });

  it('passes a start count', () => {
    expect(counter.startCount).toBe(5);
  });

  it('listens for count changes', () => {
    spyOn(console, 'log');
    const count = 5;
    counter.countChange.emit(count);
    expect(console.log).toHaveBeenCalledWith(
      'countChange event from CounterComponent',
      count,
    );
  });
});

```

Let us recap what we have gained with this type of testing the `HomeComponent`.

We have replaced a Component dependency with a fake that behaves the same, as far as `HomeComponent` is concerned. The

fake child is rendered, but the template may be empty.

The original child Component, `CounterComponent`, is imported only to create the derived fake Component. Our test remains a fast and short unit test.

ADVANTAGES

Instead of searching for an element named `app-counter`, we search for a Component instance. This is more robust. The presence of the host element is a good indicator, but it is more relevant that a Component has been rendered into this element.

Working with the Component instance is more intuitive than working with the `DebugElement` abstraction. We can read Component properties to learn about Inputs and Outputs. Basic JavaScript and Angular knowledge suffices to write specs against such an instance.

MANUAL FAKING DRAWBACKS

Our simple approach to faking a child Component has its flaws. We have created the fake manually. This is tedious and time-consuming, but also risky. The fake is only partly tied to the original.

For example, if the original changes its selector `app-counter`, the test should fail and remind us to adapt the template. Instead, it passes incorrectly since we did not inherit the Component

metadata, { selector: 'app-counter', ... }, but duplicated it in the test.

We are going to address these shortcomings in the next chapter.

🔗 [HomeComponent spec that fakes a child Component](#)

Faking a child Component with ng-mocks

We have manually created a Component fake. This is an important exercise to understand how faking Components works, but it does not produce a robust, versatile fake. In this guide, we cannot discuss all necessary bits and pieces of creating airtight fake Components.

Instead, we will use a mature solution: [ng-mocks](#) is a feature-rich library for testing Components with fake dependencies.

(Remember, this guide uses the umbrella term “fake” while other articles and tools use terms like “mock” or “stub”.)

CREATE FAKE FROM ORIGINAL

Among other things, ng-mocks helps creating fake Components to substitute children. The `MockComponent` function expects the

original Component and returns a fake that resembles the original.

Instead of creating a `FakeCounterComponent`, we call `MockComponent(CounterComponent)` and add the fake to the testing Module.

```
import { MockComponent } from 'ng-mocks';

beforeEach(async () => {
  await TestBed.configureTestingModule({
    declarations: [HomeComponent, MockComponent(CounterComponent)],
    schemas: [NO_ERRORS_SCHEMA],
  }).compileComponents();
});
```

We can then query the rendered DOM for an instance of `CounterComponent`. The found instance is in fact a fake created by ng-mocks. Still, we can declare the type `CounterComponent`.

```
describe('HomeComponent with ng-mocks', () => {
  let fixture: ComponentFixture<HomeComponent>;
  let component: HomeComponent;
  // Original type!
  let counter: CounterComponent;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [HomeComponent, MockComponent(CounterComponent)],
      schemas: [NO_ERRORS_SCHEMA],
    }).compileComponents();
```

```

    fixture = TestBed.createComponent(HomeComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();

    const counterEl = fixture.debugElement.query(
      // Original class!
      By.directive(CounterComponent)
    );
    counter = counterEl.componentInstance;
  });

  /* ... */
});

```

From a TypeScript viewpoint, the fake conforms to the `CounterComponent` type. TypeScript uses a structural type system that checks if all type requirements are met.

TYPE EQUIVALENCE

Every proposition that holds true for a `CounterComponent` holds true for the fake as well. The fake has all properties and methods that the original has. That is why we can safely replace the original with the fake and treat the fake the same in our test.

The full code:

```

describe('HomeComponent with ng-mocks', () => {
  let fixture: ComponentFixture<HomeComponent>;
  let component: HomeComponent;
  let counter: CounterComponent;

```

```
beforeEach(async () => {
  await TestBed.configureTestingModule({
    declarations: [HomeComponent, Mock(CounterComponent)],
    schemas: [NO_ERRORS_SCHEMA],
  }).compileComponents();

  fixture = TestBed.createComponent(HomeComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();

  const counterEl = fixture.debugElement.query(
    By.directive(CounterComponent)
  );
  counter = counterEl.componentInstance;
});

it('renders an independent counter', () => {
  expect(counter).toBeTruthy();
});

it('passes a start count', () => {
  expect(counter.startCount).toBe(5);
});

it('listens for count changes', () => {
  spyOn(console, 'log');
  const count = 5;
  counter.countChange.emit(count);
  expect(console.log).toHaveBeenCalledWith(
    'countChange event from CounterComponent',
    count,
  );
});
});
```


We have eliminated the manual `FakeCounterComponent`. We are using `MockComponent(CounterComponent)` to create the fake and the original class `CounterComponent`. The specs itself did not change.

This was only a glimpse of ng-mocks. The library not only helps with nested Components, but provides high-level helpers for setting up the Angular test environment. ng-mocks replaces the conventional setup with `TestBed.configureTestingModule` and helps faking Modules, Components, Directives, Pipes and Services.

 [HomeComponent spec with ng-mocks](#)

 [ng-mocks](#)

Testing Components depending on Services

LEARNING OBJECTIVES

- Choosing between a unit or an integration test for Components that talk to Services
- Creating fake Services to test the Component in isolation
- Verifying that the Component correctly interacts with the Service
- Understanding different approaches for faking a Service dependency

We have successfully tested the independent [CounterComponent](#) as well as the container [HomeComponent](#). The next Component on our list is the [ServiceCounterComponent](#).

As the name suggests, this Component depends on the [CounterService](#). The counter state is not stored in the Component itself, but in the central Service.

SHARED CENTRAL STATE

Angular's dependency injection maintains only one app-wide instance of the Service, a so-called singleton. Therefore, multiple instances of [ServiceCounterComponent](#) share the same counter state. If the user increments the count with one instance, the count also changes in the other instance.

Again, there are two fundamental ways to test the Component:

- A unit test that replaces the `CounterService` dependency with a fake.
- An integration test that includes a real `CounterService`.

This guide will demonstrate both. For your Components, you need to make a decision on an individual basis. These questions may guide you: Which type of test is more beneficial, more meaningful? Which test is easier to set up and maintain in the long run?

Service dependency integration test

For the `ServiceCounterComponent`, the integration test is much easier to set up than the unit test. The trivial `CounterService` has little logic and no further dependencies. It does not have side effects we need to suppress in the testing environment, like HTTP requests. It only changes its internal state.

The integration test looks almost identical to the `CounterComponent` test we have already written.

```
describe('ServiceCounterComponent: integration test', () => {  
  let component: ServiceCounterComponent;  
  let fixture: ComponentFixture<ServiceCounterComponent>;  
  
  beforeEach(async () => {
```

```
    await TestBed.configureTestingModule({
      declarations: [ServiceCounterComponent],
      providers: [CounterService],
    }).compileComponents();

    fixture = TestBed.createComponent(ServiceCounterComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('shows the start count', () => {
    expectText(fixture, 'count', '0');
  });

  it('increments the count', () => {
    click(fixture, 'increment-button');
    fixture.detectChanges();
    expectText(fixture, 'count', '1');
  });

  it('decrements the count', () => {
    click(fixture, 'decrement-button');
    fixture.detectChanges();
    expectText(fixture, 'count', '-1');
  });

  it('resets the count', () => {
    const newCount = 456;
    setFieldValue(fixture, 'reset-input', String(newCount));
    click(fixture, 'reset-button');
    fixture.detectChanges();
    expectText(fixture, 'count', String(newCount));
  });
});
```

Compared to the `CounterComponent` test, there is nothing new here except for one line:

```
providers: [CounterService],
```

PROVIDE SERVICE

This line adds the `CounterService` to the testing Module. Angular creates an instance of the Service and injects it into the Component under test. The test is shorter because the `ServiceCounterComponent` does not have Inputs or Outputs to test.

As the `CounterService` always starts with the count `0`, the test needs to take that for granted. Neither the Component nor the Service allow a different start count.

INTERACTION WITH SERVICE

The integration test does not examine the Component's inner workings. It only provides the Service but does not check how the Component and the Service interact. The Component might not talk to the Service at all.

If we want an integration test to verify that the Component stores the count in the Service, we would need a test with two `ServiceCounterComponents`: When increasing the count using one Component, the displayed count in the other should change accordingly.

Faking Service dependencies

Let us move on to the **unit test** for the [ServiceCounterComponent](#). To tackle this challenge, we need to learn the art of faking Service dependencies.

There are several practical approaches with pros and cons. We have discussed two main [requirements on fake dependencies](#):

1. Equivalence of fake and original: The fake must have a type derived from the original.
2. Effective faking: the original stays untouched.

RECOMMENDED FAKING APPROACH

This guide will present one solution that implements these requirements. Note that other solutions might meet these requirements as well.

The dependency we need to fake, [CounterService](#), is a simple class annotated with [@Injectable\(\)](#). This is the outer shape of [CounterService](#):

```
class CounterService {  
    public getCount(): Observable<number> { /* ... */ }
```

```
public increment(): void { /* ... */ }
public decrement(): void { /* ... */ }
public reset(newCount: number): void { /* ... */ }
private notify(): void { /* ... */ }
}
```

We need to build a fake that meets the mentioned needs.

FAKE INSTANCE

The simplest way to create a fake is an object literal `{...}` with methods:

```
const currentCount = 123;
const fakeCounterService = {
  getCount() {
    return of(currentCount);
  },
  increment() {},
  decrement() {},
  reset() {},
};
```

`getCount` returns a fixed value from a constant named `currentCount`. We will use the constant later to check whether the Component uses the value correctly.

This fake is far from perfect, but already a viable replacement for a `CounterService` instance. It walks like the original and talks like the original. The methods are empty or return fixed data.

TYPE EQUIVALENCE

The fake implementation above happens to have the same shape as the original. As discussed, it is of utter importance that the fake remains up to date with the original.

The equivalence is not yet enforced by TypeScript. We want TypeScript to check whether the fake properly replicates the original. The first attempt would be to add a type declaration:

ERRONEOUS CODE

```
// Error!
const fakeCounterService: CounterService = {
  getCount() {
    return of(currentCount);
  },
  increment() {},
  decrement() {},
  reset() {},
};
```

Unfortunately, this does not work. TypeScript complains that private methods and properties are missing:

```
Type '{ getCount(): Observable<number>; increment(): void; decrement(): void; reset(): void; }' is missing the following properties from type 'CounterService': count, subject, notify
```

That is correct. But we cannot add private members to an object literal, nor should we.

PICK PUBLIC MEMBERS

Luckily, we can use a TypeScript trick to fix this problem. Using [Pick](#) and [keyof](#), we create a derived type that only contains the public members:

```
const fakeCounterService:
  Pick<CounterService, keyof CounterService> = {
  getCount() {
    return of(currentCount);
  },
  increment() {},
  decrement() {},
  reset() {},
};
```

KEEP FAKE IN SYNC

When the `CounterService` changes its public API, the dependent `ServiceCounterComponent` needs to be adapted. Likewise, the `fakeCounterService` needs to reflect the change. The type declaration reminds you to update the fake. It prevents the fake to get out of sync with the original.

FAKE WHAT IS NECESSARY

`ServiceCounterComponent` calls all existing public `CounterService` methods, so we have added them to the fake.

If the code under test does not use the full API, the fake does not need to replicate the full API either. Only declare those methods

and properties the code under test actually uses.

For example, if the code under test only calls `getCount`, just provide this method. Make sure to add a type declaration that picks the method from the original type:

```
const fakeCounterService: Pick<CounterService, 'getCount'> = {
  getCount() {
    return of(currentCount);
  },
};
```

`Pick` and other [mapped types](#) help to bind the fake to the original type in a way that TypeScript can check the equivalence.

SPY ON METHODS

A plain object with methods is an easy way to create a fake instance. The spec needs to verify that the methods have been called with the right parameters.

[Jasmine spies](#) are suitable for this job. A first approach fills the fake with standalone spies:

```
const fakeCounterService:
  Pick<CounterService, keyof CounterService> = {
  getCount:
    jasmine.createSpy('getCount').and.returnValue(of(currentCount)),
  increment: jasmine.createSpy('increment'),
  decrement: jasmine.createSpy('decrement'),
  reset: jasmine.createSpy('reset'),
};
```

CREATESPYOBJ

This is fine, but overly verbose. Jasmine provides a handy helper function for creating an object with multiple spy methods, `createSpyObj`. It expects a descriptive name and an object with method names and return values:

```
const fakeCounterService = jasmine.createSpyObj<CounterService>(
  'CounterService',
  {
    getCount: of(currentCount),
    increment: undefined,
    decrement: undefined,
    reset: undefined,
  }
);
```

The code above creates an object with four methods, all of them being spies. They return the given values: `getCount` returns an `Observable<number>`. The other methods return `undefined`.

TYPE EQUIVALENCE

`createSpyObj` accepts a [TypeScript type variable](#) to declare the type of the created object. We pass `CounterService` between angle brackets so TypeScript checks that the fake matches the original.

Let us put our fake to work. In the *Arrange* phase, the fake is created and injected into the testing Module.

```
describe('ServiceCounterComponent: unit test', () => {
  const currentCount = 123;

  let component: ServiceCounterComponent;
  let fixture: ComponentFixture<ServiceCounterComponent>;
  // Declare shared variable
  let fakeCounterService: CounterService;

  beforeEach(async () => {
    // Create fake
    fakeCounterService = jasmine.createSpyObj<CounterService>(
      'CounterService',
      {
        getCount: of(currentCount),
        increment: undefined,
        decrement: undefined,
        reset: undefined,
      }
    );

    await TestBed.configureTestingModule({
      declarations: [ServiceCounterComponent],
      // Use fake instead of original
      providers: [
        { provide: CounterService, useValue: fakeCounterService }
      ],
    }).compileComponents();

    fixture = TestBed.createComponent(ServiceCounterComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
```

```
    /* ... */  
  });
```

There is a new pattern in the `providers` sections of the testing Module:

```
providers: [  
  { provide: CounterService, useValue: fakeCounterService }  
]
```

PROVIDE FAKE INSTEAD

This is the crucial moment where we tell Angular: For the `CounterService` dependency, use the value `fakeCounterService` instead. This is how we replace the original with a fake.

Normally, Angular instantiates and injects a `CounterService` instance whenever a Component, Service, etc. asks for the `CounterService`. By using `{ provide: ..., useValue: ... }`, we skip the instantiation and directly provide the value to inject.

The *Arrange* phase is complete now, let us write the actual specs.

The *Act* phase is the same as in the other counter Component tests: We click on buttons and fill out form fields.

VERIFY SPIES

In the *Assert* phase, we need to verify that the Service methods have been called. Thanks to `jasmine.createSpyObj`, all methods of `fakeCounterService` are spies. We use `expect` together with

an appropriate matcher like `toHaveBeenCalled`, `toHaveBeenCalled`, `toHaveBeenCalled`, etc.

```
expect(fakeCounterService.getCount).toHaveBeenCalled();
```

Applied to all specs, the test suite looks like this:

```
describe('ServiceCounterComponent: unit test', () => {
  const currentCount = 123;

  let component: ServiceCounterComponent;
  let fixture: ComponentFixture<ServiceCounterComponent>;
  // Declare shared variable
  let fakeCounterService: CounterService;

  beforeEach(async () => {
    // Create fake
    fakeCounterService = jasmine.createSpyObj<CounterService>(
      'CounterService',
      {
        getCount: of(currentCount),
        increment: undefined,
        decrement: undefined,
        reset: undefined,
      }
    );

    await TestBed.configureTestingModule({
      declarations: [ServiceCounterComponent],
      // Use fake instead of original
      providers: [
        { provide: CounterService, useValue: fakeCounterService }
      ],
    }).compileComponents();
```

```
    fixture = TestBed.createComponent(ServiceCounterComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('shows the count', () => {
    expectText(fixture, 'count', String(currentCount));
    expect(fakeCounterService.getCount).toHaveBeenCalled();
  });

  it('increments the count', () => {
    click(fixture, 'increment-button');
    expect(fakeCounterService.increment).toHaveBeenCalled();
  });

  it('decrements the count', () => {
    click(fixture, 'decrement-button');
    expect(fakeCounterService.decrement).toHaveBeenCalled();
  });

  it('resets the count', () => {
    const newCount = 456;
    setFieldValue(fixture, 'reset-input', String(newCount));
    click(fixture, 'reset-button');
    expect(fakeCounterService.reset).toHaveBeenCalledWith(newCount);
  });
});
```

 [ServiceCounterComponent: implementation and test code](#)

 [Angular guide: Dependency providers](#)

Fake Service with minimal logic

The specs above check whether user interaction calls the Service methods. They do not check whether the Component re-renders the new count after having called the Service.

`ServiceCounter`'s `getCount` method returns an `Observable<number>` and pushes a new value through the Observable whenever the count changes. The spec `it('shows the count', ...)` has proven that the Component obtained the count from the Service and renders it.

In addition, we will check that the Component updates when new values are pushed. This is not strictly necessary in our simple `ServiceCounterComponent` and `CounterService` example. But it is important in more complex interactions between a Component and a Service.

COMPONENT UPDATE

The fake `getCount` method returns `of(currentCount)`, an Observable with the fixed value 123. The Observable completes immediately and never pushes another value. We need to change that behavior in order to demonstrate the Component update.

The fake `CounterService`, devoid of logic so far, needs to gain some logic. `getCount` needs to return an Observable that emits

new values when `increment`, `decrement` and `reset` are called.

BEHAVIORSUBJECT

Instead of a fixed `Observable`, we use a `BehaviorSubject`, just like in the original `CounterService` implementation. The `BehaviorSubject` has a `next` method for pushing new values.

We declare a variable `fakeCount$` in the scope of the test suite and assign a `BehaviorSubject` in the first `beforeEach` block:

```
describe('ServiceCounterComponent: unit test with minimal Service logic', ()
=> {
  /* ... */
  let fakeCount$: BehaviorSubject<number>;

  beforeEach(async () => {
    fakeCount$ = new BehaviorSubject(0);
    /* ... */
  });

  /* ... */
});
```

Then we change the `fakeCounterService` so the methods push new values through `fakeCount$`.

```
const newCount = 123;
/* ... */
fakeCounterService = {
  getCount(): Observable<number> {
    return fakeCount$;
```

```

    },
    increment(): void {
        fakeCount$.next(1);
    },
    decrement(): void {
        fakeCount$.next(-1);
    },
    reset(): void {
        fakeCount$.next(Number(newCount));
    },
};

```

The fake above is an object with plain methods. We are not using `createSpyObj` any longer because it does not allow fake method implementations.

SPY ON METHODS

We have lost the Jasmine spies and need to bring them back. There are several ways to wrap the methods in spies. For simplicity, we install spies on all methods using `spyOn`:

```

spyOn(fakeCounterService, 'getCount').and.callThrough();
spyOn(fakeCounterService, 'increment').and.callThrough();
spyOn(fakeCounterService, 'decrement').and.callThrough();
spyOn(fakeCounterService, 'reset').and.callThrough();

```

Remember to add `.and.callThrough()` so the underlying fake methods are called.

Now our fake Service sends new counts to the Component. We can reintroduce the checks for the Component output:

```
fixture.detectChanges();  
expectText(fixture, 'count', '...');
```

Assembling all parts, the full `ServiceCounterComponent` unit test:

```
describe('ServiceCounterComponent: unit test with minimal Service logic', ()  
=> {  
  const newCount = 456;  
  
  let component: ServiceCounterComponent;  
  let fixture: ComponentFixture<ServiceCounterComponent>;  
  
  let fakeCount$: BehaviorSubject<number>;  
  let fakeCounterService: Pick<CounterService, keyof CounterService>;  
  
  beforeEach(async () => {  
    fakeCount$ = new BehaviorSubject(0);  
  
    fakeCounterService = {  
      getCount(): Observable<number> {  
        return fakeCount$;  
      },  
      increment(): void {  
        fakeCount$.next(1);  
      },  
      decrement(): void {  
        fakeCount$.next(-1);  
      },  
      reset(): void {  
        fakeCount$.next(Number(newCount));  
      },  
    };  
  
    spyOn(fakeCounterService, 'getCount').and.callThrough();  
    spyOn(fakeCounterService, 'increment').and.callThrough();  
  });  
}
```

```
spyOn(fakeCounterService, 'decrement').and.callThrough();
spyOn(fakeCounterService, 'reset').and.callThrough();

await TestBed.configureTestingModule({
  declarations: [ServiceCounterComponent],
  providers: [
    { provide: CounterService, useValue: fakeCounterService }
  ],
}).compileComponents();

fixture = TestBed.createComponent(ServiceCounterComponent);
component = fixture.componentInstance;
fixture.detectChanges();
});

it('shows the start count', () => {
  expectText(fixture, 'count', '0');
  expect(fakeCounterService.getCount).toHaveBeenCalled();
});

it('increments the count', () => {
  click(fixture, 'increment-button');
  fakeCount$.next(1);
  fixture.detectChanges();

  expectText(fixture, 'count', '1');
  expect(fakeCounterService.increment).toHaveBeenCalled();
});

it('decrements the count', () => {
  click(fixture, 'decrement-button');
  fakeCount$.next(-1);
  fixture.detectChanges();
```

```
    expectText(fixture, 'count', '-1');
    expect(fakeCounterService.decrement).toHaveBeenCalled();
  });

  it('resets the count', () => {
    setFieldValue(fixture, 'reset-input', newCount);
    click(fixture, 'reset-button');
    fixture.detectChanges();

    expectText(fixture, 'count', newCount);
    expect(fakeCounterService.reset).toHaveBeenCalledWith(newCount);
  });
});
```

Again, this example is intentionally verbose. The fake re-implements a large part of the original logic. This is because the original `CounterService` has little logic itself.

In reality, Services are more complex and Components process the data they receive from the Services. Then, the effort of faking essential logic is worthwhile.

🔗 [ServiceCounterComponent: unit test](#)

Faking Services: Summary

Creating fake Service dependencies and verifying their usage is one of the most challenging problems when testing Angular applications. This guide can only catch a glimpse on the subject.

TESTABLE SERVICES

Faking Services requires effort and steady practice. The more unit tests you write, the more experience you gain. More importantly, the practice teaches you to write *simple Services that are easy to fake*: Services with a clear API and an obvious purpose.

Unfortunately, there are no best practices when it comes to faking Services. You will find plenty of approaches online that have their strengths and weaknesses. The associated unit tests have different degrees of accuracy and completeness.

Arguing about the “right” way of faking a Service is pointless. You need to decide on a faking method that suits the Service on a case-by-case basis.

GUIDELINES

There are two guidelines that may help you:

1. Is the test valuable? Does it cover the important interaction between Component and Service? Decide whether to test the interaction superficially or in-depth.
2. Whichever approach you choose, make sure to meet the [basic requirements](#):
 1. Equivalence of fake and original: The fake must have a type derived from the original.

2. Effective faking: the original stays untouched.

Testing complex forms

LEARNING OBJECTIVES

- Filling out and submitting forms in a Component test
- Testing synchronous and asynchronous field validation and error messages
- Testing dynamic form logic
- Using tools to ensure that forms are accessible to everyone

Forms are the powerhouses of large web applications. Especially enterprise applications revolve around entering and editing data via forms. Therefore, implementing complex forms is a vital feature of the Angular framework.

We have already learned how to [fill out form fields](#) when testing the counter Component. In doing so, we developed the `setFieldValue` testing helper.

The simple forms we have dealt with served the purpose of entering one value. We have tested them by filling out the field and submitting the form. Now we will look at a more complex example.

SIGN-UP FORM

We are introducing and testing a **sign-up form** for a fictional online service.

 [Sign-up form: Source code](#)

 [Sign-up form: Run the app](#)

The sign-up form features are:

- Different types of input fields: text, radio buttons, checkboxes, select boxes
- Field validation with synchronous and asynchronous validators
- Accessible form structure, field labels and error messages
- Dynamic relations between fields

The form consists of four sections:

1. The plan selection: "Personal", "Business" or "Education & Non-profit"
2. The login credentials: username, email and password
3. The billing address
4. Terms of Services and submit button

IMPRACTICAL

Please note that this form is for demonstration purposes only. While it follows best practices regarding validation and accessibility, it is not practical from a design and user experience perspective. Among other things, it is way too complex to get new users onboard.

CLIENT & SERVER

In contrast to the other example repositories, this one is split into a [client](#) and a [server](#) directory:

- The [client directory](#) contains a standard Angular app created with Angular CLI.
- The [server directory](#) contains a simple Node.js service that simulates the user management and account creation.

Again, the Node.js service is for demonstration purposes only. The service holds the created user accounts in memory and discards them when stopped. Please do not use it in production.

With 12 form controls, the sign-up form is not particularly large. But there are subtle details we are going to explore.

Sign-up form Component

The form logic lies in the [SignupFormComponent](#). The Component depends on the [SignupService](#) for communicating with the back-end service.

You might remember that there are two fundamental approaches to forms in Angular: *Template-driven Forms* and *Reactive Forms*.

While both approaches look quite different in practice, they are based on the same underlying concepts: Form groups ([FormGroup](#) objects) and form controls ([FormControl](#) objects).

REACTIVE FORM

The [SignupFormComponent](#) is a **Reactive Form** that explicitly creates the groups and controls in the Component class. This way, it is easier to specify custom validators and to set up dynamic field relations.

As with other Angular core concepts, this guide assumes you have a basic understanding of Reactive Forms. Please refer to the [official guide on Reactive Forms](#) to brush up your knowledge.

The important bits of the [SignupFormComponent](#) class are:

```
@Component({
  selector: 'app-signup-form',
  templateUrl: './signup-form.component.html',
  styleUrls: ['./signup-form.component.scss'],
})
export class SignupFormComponent {
  /* ... */
  public form = this.formBuilder.group({
    plan: ['personal', required],
    username: [
      null,
      [required, pattern('[a-zA-Z0-9.]+'), maxLength(50)],
      (control: AbstractControl) =>
        this.validateUsername(control.value),
    ],
  });
}
```

```

    ],
    email: [
        null,
        [required, email, maxLength(100)],
        (control: AbstractControl) =>
            this.validateEmail(control.value),
    ],
    password: [
        null,
        required,
        () => this.validatePassword()
    ],
    tos: [null, requiredTrue],
    address: this.formBuilder.group({
        name: [null, required],
        addressLine1: [null],
        addressLine2: [null, required],
        city: [null, required],
        postcode: [null, required],
        region: [null],
        country: [null, required],
    }),
});
/* ... */
constructor(
    private signupService: SignupService,
    private formBuilder: FormBuilder) {
    /* ... */
}
/* ... */
}

```

FORM GROUPS AND CONTROLS

Using Angular's [FormBuilder](#), we create the `form` property, the topmost form group. Inside, there is another form group for the address-related fields.

The form controls are declared with their initial values and their validators. For example, the password control:

```
password: [  
  // The initial value (null means empty)  
  null,  
  // The synchronous validator  
  required,  
  // The asynchronous validator  
  () => this.validatePassword()  
],
```

The [SignupFormComponent template](#) uses the `formGroup`, `formGroupName` and `formControlName` directives to associate elements with a form group or control, respectively.

The stripped-down form structure with only one control looks like this:

```
<form [formGroup]="form">  
  <fieldset formGroupName="address">  
    <label>  
      Full name  
      <input type="text" formControlName="name" />  
    </label>  
  </fieldset>  
</form>
```

FORM SUBMISSION

When the form is filled out correctly and all validations pass, the user is able to submit to the form. It produces an object described by the [SignupData interface](#):

```
export interface SignupData {  
  plan: Plan;  
  username: string;  
  email: string;  
  password: string;  
  tos: true;  
  address: {  
    name: string;  
    addressLine1?: string;  
    addressLine2: string;  
    city: string;  
    postcode: string;  
    region?: string;  
    country: string;  
  };  
}
```

Plan is a union of strings:

```
export type Plan = 'personal' | 'business' | 'non-profit';
```

The **SignupService**'s **signup** method takes the **SignupData** and sends it to the server. For security reasons, the server validates the data again. But we will focus on the front-end in this guide.

 [SignupFormComponent: full code](#)

 [Angular documentation: Reactive forms](#)

Form validation and errors

SYNC VALIDATORS

Several form controls have synchronous validators. `required`, `email`, `maxLength`, `pattern` etc. are built-in, synchronous validators provided by Angular:

```
import { Validators } from '@angular/forms';

const {
  email, maxLength, pattern, required, requiredTrue
} = Validators;
```

These validators take the control value, a string most of the time, and return a `ValidationErrors` object with potential error messages. The validation happens synchronously on the client.

ASYNC VALIDATORS

For the username, the email and the password, there are custom asynchronous validators. They check whether the username and email are available and whether the password is strong enough.

The asynchronous validators use the `SignupService` to talk to the back-end service. These HTTP requests turn the validation asynchronous.

ERROR RENDERING

When a validator returns any errors, corresponding messages are shown below the form control. This repetitive task is outsourced to another Component.

INVALID && (TOUCHED || DIRTY)

The [ControlErrorsComponent](#) displays the errors when the form control is *invalid* and either *touched* or *dirty*.

- *Touched* means the user has focussed the control but it has lost the focus again (the `blur` event fired).
- *Dirty* means the user has changed the value.

For example, for the `name` control, the interaction between the `input` element and the [ControlErrorsComponent](#) looks like this:

```
<label>
  Full name
  <input
    type="text"
    formControlName="name"
    aria-required="true"
    appErrorMessage="name-errors"
  />
</label>
<!-- ... -->
<app-control-errors controlName="name" id="name-errors">
  <ng-template let-errors>
    <ng-container *ngIf="errors.required">
      Name must be given.
    </ng-container>
  </ng-template>
</app-control-errors>
```



```
</ng-template>
</app-control-errors>
```

ARIA ATTRIBUTES

The `appErrorMessage` attribute activates the [`ErrorMessageDirective`](#). When the form control is invalid and either touched or dirty, the Directive adds `aria-invalid` and `aria-errormessage` attributes.

`aria-invalid` marks the control as invalid for assistive technologies like screen readers. `aria-errormessage` points to another element that contains the error messages.

CONNECT CONTROL WITH ERRORS

In case of an error, the Directive sets `aria-errormessage` to the id of the corresponding `app-control-errors` element. In the example above, the id is `name-errors`. This way, a screen reader user finds the associated error messages quickly.

The control-specific error messages are still located in `signup-form.component.html`. They are passed to `ControlErrorsComponent` as an `ng-template`. The `ControlErrorsComponent` renders the template dynamically, passing the `errors` object as a variable:

```
<ng-template let-errors>
  <ng-container *ngIf="errors.required">
    Name must be given.
```

```
</ng-container>  
</ng-template>
```

You do not have to understand the details of this particular implementation. The solution in the sign-up form is just one possibility to display errors, avoid repetition and set ARIA attributes for accessibility.

From the user perspective and also from a testing perspective, it does not matter how you implement the rendering of error messages – as long as they are present and accessible.

IMPLEMENTATION DETAILS

We are going to test the `SignupFormComponent` in conjunction with `ControlErrorsComponent` and `ErrorMessageDirective` in a **black-box integration test**. For this test, the latter two will be irrelevant implementation details.

- 🔗 [Angular guide: Validating form input](#)
- 🔗 [MDN: Introduction to ARIA](#)
- 🔗 [MDN: aria-invalid](#)
- 🔗 [ARIA specification: aria-errormessage](#)
- 🔗 [ErrorMessageDirective: full code](#)
- 🔗 [ControlErrorsComponent: full code](#)

Test plan

What are the important parts of the sign-up form that need to be tested?

1. Form submission

- [Successful submission](#)
- [Do not submit the invalid form](#)
- [Submission failure](#)

2. [Required fields are marked as such and display error messages](#)

3. [Asynchronous validation of username, email and password](#)

4. [Dynamic field relations](#)

5. [Password type toggle](#)

6. [Accessibility of the form structure, field labels and error messages](#)

Test setup

Before writing the individual specs, we need to set up the suite in [signup-form.component.spec.ts](#). Let us start with the testing Module configuration.

```
await TestBed.configureTestingModule({
  imports: [ReactiveFormsModule],
  declarations: [
    SignupFormComponent,
    ControlErrorsComponent,
    ErrorMessageDirective
  ],
  providers: [
    { provide: SignupService, useValue: signupService }
  ],
}).compileComponents();
```

The Component under test contains a Reactive Form. That is why we import the `ReactiveFormsModule`:

```
imports: [ReactiveFormsModule],
```

DEEP RENDERING

As described, we are writing an integration test, so we declare the Component and its child components:

```
declarations: [
  SignupFormComponent,
  ControlErrorsComponent,
  ErrorMessageDirective
],
```

FAKE SERVICE

The `SignupFormComponent` depends on the `SignupService`. We do not want HTTP requests to the back-end when the tests run, so we [replace the Service with a fake instance](#).

```
providers: [  
  { provide: SignupService, useValue: signupService }  
],
```

A possible `SignupService` fake looks like this:

```
const signupService:  
  Pick<SignupService, keyof SignupService> = {  
  isUsernameTaken() {  
    return of(false);  
  },  
  isEmailTaken() {  
    return of(false);  
  },  
  getPasswordStrength() {  
    return of(strongPassword);  
  },  
  signup() {  
    return of({ success: true });  
  },  
};
```

This fake implements the *success case*: the username and email are available, the password is strong enough and the form submission was successful.

Since we are going to test several error cases as well, we need to create `SignupService` fakes dynamically. Also we need Jasmine spies to verify that the Service methods are called correctly.

CREATESPYOBJ

This is a job for Jasmine's `createSpyObj` (see [Faking Service dependencies](#)).

```
const signupService = jasmine.createSpyObj<SignupService>(
  'SignupService',
  {
    // Successful responses per default
    setUsernameTaken: of(false),
    setEmailTaken: of(false),
    getPasswordStrength: of(strongPassword),
    signup: of({ success: true }),
  }
);
```

SETUP FUNCTION

Together with the testing Module configuration, we put this code into a setup function. To adjust the `SignupService` fake behavior, we allow passing method return values.

```
describe('SignupFormComponent', () => {
  let fixture: ComponentFixture<SignupFormComponent>;
  let signupService: jasmine.SpyObj<SignupService>;

  const setup = async (
    signupServiceReturnValues?:
      jasmine.SpyObjMethodNames<SignupService>,
  ) => {
    signupService = jasmine.createSpyObj<SignupService>(
      'SignupService',
      {
        // Successful responses per default
        setUsernameTaken: of(false),
```

```

        isEmailTaken: of(false),
        getPasswordStrength: of(strongPassword),
        signup: of({ success: true }),
        // Overwrite with given return values
        ...signupServiceReturnValues,
    }
});

await TestBed.configureTestingModule({
    imports: [ReactiveFormsModule],
    declarations: [
        SignupFormComponent,
        ControlErrorsComponent,
        ErrorMessageDirective
    ],
    providers: [
        { provide: SignupService, useValue: signupService }
    ],
}).compileComponents();

fixture = TestBed.createComponent(SignupFormComponent);
fixture.detectChanges();

};

/* ... */
});

```

In all following specs, we are going to call `setup` first. If we simply write `async setup()`, the `SignupService` fake returns successful responses.

We can pass an object with different return values to simulate failure. For example, when testing that the username is taken:

```
await setup({
  // Let the API return that the username is taken
  isUsernameTaken: of(true),
});
```

Such a `setup` function is just one way to create fakes and avoid repetition. You might come up with a different solution that serves the same purpose.

 [SignupFormComponent: test code](#)

Successful form submission

The first case we need to test is the successful form submission. If the user fills out all required fields and the validations pass, we expect the Component to call `SignupService`'s `signup` method with the entered form data.

TEST DATA

The first step is to define *valid test data* we can fill into the form. We put this in a separate file, [signup-data.spec-helper.ts](#):

```
export const username = 'quickBrownFox';
export const password = 'dog lazy the over jumps fox brown quick the';
export const email = 'quick.brown.fox@example.org';
export const name = 'Mr. Fox';
export const addressLine1 = '';
export const addressLine2 = 'Under the Tree 1';
export const city = 'Farmtown';
```



```
export const postcode = '123456';
export const region = 'Upper South';
export const country = 'Luggnagg';

export const signupData: SignupData = {
  plan: 'personal',
  username,
  email,
  password,
  address: {
    name, addressLine1, addressLine2,
    city, postcode, region, country
  },
  tos: true,
};
```

In the [signup-form.component.html](#) template, all field elements need to be marked with test ids so we can find them and enter the values programmatically.

For example, the username input gets the test id `username`, the email input gets `email` and so on.

Back in `signup-form.component.spec.ts`, we create a new spec that calls the setup function.

```
it('submits the form successfully', async () => {
  await setup();

  /* ... */
});
```

FILL OUT FORM

Next, we fill out all required fields with valid values. Since we need to do that in several upcoming specs, let us create a reusable function.

```
const fillForm = () => {
  setFieldValue(fixture, 'username', username);
  setFieldValue(fixture, 'email', email);
  setFieldValue(fixture, 'password', password);
  setFieldValue(fixture, 'name', name);
  setFieldValue(fixture, 'addressLine1', addressLine1);
  setFieldValue(fixture, 'addressLine2', addressLine2);
  setFieldValue(fixture, 'city', city);
  setFieldValue(fixture, 'postcode', postcode);
  setFieldValue(fixture, 'region', region);
  setFieldValue(fixture, 'country', country);
  checkField(fixture, 'tos', true);
};
```

The `fillForm` function lies in the scope of `describe` so it may access the `fixture` variable. It uses the `setFieldValue` and `checkField` [element testing helpers](#).

In the spec, we call `fillForm`:

```
it('submits the form successfully', async () => {
  await setup();

  fillForm();

  /* ... */
});
```

Let us try to submit the form immediately after. The form under test listens for an [ngSubmit event](#) at the `form` element. This boils down to a native `submit` event.

SUBMIT FORM

We find the `form` element by its test id and simulate a `submit` event (see [Triggering event handlers](#)).

Then we expect the `signup` spy to have been called with the entered data.

```
it('submits the form successfully', async () => {  
  await setup();  
  
  fillForm();  
  
  findEl(fixture, 'form').triggerEventHandler('submit', {});  
  
  expect(signupService.signup).toHaveBeenCalledWith(signupData);  
});
```

If we run this spec, we find that it fails:

```
Expected spy SignupService.signup to have been called with:  
  [ Object({ plan: 'personal', ... }) ]  
but it was never called.
```

The spec fails because the form is still in the *invalid* state even though we have filled out all fields correctly.

ASYNC VALIDATORS

The cause are the **asynchronous validators** for username, email and password. When the user stops typing into these fields, they wait for one second before sending a request to the server.

In production, the HTTP request takes additional time, but our fake `SignupService` returns the response instantly.

ONE SECOND DEBOUNCE

This technique to reduce the amount of requests is called *debouncing*. For example, typing the username “fox” should send *one* request with “fox”, not *three* subsequent requests with “f”, “fo”, “fox”.

The spec above submits the form immediately after filling out the fields. At this point in time, the asynchronous validators have been called but have not returned a value yet. They are still waiting for the debounce period to pass.

In consequence, the test needs to wait one second for the asynchronous validators. An easy way would be to write an asynchronous test that uses `setTimeout(() => { /* ... */}, 1000)`. But this would slow down our specs.

FAKEASYNC AND TICK

Instead, we are going to use Angular’s `fakeAsync` and `tick` functions to *simulate* the passage of time. They are a powerful

couple to test asynchronous behavior.

`fakeAsync` freezes time. It hooks into the processing of asynchronous tasks created by timers, intervals, Promises and Observables. It prevents these tasks from being executed.

SIMULATE PASSAGE OF TIME

Inside the time warp created by `fakeAsync`, we use the `tick` function to simulate the passage of time. The scheduled tasks are executed and we can test their effect.

The specialty of `fakeAsync` and `tick` is that the passage of time is only virtual. Even if one second passes in the simulation, the spec still completes in a few milliseconds.

`fakeAsync` wraps the spec function which is also an `async` function due to the `setup` call. After filling out the form, we simulate the waiting with `tick(1000)`.

```
it('submits the form successfully', fakeAsync(async () => {
  await setup();

  fillForm();

  // Wait for async validators
  tick(1000);

  findEl(fixture, 'form').triggerEventHandler('submit', {});
```

```
    expect(signupService.signup).toHaveBeenCalledWith(signupData);
  }));
```

This spec passes! Now we should add some expectations to test the details.

First, we expect the asynchronous validators to call the `SignupService` methods with the user input. The methods are `isUsernameTaken`, `isEmailTaken` and `getPasswordStrength`.

```
it('submits the form successfully', fakeAsync(async () => {
  await setup();

  fillForm();

  // Wait for async validators
  tick(1000);

  findEl(fixture, 'form').triggerEventHandler('submit', {});

  expect(signupService.isUsernameTaken).toHaveBeenCalledWith(username);
  expect(signupService.isEmailTaken).toHaveBeenCalledWith(email);
  expect(signupService.getPasswordStrength).toHaveBeenCalledWith(password);
  expect(signupService.signup).toHaveBeenCalledWith(signupData);
}));
```

SUBMIT BUTTON

Next, we make sure that the submit button is disabled initially. After successful validation, the button is enabled. (The submit button carries the test id `submit`.)

STATUS MESSAGE

Also, when the form has been submitted successfully, the status message “Sign-up successful!” needs to appear. (The status message carries the test id `status`.)

This brings us to the final spec:

```
it('submits the form successfully', fakeAsync(async () => {
  await setup();

  fillForm();
  fixture.detectChanges();

  expect(findEl(fixture, 'submit').properties.disabled).toBe(true);

  // Wait for async validators
  tick(1000);
  fixture.detectChanges();

  expect(findEl(fixture, 'submit').properties.disabled).toBe(false);

  findEl(fixture, 'form').triggerEventHandler('submit', {});
  fixture.detectChanges();

  expectText(fixture, 'status', 'Sign-up successful!');

  expect(signupService.isUsernameTaken).toHaveBeenCalledWith(username);
  expect(signupService.isEmailTaken).toHaveBeenCalledWith(email);
  expect(signupService.getPasswordStrength).toHaveBeenCalledWith(password);
  expect(signupService.signup).toHaveBeenCalledWith(signupData);
}));
```

Because we are testing DOM changes, we have to call `detectChanges` after each *Act* phase.

 [SignupFormComponent: test code](#)

 [Angular API reference: fakeAsync](#)

 [Angular API reference: tick](#)

Invalid form

Now that we have tested the successful form submission, let us check the handling of an invalid form. What happens if we do not fill out any fields, but submit the form?

We create a new spec for this case:

```
it('does not submit an invalid form', fakeAsync(async () => {
  await setup();

  // Wait for async validators
  tick(1000);

  findEl(fixture, 'form').triggerEventHandler('submit', {});

  expect(signupService.isUsernameTaken).not.toHaveBeenCalled();
  expect(signupService.isEmailTaken).not.toHaveBeenCalled();
  expect(signupService.getPasswordStrength).not.toHaveBeenCalled();
  expect(signupService.signup).not.toHaveBeenCalled();
}));
```


This spec does less than the previous. We wait for a second and submit the form without entering data. Finally, we expect that no `SignupService` method has been called.

 [SignupFormComponent: test code](#)

Form submission failure

We have already tested the successful form submission. Now let us test the form submission failure.

REASONS FOR FAILURE

Despite correct input, the submission may fail for several reasons:

- The network is unavailable
- The back-end processed the request but returned an error:
 - The server-side validation failed
 - The request structure is not as expected
 - The server code has bugs, has crashed or is frozen

OBSERVABLE

When the user submits the form, the Component under tests calls the `SignupService`'s `signup` method.

- In the *success case*, the `signup` method returns an Observable that emits the value `{ success: true }` and completes. The form displays a status message "Sign-up successful!".
- In the *error case*, the Observable fails with an error. The form displays a status message "Sign-up error".

Let us test the latter case in a new spec. The structure resembles the spec for the successful submission. But we configure the fake `signup` method to return an Observable that fails with an error.

```
import { throwError } from 'rxjs';

it('handles signup failure', fakeAsync(async () => {
  await setup({
    // Let the API report a failure
    signup: throwError(new Error('Validation failed')),
  });

  /* ... */
});
```

We fill out the form, wait for the validators and submit the form.

```
fillForm();

// Wait for async validators
tick(1000);
```

```
findEl(fixture, 'form').triggerEventHandler('submit', {});  
fixture.detectChanges();
```

STATUS MESSAGE

Finally, we expect the “Sign-up error” status message to appear. Also, we verify that the relevant `SignupService` methods have been called.

```
expectText(fixture, 'status', 'Sign-up error');  
  
expect(signupService.isUsernameTaken).toHaveBeenCalledWith(username);  
expect(signupService.getPasswordStrength).toHaveBeenCalledWith(password);  
expect(signupService.signup).toHaveBeenCalledWith(signupData);
```

The full spec:

```
it('handles signup failure', fakeAsync(async () => {  
  await setup({  
    // Let the API report a failure  
    signup: throwError(new Error('Validation failed')),  
  });  
  
  fillForm();  
  
  // Wait for async validators  
  tick(1000);  
  
  findEl(fixture, 'form').triggerEventHandler('submit', {});  
  fixture.detectChanges();  
  
  expectText(fixture, 'status', 'Sign-up error');
```

```
expect(signupService.isUsernameTaken).toHaveBeenCalledWith(username);  
expect(signupService.getPasswordStrength).toHaveBeenCalledWith(password);  
expect(signupService.signup).toHaveBeenCalledWith(signupData);  
}));
```

 [SignupFormComponent: test code](#)

Required fields

A vital form logic is that certain fields are required and that the user interface conveys the fact clearly. Let us write a spec that checks whether required fields as marked as such.

REQUIREMENTS

The requirements are:

- A required field has an `aria-required` attribute.
- A required, invalid field has an `aria-errormessage` attribute. It contains the id of another element.
- This element contains an error message "... must be given".
(The text for the Terms of Services checkbox reads "Please accept the Terms and Services" instead.)

Our spec needs to verify all required fields, so we compile a list of their respective test ids:

```
const requiredFields = [  
  'username',  
  'email',  
  'name',  
  'addressLine2',  
  'city',  
  'postcode',  
  'country',  
  'tos',  
];
```

INVALID && (TOUCHED || DIRTY)

Before examining the fields, we need to trigger the display of form errors. As described in [Form validation and errors](#), error messages are shown when the field is *invalid* and either *touched* or *dirty*.

Luckily, the empty, but required fields are already invalid. Entering text would make them *dirty* but also *valid*.

MARK AS TOUCHED

So we need to *touch* the fields. If a field is focussed and loses focus again, Angular considers it as *touched*. Under the hood, Angular listens for the `blur` event.

In our spec, we simulate a `blur` event using the `dispatchFakeEvent` testing helper. Let us put the call in a reusable function:

```
const markFieldAsTouched = (element: DebugElement) => {  
  dispatchFakeEvent(element.nativeElement, 'blur');  
};
```

We can now write the *Arrange* and *Act* phases of the spec:

```
it('marks fields as required', async () => {  
  await setup();  
  
  // Mark required fields as touched  
  requiredFields.forEach((testId) => {  
    markFieldAsTouched(findEl(fixture, testId));  
  });  
  fixture.detectChanges();  
  
  /* ... */  
});
```

A `forEach` loop walks through the required field test ids, finds the element and marks the field as touched. We call `detectChanges` afterwards so the error messages appear.

ARIA-REQUIRED

Next, the *Assert* phase. Again we walk through the required fields to examine each one of them. Let us start with the `aria-required` attribute.

```
requiredFields.forEach((testId) => {  
  const el = findEl(fixture, testId);  
  
  // Check aria-required attribute  
  expect(el.attributes['aria-required']).toBe(
```

```

    'true',
    `${testId} must be marked as aria-required`,
  );

  /* ... */
});

```

`findEl` returns a `DebugElement` with an `attributes` property. This object contains all attributes set by the template. We expect the attribute `aria-required="true"` to be present.

ARIA-ERRORMESSAGE

The next part tests the error message with three steps:

1. Read the `aria-errormessage` attribute. Expect that it is set.
2. Find the element that `aria-errormessage` refers to. Expect that it exists.
3. Read the text content. Expect an error message.

Step 1 looks like this:

```

// Check aria-errormessage attribute
const errorMessageId = el.attributes['aria-errormessage'];
if (!errorMessageId) {
  throw new Error(`Error message id for ${testId} not present`);
}

```

Normally, we would use a Jasmine expectation like `expect(errorMessageId).toBeDefined()`. But `errorMessageId`

has the type `string | null` whereas we need a `string` in the upcoming commands.

TYPE ASSERTIONS

We need a TypeScript type assertion that rules out the `null` case and narrows down the type to `string`. If the attribute is absent or empty, we throw an exception. This fails the test with the given error and ensures that `errorMessageId` is a string for the rest of the spec.

Step 2 finds the error message element:

```
// Check element with error message
const errorMessageEl = document.getElementById(errorMessageId);
if (!errorMessageEl) {
  throw new Error(`Error message element for ${testId} not found`);
}
```

We use the native DOM method `document.getElementById` to find the element. `errorMessageEl` has the type `HTMLElement | null`, so we rule out the `null` case to work with `errorMessageEl`.

ERROR MESSAGE

Finally, we ensure that the element contains an error message, with a special treatment of the Terms and Services message.

```
if (errorMessageId === 'tos-errors') {
  expect(errorMessageEl.textContent).toContain(
    'Please accept the Terms and Services',
  );
}
```



```
    );  
  } else {  
    expect(errorMessageEl.textContent).toContain('must be given');  
  }  
}
```

The full spec looks like this:

```
it('marks fields as required', async () => {  
  await setup();  
  
  // Mark required fields as touched  
  requiredFields.forEach((testId) => {  
    markFieldAsTouched(findEl(fixture, testId));  
  });  
  fixture.detectChanges();  
  
  requiredFields.forEach((testId) => {  
    const el = findEl(fixture, testId);  
  
    // Check aria-required attribute  
    expect(el.attributes['aria-required']).toBe(  
      'true',  
      `${testId} must be marked as aria-required`,  
    );  
  
    // Check aria-errormessage attribute  
    const errorMessageId = el.attributes['aria-errormessage'];  
    if (!errorMessageId) {  
      throw new Error(  
        `Error message id for ${testId} not present`  
      );  
    }  
  
    // Check element with error message  
    const errorMessageEl = document.getElementById(errorMessageId);
```

```
if (!errorMessageEl) {  
  throw new Error(  
    `Error message element for ${testId} not found`  
  );  
}  
if (errorMessageId === 'tos-errors') {  
  expect(errorMessageEl.textContent).toContain(  
    'Please accept the Terms and Services',  
  );  
} else {  
  expect(errorMessageEl.textContent).toContain('must be given');  
}  
});  
});
```

 [SignupFormComponent: test code](#)

Asynchronous validators

The sign-up form features asynchronous validators for username, email and password. They are asynchronous because they wait for a second and make an HTTP request. Under the hood, they are implemented using RxJS Observables.

ASYNC VALIDATION FAILURE

We have already covered the “happy path” in which the entered username and email are available and the password is strong enough. We need to write three specs for the error cases: Username or email are taken and the password is too weak.

The validators call `SignupService` methods. Per default, the `SignupService` fake returns successful responses.

```
const setup = async (
  signupServiceReturnValues?:
    jasmine.SpyObjMethodNames<SignupService>,
) => {
  signupService = jasmine.createSpyObj<SignupService>(
    'SignupService',
    {
      // Successful responses per default
      isUsernameTaken: of(false),
      isEmailTaken: of(false),
      getPasswordStrength: of(strongPassword),
      signup: of({ success: true }),
      // Overwrite with given return values
      ...signupServiceReturnValues,
    }
  );

  /* ... */
};
```

The `setup` function allows us to overwrite the fake behavior. It accepts an object with `SignupService` method return values.

We add three specs that configure the fake accordingly:

```
it('fails if the username is taken', fakeAsync(async () => {
  await setup({
    // Let the API return that the username is taken
    isUsernameTaken: of(true),
  });
```

```

    /* ... */
  }));

it('fails if the email is taken', fakeAsync(async () => {
  await setup({
    // Let the API return that the email is taken
    isEmailTaken: of(true),
  });
  /* ... */
}));

it('fails if the password is too weak', fakeAsync(async () => {
  await setup({
    // Let the API return that the password is weak
    getPasswordStrength: of(weakPassword),
  });
  /* ... */
}));

```

The rest is the same for all three specs. Here is the first spec:

```

it('fails if the username is taken', fakeAsync(async () => {
  await setup({
    // Let the API return that the username is taken
    isUsernameTaken: of(true),
  });

  fillForm();

  // Wait for async validators
  tick(1000);
  fixture.detectChanges();

  expect(findEl(fixture, 'submit').properties.disabled).toBe(true);

```

```
findEl(fixture, 'form').triggerEventHandler('submit', {});

expect(signupService.isUsernameTaken).toHaveBeenCalledWith(username);
expect(signupService.isEmailTaken).toHaveBeenCalledWith(email);
expect(signupService.getPasswordStrength).toHaveBeenCalledWith(password);
expect(signupService.signup).not.toHaveBeenCalled();
}));
```

We fill out the form, wait for the async validators and try to submit the form.

We expect that the three async validators call the respective `SignupService` methods.

The username validation fails, so we expect that the Component prevents the form submission. The `signup` method must not be called.

As stated above, the two other specs `it('fails if the email is taken', /* ... */)` and `it('fails if the password is too weak', /* ... */)` look the same apart from the fake setup.

 [SignupFormComponent: test code](#)

 [Angular documentation: Creating asynchronous validators](#)

Dynamic field relations

The sign-up form has a fixed set of fields. But the `addressLine1` field depends on the value of the `plan` field:

- If the selected plan is “Personal”, the field is optional and the label reads “Address line 1”.
- If the selected plan is “Business”, the field is required and the label reads “Company”.
- If the selected plan is “Education & Non-profit”, the field is required and the label reads “Organization”.

The implementation in the Component class looks like this:

```
this.plan.valueChanges.subscribe((plan: Plan) => {  
  if (plan !== this.PERSONAL) {  
    this.addressLine1.setValidators(required);  
  } else {  
    this.addressLine1.setValidators(null);  
  }  
  this.addressLine1.updateValueAndValidity();  
});
```

We listen for value changes of the `plan` control. Depending on the selection, we either add the `required` validator to the `addressLine1` control or remove all validators.

Finally, we need to tell Angular to revalidate the field value, now that the validators have changed.

Let us write a spec to ensure that `addressLine1` is required for certain plans.

```
it('requires address line 1 for business and non-profit plans', async () =>
{
  await setup();

  /* ... */
});
```

First, we need inspect the initial state: The “Personal” plan is selected and `addressLine1` is optional.

We do so by looking at attributes of the `addressLine1` field element: The `ng-invalid` class and the `aria-required` attribute must be absent.

```
// Initial state (personal plan)
const addressLine1El = findEl(fixture, 'addressLine1');
expect('ng-invalid' in addressLine1El.classes).toBe(false);
expect('aria-required' in addressLine1El.attributes).toBe(false);
```

From this baseline, let us change the plan from “Personal” to “Business”. We use the `checkField` spec helper to activate the corresponding radio button.

```
// Change plan to business
checkField(fixture, 'plan-business', true);
fixture.detectChanges();
```

To see the effect of the change, we need to tell Angular to update the DOM. Then we expect the `ng-invalid` class and the `aria-required` to be present.

```
expect(addressLine1El.attributes['aria-required']).toBe('true');  
expect(addressLine1El.classes['ng-invalid']).toBe(true);
```

We perform the same check for the “Education & Non-profit” plan.

```
// Change plan to non-profit  
checkField(fixture, 'plan-non-profit', true);  
fixture.detectChanges();
```

```
expect(addressLine1El.attributes['aria-required']).toBe('true');  
expect(addressLine1El.classes['ng-invalid']).toBe(true);
```

This is it! Here is the full spec:

```
it('requires address line 1 for business and non-profit plans', async () =>  
{  
  await setup();  
  
  // Initial state (personal plan)  
  const addressLine1El = findEl(fixture, 'addressLine1');  
  expect('ng-invalid' in addressLine1El.classes).toBe(false);  
  expect('aria-required' in addressLine1El.attributes).toBe(false);  
  
  // Change plan to business  
  checkField(fixture, 'plan-business', true);  
  fixture.detectChanges();  
  
  expect(addressLine1El.attributes['aria-required']).toBe('true');
```



```
expect(addressLine1El.classes['ng-invalid']).toBe(true);

// Change plan to non-profit
checkField(fixture, 'plan-non-profit', true);
fixture.detectChanges();

expect(addressLine1El.attributes['aria-required']).toBe('true');
expect(addressLine1El.classes['ng-invalid']).toBe(true);
});
```

We have already checked the presence of `aria-required` attributes when testing the [required fields](#). For consistency, we check for `aria-required` in this spec as well.

As a second indicator, we check for the `ng-invalid` class. This class is set by Angular itself on invalid form fields without us having to add it via the template. Note that the mere presence of the class does not imply that the invalid state is conveyed visually.

Alternatively, we could check for the presence of an error message, like we did in the required fields spec.

 [SignupFormComponent: test code](#)

Password type toggle

Another small feature of the sign-up form is the password type switcher. This button toggles the visibility of the entered

password. Under the hood, it changes the input type from `password` to `text` and vice versa.

The Component class stores the visibility in a boolean property:

```
public showPassword = false;
```

In the template, the input type depends on the property (shortened code):

```
<input [type]="showPassword ? 'text' : 'password'" />
```

Finally, the button toggles the boolean value (shortened code):

```
<button
  type="button"
  (click)="showPassword = !showPassword"
>
  {{ showPassword ? '🔒 Hide password' : '👁 Show password' }}
</button>
```

To test this feature, we create a new spec:

```
it('toggles the password display', async () => {
  await setup();

  /* ... */
});
```

Initially, the field has the `password` type so the entered text is obfuscated. Let us test this baseline.

First, we enter a password into the field. This is not strictly necessary but makes the test more realistic and debugging easier. (The password field has the test id `password`.)

```
setFieldValue(fixture, 'password', 'top secret');
```

We find the input element by its test id again to check the `type` attribute.

```
const passwordEl = findEl(fixture, 'password');  
expect(passwordEl.attributes.type).toBe('password');
```

Now we click on the toggle button for the first time. We let Angular update the DOM and check the input type again.

```
click(fixture, 'show-password');  
fixture.detectChanges();  
  
expect(passwordEl.attributes.type).toBe('text');
```

We expect that the type has changed from `password` to `text`. The password is now visible.

With a second click on the toggle button, the type switches back to `password`.

```
click(fixture, 'show-password');  
fixture.detectChanges();  
  
expect(passwordEl.attributes.type).toBe('password');
```

This is the whole spec:

```
it('toggles the password display', async () => {  
  await setup();  
  
  setFieldValue(fixture, 'password', 'top secret');  
  const passwordEl = findEl(fixture, 'password');  
  expect(passwordEl.attributes.type).toBe('password');  
  
  click(fixture, 'show-password');  
  fixture.detectChanges();  
  
  expect(passwordEl.attributes.type).toBe('text');  
  
  click(fixture, 'show-password');  
  fixture.detectChanges();  
  
  expect(passwordEl.attributes.type).toBe('password');  
});
```

 [SignupFormComponent: test code](#)

Testing form accessibility

Web accessibility means that all people can use a web site, regardless of their physical or mental abilities or web access technologies. It is part of a greater effort called Inclusive Design, the process of creating information systems that account for people with diverse abilities and needs.

Designing web forms is a usability and accessibility challenge. Web forms often pose a barrier for users with disabilities and users of assistive technologies.

ACCESSIBLE FORM

The sign-up form has several accessibility features, among others:

- The form is well-structured with heading, `fieldset` and `legend` elements.
- All fields have proper labels, e.g. "Username (required)".
- Some fields have additional descriptions. The descriptions are linked with `aria-describedby` attributes.
- Required fields are marked with `aria-required="true"`.
- Invalid fields are marked with `aria-invalid="true"`. The error messages are linked with `aria-errormessage` attributes.
- When the form is submitted, the result is communicated using a status message with `role="status"`.
- The structure and styling clearly conveys the current focus as well as the validity state.

AUTOMATED ACCESSIBILITY TESTING

There are many more accessibility requirements and best practices that we have not mentioned. Since this guide is not about creating accessible forms primarily, let us explore how to **test accessibility in an automated way**.

We have tested some of the features above in the [SignupFormComponent](#)'s integration test. Instead of writing more specs for accessibility requirements by hand, let us test the accessibility with a proper tool.

 [Inclusive Design Principles](#)

pa11y

In this guide, we will look at **pa11y**, a Node.js program that checks the accessibility of a web page.

TESTS IN CHROME

pa11y starts and remotely-controls a Chrome or Chromium browser. The browser navigates to the page under test. pa11y then injects *axe-core* and/or *HTML CodeSniffer*, two accessibility testing engines.

These engines check compliance with the Web Content Accessibility Guidelines (WCAG), the authoritative technical standard for web accessibility.

CLI VS. CI

pa11y has two modes of operation: The command line interface (CLI) for checking one web page and the continuous integration (CI) mode for checking multiple web pages.

For quickly testing a single page of your Angular application, use the command line interface. When testing the whole application on a regular basis, use the continuous integration mode.

To use the command line interface, install pa11y as a global npm module:

```
npm install -g pa11y
```

TEST SINGLE PAGE

This installs the global command `pa11y`. To test a page on the local Angular development server, run:

```
pa11y http://localhost:4200/
```

For the sign-up form, pa11y does not report any errors:

```
Welcome to Pa11y
```

```
> Running Pa11y on URL http://localhost:4200/
```

```
No issues found!
```

ERROR REPORT

If one of the form fields did not have a proper label, pa11y would complain:

- Error: This textinput element does not have a name available to an accessibility API. Valid names are: label element, title undefined, aria-label undefined, aria-labelledby undefined.
└─ WCAG2AA.Principle4.Guideline4_1.4_1_2.H91.InputText.Name
└─ html > body > app-root > main > app-signup-form > form > fieldset:nth-child(2) > div:nth-child(2) > p > span > input
└─ <input _ngcontent-srr-c42="" type="text" formcontrolname="username" ...
- Error: This form field should be labelled in some way. Use the label element (either with a "for" attribute or wrapped around the form field), or "title", "aria-label" or "aria-labelledby" attributes as appropriate.
└─ WCAG2AA.Principle1.Guideline1_3.1_3_1.F68
└─ html > body > app-root > main > app-signup-form > form > fieldset:nth-child(2) > div:nth-child(2) > p > span > input
└─ <input _ngcontent-srr-c42="" type="text" formcontrolname="username" ...

Each error message contains the violated WCAG rule, the DOM path to the violating element and its HTML code.

- 🔗 [pa11y: Accessibility testing tools](#)
- 🔗 [axe-core: Accessibility engine for automated Web UI testing](#)
- 🔗 [HTML CodeSniffer: Accessibility auditor](#)
- 🔗 [Web Content Accessibility Guidelines \(WCAG\) 2.1](#)

pa11y-ci

For comprehensive test runs both during development and on a build server, we will set up pa11y in the continuous integration mode.

In your Angular project directory, install the `pa11y-ci` package:

```
npm install pa11y-ci
```

pa11y-ci expects a configuration file named `.pa11y-ci` in the project directory. Create the file and paste this JSON:

```
{
  "defaults": {
    "runner": [
      "axe",
      "htmlcs"
    ]
  },
  "urls": [
    "http://localhost:4200"
  ]
}
```

TEST MULTIPLE URLS

This configuration tells pa11y to check the URL `http://localhost:4200` and to use both available testing engines, `axe` and `htmlcs`. You can add many URLs to the `urls` array.

We can now run pa11y-ci with:

```
npx pa11y-ci
```

For the sign-up form, we get this output:

Running Pa11y on 1 URLs:

> http://localhost:4200 - 0 errors

✓ 1/1 URLs passed

 [pa11y-ci: CI-centric accessibility test runner](#)

Start server and run pa11y-ci

The configuration above expects that the development server is already running at http://localhost:4200. Both in development and on a build server, it is useful to start the Angular server, run the accessibility tests and then stop the server again.

We can achieve this with another handy Node.js package, [start-server-and-test](#).

```
npm install start-server-and-test
```

[start-server-and-test](#) first runs an npm script that is supposed to start an HTTP server. Then it waits for the server to boot up. Once a given URL is available, it runs another npm script.

In our case, the first script is [start](#), an alias for [ng serve](#). We need to create the second script to run [pa11y-ci](#).

We edit package.json, and add two scripts:

```
{
  "scripts": {
    "a11y": "start-server-and-test start http-get://localhost:4200/ pa11y-ci",
    "pa11y-ci": "pa11y-ci"
  },
}
```

Now, `npm run a11y` starts the Angular development server, then runs pa11y-ci, finally stops the server. The audit result is written to the standard output.

 [start-server-and-test: Starts server, waits for URL, then runs test command](#)

Form accessibility: Summary

pa11y is a powerful set of tools with many options. We have barely touched on its features.

Automated accessibility testing is a valuable addition to unit, integration and end-to-end tests. You should run an accessibility tester like pa11y against the pages of your Angular application. It is especially helpful to ensure the accessibility of complex forms.

Bear in mind that automated testing only points out certain accessibility barriers that can be detected programmatically.

The Web Content Accessibility Guidelines (WCAG) establish – from abstract to specific – *principles, guidelines* and *success criteria*. The latter are the practical rules some of which can be checked automatically.

The WCAG success criteria are accompanied by *techniques* for HTML, CSS, JavaScript etc. For JavaScript web applications, techniques like ARIA are especially relevant.

In summary, you need to learn about accessibility and Inclusive Design first, apply the rules while designing and implementing the application. Then check the compliance manually and automatically.

 [Web Content Accessibility Guidelines \(WCAG\) 2.1](#)

 [Quick Reference: How to Meet WCAG](#)

 [WebAIM: Introduction to ARIA - Accessible Rich Internet Applications](#)

 [Web Accessibility Tutorial: Forms](#)

Testing Components with Spectator

LEARNING OBJECTIVES

- Simplifying Component tests with the Spectator library
- Using the unified Spectator interface
- Interacting with the Component and the rendered DOM
- Dispatching synthetic DOM events to simulate user input
- Using Spectator and ng-mocks to fake child Components and Services

We have used Angular's testing tools to set up modules, render Components, query the DOM and more. These tools are [TestBed](#), [ComponentFixture](#) and [DebugElement](#), also [HttpClientTestingModule](#) and [RouterTestingModule](#).

STRUCTURAL WEAKNESSES

The built-in tools are fairly low-level and unopinionated. They have several drawbacks:

- [TestBed](#) requires a large amount of boilerplate code to set up a common Component or Service test.
- [DebugElement](#) lacks essential features and is a "leaky" abstraction. You are forced to work with the wrapped native DOM element for common tasks.

- There are no default solutions for faking Components and Service dependencies safely.
- The tests itself get verbose and repetitive. You have to establish testing conventions and write helpers yourself.

We have already used small [element testing helpers](#). They solve isolated problems in order to write more consistent and compact specs.

If you write hundreds or thousands of specs, you will find that these helper functions do not suffice. They do not address the above-mentioned structural problems.

UNIFIED TESTING API

[Spectator](#) is an opinionated library for testing Angular applications. Technically, it sits on top of [TestBed](#), [ComponentFixture](#) and [DebugElement](#). But the main idea is to unify all these APIs in one consistent, powerful and user-friendly interface – the [Spectator](#) object.

Spectator simplifies testing Components, Services, Directives, Pipes, routing and HTTP communication. Spectator's strength are Component tests with Inputs, Outputs, children, event handling, Service dependencies and more.

For [faking child Components](#), Spectator resorts to the ng-mocks library just like we did.

This guide cannot introduce all Spectator features, but we will discuss the basics of Component testing using Spectator.

Both [example applications](#) are tested with our element helpers and also with Spectator. The former specs use the suffix `.spec.ts`, while the latter use the suffix `.spectator.spec.ts`. This way, you can compare the tests side-by-side.

In this chapter, we will discuss testing the Flickr search with Spectator.

Component with an Input

Let us start with the [FullPhotoComponent](#) because it is a [presentational Component](#), a leaf in the Component tree. It expects a `Photo` object as Input and renders an image as well as the photo metadata. No Outputs, no children, no Service dependencies.

The [FullPhotoComponent suite with our helpers](#) looks like this:

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

```

beforeEach(async () => {
  await TestBed.configureTestingModule({
    declarations: [FullPhotoComponent],
    schemas: [NO_ERRORS_SCHEMA],
  }).compileComponents();

  fixture = TestBed.createComponent(FullPhotoComponent);
  component = fixture.componentInstance;
  component.photo = photo1;
  fixture.detectChanges();
});

it('renders the photo information', () => {
  expectText(fixture, 'full-photo-title', photo1.title);

  const img = findEl(fixture, 'full-photo-image');
  expect(img.properties.src).toBe(photo1.url_m);
  expect(img.properties.alt).toBe(photo1.title);

  expectText(fixture, 'full-photo-ownername', photo1.ownername);
  expectText(fixture, 'full-photo-datetaken', photo1.datetaken);
  expectText(fixture, 'full-photo-tags', photo1.tags);

  const link = findEl(fixture, 'full-photo-link');
  expect(link.properties.href).toBe(photo1Link);
  expect(link.nativeElement.textContent.trim()).toBe(photo1Link);
});
});

```

This suite already benefits from `expectText` and `findEl`, but it is still using the leaky `DebugElement` abstraction.

COMPONENT FACTORY

When using Spectator, the Module configuration and the Component creation looks different. In the scope of the test suite, we create a **Component factory**:

```
import { createComponentFactory } from '@ngneat/spectator';

describe('FullPhotoComponent with spectator', () => {
  /* ... */

  const createComponent = createComponentFactory({
    component: FullPhotoComponent,
    shallow: true,
  });

  /* ... */
});
```

`createComponentFactory` expects a configuration object. `component: FullPhotoComponent` specifies the Component under test. `shallow: true` means we want [shallow, not deep rendering](#). It does not make a difference for `FullPhotoComponent` though since it has no children.

The configuration object may include more options for the testing Module, as we will see later.

Internally, `createComponentFactory` creates a `beforeEach` block that calls `TestBed.configureTestingModule` and

`TestBed.compileComponents`, just like we did manually.

`createComponentFactory` returns a factory function for creating a `FullPhotoComponent`. We save that function in the `createComponent` constant.

CREATE COMPONENT

The next step is to add a `beforeEach` block that creates the Component instance. `createComponent` again takes an options object. To set the `photo` Input property, we pass `props: { photo: photo1 }`.

```
import { createComponentFactory, Spectator } from '@ngneat/spectator';

describe('FullPhotoComponent with spectator', () => {
  let spectator: Spectator<FullPhotoComponent>;

  const createComponent = createComponentFactory({
    component: FullPhotoComponent,
    shallow: true,
  });

  beforeEach(() => {
    spectator = createComponent({ props: { photo: photo1 } });
  });

  /* ... */
});
```

SPECTATOR

`createComponent` returns a `Spectator` object. This is the powerful interface we are going to use in the specs.

The spec `it('renders the photo information', /* ... */)` repeats three essential tasks several times:

1. Find an element by test id
2. Check its text content
3. Check its attribute value

First, the spec finds the element with the test id `full-photo-title` and expects it to contain the photo's title.

With Spectator, it reads:

```
expect(
  spectator.query(byTestId('full-photo-title'))
).toHaveText(photo1.title);
```

`SPECTATOR.QUERY`

The central `spectator.query` method finds an element in the DOM. This guide recommends to [find elements by test ids](#) (`data-testid` attributes).

Spectator supports test ids out of the box, so we write:

```
spectator.query(byTestId('full-photo-title'))
```

`spectator.query` returns a native DOM element or `null` in case no match was found. Note that it does not return a `DebugElement`.

When using Spectator, you work directly with DOM element objects. What seems cumbersome at first glance, in fact lifts the burden of the leaky `DebugElement` abstraction.

JASMINE MATCHERS

Spectator makes it easy to work with plain DOM elements. Several matchers are added to Jasmine to create expectations on an element.

For checking an element's text content, Spectator provides the `toHaveText` matcher. This leads us to the following expectation:

```
expect(
  spectator.query(byTestId('full-photo-title'))
).toHaveText(photo1.title);
```

This code is equivalent to our `expectText` helper, but more idiomatic and fluent to read.

Next, we need to verify that the Component renders the full photo using an `img` element.

```
const img = spectator.query(byTestId('full-photo-image'));
expect(img).toHaveAttribute('src', photo1.url_m);
expect(img).toHaveAttribute('alt', photo1.title);
```

Here, we find the element with the test id `full-photo-image` to check its `src` and `alt` attributes. We use Spectator's matcher `toHaveAttribute` for this purpose.

The rest of the spec finds more elements to inspect their contents and attributes.

The full test suite using Spectator (only imports from Spectator are shown):

```
import {
  byTestId, createComponentFactory, Spectator
} from '@ngneat/spectator';

describe('FullPhotoComponent with spectator', () => {
  let spectator: Spectator<FullPhotoComponent>;

  const createComponent = createComponentFactory({
    component: FullPhotoComponent,
    shallow: true,
  });

  beforeEach(() => {
    spectator = createComponent({ props: { photo: photo1 } });
  });

  it('renders the photo information', () => {
    expect(
      spectator.query(byTestId('full-photo-title'))
    ).toHaveText(photo1.title);

    const img = spectator.query(byTestId('full-photo-image'));
```

```
expect(img).toHaveAttribute('src', photo1.url_m);
expect(img).toHaveAttribute('alt', photo1.title);

expect(
  spectator.query(byTestId('full-photo-ownername'))
).toHaveText(photo1.ownername);
expect(
  spectator.query(byTestId('full-photo-datetaken'))
).toHaveText(photo1.datetaken);
expect(
  spectator.query(byTestId('full-photo-tags'))
).toHaveText(photo1.tags);

const link = spectator.query(byTestId('full-photo-link'));
expect(link).toHaveAttribute('href', photo1Link);
expect(link).toHaveText(photo1Link);
});
});
```

Compared to the version with custom testing helpers, the Spectator version is not necessarily shorter. But it works on a *consistent abstraction level*.

Instead of a wild mix of [TestBed](#), [ComponentFixture](#), [DebugElement](#) plus helper functions, there is the [createComponentFactory](#) function and one [Spectator](#) instance.

Spectator avoids wrapping DOM elements, but offers convenient Jasmine matchers for common DOM expectations.

🔗 [FullPhotoComponent: implementation code and the two tests](#)

🔗 [Spectator: Queries](#)

Component with children and Service dependency

Spectator really shines when testing [container Components](#). These are Components with children and Service dependencies.

In the Flickr search, the topmost [FlickrSearchComponent](#) calls the [FlickrService](#) and holds the state. It orchestrates three other Components, passes down the state and listens for Outputs.

The [FlickrSearchComponent](#) template:

```
<app-search-form (search)="handleSearch($event)"></app-search-form>
```

```
<div class="photo-list-and-full-photo">  
  <app-photo-list  
    [title]="searchTerm"  
    [photos]="photos"  
    (focusPhoto)="handleFocusPhoto($event)"  
    class="photo-list"  
  ></app-photo-list>
```

```
<app-full-photo  
  *ngIf="currentPhoto"  
  [photo]="currentPhoto"  
  class="full-photo"
```

```
        data-testid="full-photo"
    ></app-full-photo>
</div>
```

The FlickrSearchComponent class:

```
@Component({
  selector: 'app-flickr-search',
  templateUrl: './flickr-search.component.html',
  styleUrls: ['./flickr-search.component.css'],
})
export class FlickrSearchComponent {
  public searchTerm = '';
  public photos: Photo[] = [];
  public currentPhoto: Photo | null = null;

  constructor(private flickrService: FlickrService) {}

  public handleSearch(searchTerm: string): void {
    this.flickrService.searchPublicPhotos(searchTerm).subscribe(
      (photos) => {
        this.searchTerm = searchTerm;
        this.photos = photos;
        this.currentPhoto = null;
      }
    );
  }

  public handleFocusPhoto(photo: Photo): void {
    this.currentPhoto = photo;
  }
}
```

CHILD COMPONENTS

Since this is the Component where all things come together, there is much to test.

1. Initially, the `SearchFormComponent` and the `PhotoListComponent` are rendered, not the `FullPhotoComponent`. The photo list is empty.
2. When the `SearchFormComponent` emits the `search` Output, the `FlickrService` is called with the search term.
3. The search term and the photo list are passed down to the `PhotoListComponent` via Inputs.
4. When the `PhotoListComponent` emits the `focusPhoto` Output, the `FullPhotoComponent` is rendered. The selected photo is passed down via Input.

WITHOUT SPECTATOR

The [FlickrSearchComponent test suite with our helpers](#) looks like this:

```
describe('FlickrSearchComponent', () => {  
  let fixture: ComponentFixture<FlickrSearchComponent>;  
  let component: FlickrSearchComponent;  
  let fakeFlickrService: Pick<FlickrService, keyof FlickrService>;  
  
  let searchForm: DebugElement;  
  let photoList: DebugElement;
```

```

beforeEach(async () => {
  fakeFlickrService = {
    searchPublicPhotos: jasmine
      .createSpy('searchPublicPhotos')
      .and.returnValue(of(photos)),
  };

  await TestBed.configureTestingModule({
    imports: [HttpClientTestingModule],
    declarations: [FlickrSearchComponent],
    providers: [
      { provide: FlickrService, useValue: fakeFlickrService }
    ],
    schemas: [NO_ERRORS_SCHEMA],
  }).compileComponents();
});

beforeEach(() => {
  fixture = TestBed.createComponent(FlickrSearchComponent);
  component = fixture.debugElement.componentInstance;
  fixture.detectChanges();

  searchForm = findComponent(fixture, 'app-search-form');
  photoList = findComponent(fixture, 'app-photo-list');
});

it('renders the search form and the photo list, not the full photo', () =>
{
  expect(searchForm).toBeTruthy();
  expect(photoList).toBeTruthy();
  expect(photoList.properties.title).toBe('');
  expect(photoList.properties.photos).toEqual([]);

  expect(() => {

```

```

        findComponent(fixture, 'app-full-photo');
    }).toThrow();
});

it('searches and passes the resulting photos to the photo list', () => {
    const searchTerm = 'beautiful flowers';
    searchForm.triggerEventHandler('search', searchTerm);
    fixture.detectChanges();

    expect(fakeFlickrService.searchPublicPhotos).toHaveBeenCalledWith(searchTerm
);
    expect(photoList.properties.title).toBe(searchTerm);
    expect(photoList.properties.photos).toBe(photos);
});

it('renders the full photo when a photo is focussed', () => {
    expect(() => {
        findComponent(fixture, 'app-full-photo');
    }).toThrow();

    photoList.triggerEventHandler('focusPhoto', photo1);

    fixture.detectChanges();

    const fullPhoto = findComponent(fixture, 'app-full-photo');
    expect(fullPhoto.properties.photo).toBe(photo1);
});
});

```

Without going too much into detail, a few notes:

- We use [shallow rendering](#). The child Components are not declared and only empty shell elements are rendered ([app-search-form](#), [app-photo-list](#) and [app-full-photo](#)). This lets us check their presence, their Inputs and Outputs.
- We use our [findComponent](#) testing helper to find the child elements.
- To check the Input values, we use the [properties](#) of [DebugElements](#).
- To simulate that an Output emits, we use [triggerEventListener](#) on [DebugElements](#).
- We provide our own fake [FlickrService](#). It contains one Jasmine spy that returns a Observable with a fixed list of photos.

```
fakeFlickrService = {  
  searchPublicPhotos: jasmine  
    .createSpy('searchPublicPhotos')  
    .and.returnValue(of(photos)),  
};
```

WITH SPECTATOR

Rewriting this suite with Spectator brings two major changes:

1. We replace the child Components with fakes created by [ng-mocks](#). The fake Components mimic the originals regarding

their Inputs and Outputs, but they do not render anything. We will work with these Component instances instead of operating on `DebugElements`.

2. We use Spectator to create the fake `FlickrService`.

The test suite setup:

```
import {
  createComponentFactory, mockProvider, Spectator
} from '@ngneat/spectator';

describe('FlickrSearchComponent with spectator', () => {
  /* ... */

  const createComponent = createComponentFactory({
    component: FlickrSearchComponent,
    shallow: true,
    declarations: [
      MockComponents(
        SearchFormComponent, PhotoListComponent, FullPhotoComponent
      ),
    ],
    providers: [mockProvider(FlickrService)],
  });

  /* ... */
});
```

Again, we use Spectator's `createComponentFactory`. This time, we replace the child Components with fakes created by ng-mocks' `MockComponents` function.

MOCKPROVIDER

Then we use Spectator's `mockProvider` function to create a fake `FlickrService`. Under the hood, this works roughly the same as our manual `fakeFlickrService`. It creates an object that resembles the original, but the methods are replaced with Jasmine spies.

In a `beforeEach` block, the Component is created.

```
import {
  createComponentFactory, mockProvider, Spectator
} from '@ngneat/spectator';

describe('FlickrSearchComponent with spectator', () => {
  let spectator: Spectator<FlickrSearchComponent>;

  let searchForm: SearchFormComponent | null;
  let photoList: PhotoListComponent | null;
  let fullPhoto: FullPhotoComponent | null;

  const createComponent = createComponentFactory(/* ... */);

  beforeEach(() => {
    spectator = createComponent();

    spectator.inject(FlickrService).searchPublicPhotos.and.returnValue(of(photos
  ));

    searchForm = spectator.query(SearchFormComponent);
    photoList = spectator.query(PhotoListComponent);
    fullPhoto = spectator.query(FullPhotoComponent);
```

```
});  
  
/* ... */  
});
```

`spectator.inject` is the equivalent of `TestBed.inject`. We get hold of the `FlickrService` fake instance and configure the `searchPublicPhotos` spy to return fixed data.

FIND CHILDREN

`spectator.query` not only finds elements in the DOM, but also child Components and other nested Directives. We find the three child Components and save them in variables since they will be used in all specs.

Note that `searchForm`, `photoList` and `fullPhoto` are typed as Component instances, not `DebugElements`. This is accurate because the fakes have the same public interfaces, the same Inputs and Output.

Due to the [equivalence of fake and original](#), we can access Inputs with the pattern `componentInstance.input`. And we let an Output emit with the pattern `componentInstance.output.emit(...)`.

The first spec checks the initial state:

```
it('renders the search form and the photo list, not the full photo', () => {  
  if (!(searchForm && photoList)) {
```

```
        throw new Error('searchForm or photoList not found');
    }
    expect(photoList.title).toBe('');
    expect(photoList.photos).toEqual([]);
    expect(fullPhoto).not.toExist();
});
```

`spectator.query(PhotoListComponent)` either returns the Component instance or `null` if there is no such nested Component. Hence, the `photoList` variable is typed as `PhotoListComponent | null`.

MANUAL TYPE GUARD

Unfortunately, `expect` is not a [TypeScript type guard](#). Jasmine expectations cannot narrow down the type from `PhotoListComponent | null` to `PhotoListComponent`.

We cannot call `expect(photoList).not.toBe(null)` and continue with `expect(photoList.title).toBe('')`. The first expectation throws an error in the `null` case, but TypeScript does not know this. TypeScript still assumes the type `PhotoListComponent | null`, so it would complain about `photoList.title`.

This is why we manually throw an error when `photoList` is `null`. TypeScript infers that the type must be `PhotoListComponent` in the rest of the spec.

In contrast, our `findComponent` helper function throws an exception directly if no match was found, failing the test early. To verify that a child Component is absent, we had to expect this exception:

```
expect(() => {
  findComponent(fixture, 'app-full-photo');
}).toThrow();`
```

The Spectator spec goes on and uses `expect(fullPhoto).not.toExist()`, which is equivalent to `expect(fullPhoto).toBe(null)`. The Jasmine matcher `toExist` comes from Spectator.

TEST SEARCH

The second spec covers the search:

```
it('searches and passes the resulting photos to the photo list', () => {
  if (!(searchForm && photoList)) {
    throw new Error('searchForm or photoList not found');
  }
  const searchTerm = 'beautiful flowers';
  searchForm.search.emit(searchTerm);

  spectator.detectChanges();

  const flickrService = spectator.inject(FlickrService);
  expect(flickrService.searchPublicPhotos).toHaveBeenCalledWith(searchTerm);
  expect(photoList.title).toBe(searchTerm);
  expect(photoList.photos).toBe(photos);
});
```

When the `SearchFormComponent` emits a search term, we expect that the `FlickrService` has been called. In addition, we expect that the search term and the photo list from Service are passed to the `PhotoListComponent`.

`spectator.detectChanges()` is just Spectator's shortcut to `fixture.detectChanges()`.

TEST FOCUS PHOTO

The last spec focusses a photo:

```
it('renders the full photo when a photo is focussed', () => {
  expect(fullPhoto).not.toExist();

  if (!photoList) {
    throw new Error('photoList not found');
  }
  photoList.focusPhoto.emit(photo1);

  spectator.detectChanges();

  fullPhoto = spectator.query(FullPhotoComponent);
  if (!fullPhoto) {
    throw new Error('fullPhoto not found');
  }
  expect(fullPhoto.photo).toBe(photo1);
});
```

Again, the main difference is that we directly work with Inputs and Outputs.

- 🔗 [FlickrSearchComponent: implementation code and the two tests](#)
- 🔗 [ng-mocks: How to mock a component](#)
- 🔗 [Spectator: Mocking providers](#)

Event handling with Spectator

Most Components handle input events like mouse clicks, keypresses or form field changes. To simulate them, we have used the `triggerEventHandler` method on `DebugElements`. This method does not actually simulate DOM events, it merely calls the event handlers registered by `(click)="handler($event)"` and the like.

`triggerEventHandler` requires you to create an event object that becomes `$event` in the template. For this reason, we have introduced the `click` and `makeClickEvent` helpers.

SYNTHETIC EVENTS

Spectator takes a different approach: It dispatches synthetic DOM events. This makes the test more realistic. Synthetic events can bubble up in the DOM tree like real events. Spectator creates the event objects for you while you can configure the details.

SPECTATOR.CLICK

To perform a simple click, we use `spectator.click` and pass the target element or a `byTestId` selector. An example from the [PhotoItemComponent test](#):

```
describe('PhotoItemComponent with spectator', () => {
  /* ... */

  it('focusses a photo on click', () => {
    let photo: Photo | undefined;

    spectator.component.focusPhoto.subscribe((otherPhoto: Photo) => {
      photo = otherPhoto;
    });

    spectator.click(byTestId('photo-item-link'));

    expect(photo).toBe(photo1);
  });

  /* ... */
});
```

Another common task is to simulate form field input. So far, we have used the [setFieldValue helper](#) for this purpose.

SPECTATOR.TYPEINELEMENT

Spectator has an equivalent method named `spectator.typeInElement`. It is used by the [SearchFormComponent test](#):

```
describe('SearchFormComponent with spectator', () => {
  /* ... */

  it('starts a search', () => {
    let actualSearchTerm: string | undefined;

    spectator.component.search.subscribe((otherSearchTerm: string) => {
      actualSearchTerm = otherSearchTerm;
    });

    spectator.typeInElement(searchTerm, byTestId('search-term-input'));




    spectator.dispatchFakeEvent(byTestId('form'), 'submit');

    expect(actualSearchTerm).toBe(searchTerm);
  });
});
```

DISPATCH NGSUBMIT

The spec simulates typing the search term into the search field. Then it simulates an `ngSubmit` event at the `form` element. We use the generic method `spectator.dispatchFakeEvent` for this end.

Spectator offers many more convenient shortcuts for triggering events. The Flickr search Spectator tests just use the most common ones.

-  [PhotoItemComponent: implementation code and the two tests](#)
-  [SearchFormComponent: implementation code and the two tests](#)
-  [Spectator: Events API](#)

Spectator: Summary

Spectator is a mature library that addresses the practical needs of Angular developers. It offers solutions for common Angular testing problems. The examples above presented only a few of Spectator's features.

Test code should be both concise and easy to understand. Spectator provides an expressive, high-level language for writing Angular tests. Spectator makes simple tasks simple without losing any power.

Spectator's success underlines that the standard Angular testing tools are cumbersome and inconsistent. Alternative concepts are both necessary and beneficial.

Once you are familiar with the standard tools, you should try out alternatives like Spectator and ng-mocks. Then decide whether you stick with isolated testing helpers or switch to more comprehensive testing libraries.

 [Spectator project site](#)

 [ng-mocks project site](#)

Testing Services

LEARNING OBJECTIVES

- Writing tests for Services with internal state
- Testing Observables returned by Services
- Verifying HTTP requests and payload processing
- Covering HTTP success and error cases

In an Angular application, Services are responsible for fetching, storing and processing data. Services are singletons, meaning there is only one instance of a Service during runtime. They are fit for central data storage, HTTP and WebSocket communication as well as data validation.

SINGLETON

The single Service instance is shared among Components and other application parts. Therefore, a Service is used when Components that are not parent and child need to communicate with each other and exchange data.

INJECTABLE

“Service” is an umbrella term for any object that serves a specific purpose and is injected as a dependency. Technically, Services have little in common. There are no rules regarding the structure or behavior of a Service.

Typically, Services are classes, but not necessarily. While Modules, Components and Directives are marked with respective decorators – `@Module`, `@Component`, `@Directive` –, Services are marked with the generic `@Injectable`.

RESPONSIBILITIES

So what does a Service do and how do we test it? Services are diverse, but some patterns are widespread.

- Services have public methods that return values.

In the test, we check whether a method returns correct data.

- Services store data. They hold an internal state. We can get or set the state.

In the test, we check whether the state is changed correctly. Since the state should be held in private properties, we cannot access the state directly. We test the state change by calling public methods. We should not peek into the [black box](#).

- Services interact with dependencies. These are often other Services. For example, a Service might send HTTP requests via Angular's `HttpClient`.

In the unit test, we replace the dependency with a fake that returns canned responses.

Testing a Service with internal state

Let us start with testing the [CounterService](#). By now, you should be familiar with the Service. As a reminder, here is the shape including private members:

```
class CounterService {  
    private count: number;  
    private subject: BehaviorSubject<number>;  
    public getCount(): Observable<number> { /* ... */ }  
    public increment(): void { /* ... */ }  
    public decrement(): void { /* ... */ }  
    public reset(newCount: number): void { /* ... */ }  
    private notify(): void { /* ... */ }  
}
```

We need to identify what the Service does, what we need test and how we test it.

WHAT IT DOES

- The Service holds an internal state, namely in the private `count` and `subject` properties. We cannot and should not access these properties in the test.
- For reading the state, the Service has the `getCount` method. It does not return a synchronous value, but an RxJS Observable. We will use `getCount` to get the current count and also to subscribe to changes.

- For changing the state, the Service provides the methods `increment`, `decrement` and `reset`. We will call them and check whether the state has changed accordingly.

Let us write the test code! We create a file called `counter.service.spec.ts` and fill it with test suite boilerplate code:

```
describe('CounterService', () => {  
  /* ... */  
});
```

We already know what the Service does and what needs to be tested. So we add specs for all features:

```
describe('CounterService', () => {  
  it('returns the count', () => { /* ... */ });  
  it('increments the count', () => { /* ... */ });  
  it('decrements the count', () => { /* ... */ });  
  it('resets the count', () => { /* ... */ });  
});
```

INstantiate without `TestBed`

In the *Arrange* phase, each spec needs to create an instance of `CounterService`. The simplest way to do that is:

```
const counterService = new CounterService();
```

This is fine for simple Services without dependencies. For testing Services with dependencies, we will use the `TestBed` later.

We create the fresh instance in a `beforeEach` block since every spec needs it:

```
describe('CounterService', () => {
  let counterService: CounterService;

  beforeEach(() => {
    counterService = new CounterService();
  });

  it('returns the count', () => { /* ... */ });
  it('increments the count', () => { /* ... */ });
  it('decrements the count', () => { /* ... */ });
  it('resets the count', () => { /* ... */ });
});
```

Let us start with writing the spec `it('returns the count', /* ... */)`. It tests the `getCount` method that returns an Observable.

CHANGE VARIABLE VALUE

For testing the Observable, we use the same pattern that we have used for [testing a Component Output](#):

1. We declare a variable `actualCount` that is initially undefined.
2. We subscribe to the Observable. We assign the emitted value to the `actualCount` variable.
3. Finally, outside of the subscriber function, we compare the actual to the expected value.

```
it('returns the count', () => {
  let actualCount: number | undefined;
  counterService.getCount().subscribe((count) => {
    actualCount = count;
  });
  expect(actualCount).toBe(0);
});
```

This works because the Observable is backed by a **BehaviorSubject** that stores the latest value and sends it to new subscribers immediately.

STATE CHANGE

The next spec tests the **increment** method. We call the method and verify that the count state has changed.

As mentioned before, we cannot access the private properties for this purpose. Just like in the spec above, we need to use the public **getCount** method to read the count.

```
it('increments the count', () => {
  counterService.increment();

  let actualCount: number | undefined;
  counterService.getCount().subscribe((count) => {
    actualCount = count;
  });
  expect(actualCount).toBe(1);
});
```

EXPECT CHANGED VALUE

The order here is important: First, we call `increment`, then we subscribe to the Observable to read and verify the changed value. Again, the `BehaviorSubject` emits the current value to new subscribers synchronously.

The two remaining specs work almost the same. We just call the respective methods.

```
it('decrements the count', () => {
  counterService.decrement();

  let actualCount: number | undefined;
  counterService.getCount().subscribe((count) => {
    actualCount = count;
  });
  expect(actualCount).toBe(-1);
});
```

```
it('resets the count', () => {
  const newCount = 123;
  counterService.reset(newCount);

  let actualCount: number | undefined;
  counterService.getCount().subscribe((count) => {
    actualCount = count;
  });
  expect(actualCount).toBe(newCount);
});
```

REPEATING PATTERNS

We quickly notice that the specs are highly repetitive and noisy. In every spec's *Assert* phase, we are using this pattern to inspect the Service state:

```
let actualCount: number | undefined;
counterService.getCount().subscribe((count) => {
  actualCount = count;
});
expect(actualCount).toBe(/* ... */);
```

This is a good candidate for a helper function. Let us call it `expectCount`.

```
function expectCount(count: number): void {
  let actualCount: number | undefined;
  counterService.getCount().subscribe((actualCount2) => {
    actualCount = actualCount2;
  });
  expect(actualCount).toBe(count);
}
```

The pattern has one variable bit, the expected count. That is why the helper function has one parameter.

UNSUBSCRIBE

Now that we have pulled out the code into a central helper function, there is one optimization we should add. The First Rule of RxJS Observables states: "Anyone who subscribes, must unsubscribe as well".

In `expectCount`, we need to get the current count only once. We do not want to create a long-lasting subscription. We are not interested in future changes.

If we call `expectCount` only once per spec, this is not a huge problem. If we wrote a more complex spec with several `expectCount` calls, we would create pointless subscriptions. This is likely to cause confusion when debugging the subscriber function.

In short, we want to fetch the count and then unsubscribe to reduce unwanted subscriptions.

UNSUBSCRIBE MANUALLY

One possible solution is unsubscribing immediately after subscribing. The `subscribe` method returns a `Subscription` with the useful `unsubscribe` method.

```
function expectCount(count: number): void {
  let actualCount: number | undefined;
  counterService
    .getCount()
    .subscribe((actualCount2) => {
      actualCount = actualCount2;
    })
    .unsubscribe();
  expect(actualCount).toBe(count);
}
```

RXJS OPERATOR

A more idiomatic way is to use an RxJS operator that completes the Observable after the first value: [first](#).

```
import { first } from 'rxjs/operators';

function expectCount(count: number): void {
  let actualCount: number | undefined;
  counterService
    .getCount()
    .pipe(first())
    .subscribe((actualCount2) => {
      actualCount = actualCount2;
    });
  expect(actualCount).toBe(count);
}
```

If you are not familiar with this arcane RxJS magic, do not worry. In the simple [CounterService](#) test, unsubscribing is not strictly necessary. But it is a good practice that avoids weird errors when testing more complex Services that make use of Observables.

The complete test suite now looks like this:

```
describe('CounterService', () => {
  let counterService: CounterService;

  function expectCount(count: number): void {
    let actualCount: number | undefined;
    counterService
      .getCount()
      .pipe(first())
```



```
        .subscribe((actualCount2) => {
            actualCount = actualCount2;
        });
    expect(actualCount).toBe(count);
}

beforeEach(() => {
    counterService = new CounterService();
});

it('returns the count', () => {
    expectCount(0);
});

it('increments the count', () => {
    counterService.increment();
    expectCount(1);
});

it('decrements the count', () => {
    counterService.decrement();
    expectCount(-1);
});

it('resets the count', () => {
    const newCount = 123;
    counterService.reset(newCount);
    expectCount(newCount);
});
});
```

 [CounterService: test code](#)

Testing a Service that sends HTTP requests

Services without dependencies, like [CounterService](#), are relatively easy to test. Let us examine a more complex Service with a dependency.

In the [Flickr search](#), the [FlickrService](#) is responsible for searching photos via the Flickr API. It makes an HTTP GET request to www.flickr.com. The server responds with JSON. Here is the full code:

```
@Injectable()
export class FlickrService {
  constructor(private http: HttpClient) {}

  public searchPublicPhotos(searchTerm: string): Observable<Photo[]> {
    return this.http
      .get<FlickrAPIResponse>(
        'https://www.flickr.com/services/rest/',
        {
          params: {
            tags: searchTerm,
            method: 'flickr.photos.search',
            format: 'json',
            nojsoncallback: '1',
            tag_mode: 'all',
            media: 'photos',
            per_page: '15',
            extras: 'tags,date_taken,owner_name,url_q,url_m',
          }
        }
      )
  }
}
```

```

        api_key: 'XYZ',
      },
    }
  )
  .pipe(map((response) => response.photos.photo));
}
}

```

The Service is marked with `@Injectable()` so it takes part in Angular's Dependency Injection. It depends on Angular's standard HTTP library, `HttpClient` from the `@angular/common/http` package. Most Angular applications use `HttpClient` to communicate with HTTP APIs.

There are two ways to test the `FlickrService`: an integration test or a unit test.

REQUESTS AGAINST PRODUCTION

An **integration test** provides the real `HttpClient`. This leads to HTTP requests to the Flickr API when the running the tests. This makes the whole test unreliable.

The network or the web service might be slow or unavailable. Also the Flickr API endpoint returns a different response for each request. It is hard to expect a certain `FlickrService` behavior if the input is unknown.

Requests to third-party production APIs make little sense in a testing environment. If you want to write an integration test for a Service that makes HTTP request, better use a dedicated testing API that returns fixed data. This API can run on the same machine or in the local network.

INTERCEPT REQUESTS

In the case of `FlickrService`, we better write a **unit test**. Angular has a powerful helper for testing code that depends on `HttpClient`: the `HttpClientTestingModule`.

For testing a Service with dependencies, it is tedious to instantiate the Service with `new`. Instead, we use the `TestBed` to set up a testing Module.

In place of the `HttpClient`, we import the `HttpClientTestingModule`.

```
TestBed.configureTestingModule({
  imports: [HttpClientTestingModule],
  providers: [FlickrService],
});
```

The `HttpClientTestingModule` provides a fake implementation of `HttpClient`. It does not actually send out HTTP requests. It merely intercepts them and records them internally.

In the test, we inspect that record of HTTP requests. We respond to pending requests manually with fake data.

FIND, RESPOND, VERIFY

Our test will perform the following steps:

1. Call the method under test that sends HTTP requests
2. Find pending requests
3. Respond to these requests with fake data
4. Check the result of the method call
5. Verify that all requests have been answered

 [Angular guide: Communicating with backend services using HTTP](#)

 [Angular API reference: HttpClient](#)

 [Angular guide: Testing HTTP requests](#)

 [Angular API reference: HttpClientTestingModule](#)

Call the method under test

In the first step, we call the method under test, `searchPublicPhotos`. The search term is simply a fixed string.

```
const searchTerm = 'dragonfly';

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

```

let flickrService: FlickrService;

beforeEach(() => {
  TestBed.configureTestingModule({
    imports: [HttpClientTestingModule],
    providers: [FlickrService],
  });
  flickrService = TestBed.inject(FlickrService);
});

it('searches for public photos', () => {
  flickrService.searchPublicPhotos(searchTerm).subscribe(
    (actualPhotos) => {
      /* ... */
    }
  );
  /* ... */
});
});

```

We subscribe to the Observable returned by `searchPublicPhotos` so the (fake) HTTP request is sent. We will investigate the response, `actualPhotos`, later in step four.

Find pending requests

In the second step, we find the pending request using the [HttpTestingController](#). This class is part of the `HttpClientTestingModule`. We get hold of the instance by calling `TestBed.inject(HttpTestingController)`.

EXPECTONE

The controller has methods to find requests by different criteria. The simplest is `expectOne`. It finds a request matching the given criteria and expects that there is exactly one match.

In our case, we search for a request with a given URL of the Flickr API.

```
const searchTerm = 'dragonfly';
const expectedUrl = `https://www.flickr.com/services/rest/?
tags=${searchTerm}&method=flickr.photos.search&format=json&nojsoncallback=1&
tag_mode=all&media=photos&per_page=15&extras=tags,date_taken,owner_name,url_
q,url_m&api_key=XYZ`;
```

```
describe('FlickrService', () => {
  let flickrService: FlickrService;
  let controller: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [FlickrService],
    });
    flickrService = TestBed.inject(FlickrService);
    controller = TestBed.inject(HttpTestingController);
  });

  it('searches for public photos', () => {
    flickrService.searchPublicPhotos(searchTerm).subscribe(
      (actualPhotos) => {
        /* ... */
      }
    );
  });
});
```

```
);

    const request = controller.expectOne(expectedUrl);
    /* ... */
});
});
```

`expectOne` returns the found request, that is an instance of `TestRequest`. If there is no pending request that matches the URL, `expectOne` throws an exception, failing the spec.

 [Angular API reference: HttpTestingController](#)

 [Angular API reference: TestRequest](#)

Respond with fake data

Now that we have the pending request at hand, we respond to it with an object that mimics the original API response. The Flickr API returns a complex object with an array of photos objects deep within. In the `FlickrService` test, we only care about the payload, the photos array.

The Flickr search repository contains [fake photo objects](#) that are used throughout the tests. For the `FlickrService` test, we import the `photos` array with two fake photo objects.

We use the request's `flush` method to respond with fake data. This simulates a successful "200 OK" server response.


```
request.flush({ photos: { photo: photos } });
```

Check the result of the method call

The spec has proven that `searchPublicPhotos` makes a request to the expected URL. It still needs to prove that the method passes through the desired part of the API response. In particular, it needs to prove that the Observable emits the `photos` array.

We have already subscribed to the Observable:

```
flickrService.searchPublicPhotos(searchTerm).subscribe(  
  (actualPhotos) => {  
    /* ... */  
  }  
);
```

We expect that the Observable emits a photos array that equals to the one from the API response:

```
flickrService.searchPublicPhotos(searchTerm).subscribe(  
  (actualPhotos) => {  
    expect(actualPhotos).toEqual(photos);  
  }  
);
```

This leads to a problem that is known from [testing Outputs](#): If the code under test is broken, the Observable never emits. The `next`

callback with `expect` will not be called. Despite the defect, Jasmine thinks that all is fine.

EXPECT CHANGED VALUE

There are several ways to solve this problem. We have opted for a variable that is `undefined` initially and is assigned a value.

```
let actualPhotos: Photo[] | undefined;
flickrService.searchPublicPhotos(searchTerm).subscribe(
  (otherPhotos) => {
    actualPhotos = otherPhotos;
  }
);
```

```
const request = controller.expectOne(expectedUrl);
// Answer the request so the Observable emits a value.
request.flush({ photos: { photo: photos } });
```

```
// Now verify emitted valued.
expect(actualPhotos).toEqual(photos);
```

The `expect` call is located outside of the `next` callback function to ensure it is definitely called. If the Observable emits no value or a wrong value, the spec fails.

Verify that all requests have been answered

In the last step, we ensure that there are no pending requests left. We expect the method under test to make *one* request to a specific URL. We have found the request with `expectOne` and have answered it with `flush`.

Finally, we call:

```
controller.verify();
```

This fails the test if there are any outstanding requests.

`verify` guarantees that the code under test is not making excess requests. But it also guarantees that your spec checks all requests, for example by inspecting their URLs.

Putting the parts together, the full test suite:

```
const searchTerm = 'dragonfly';
const expectedUrl = `https://www.flickr.com/services/rest/?
tags=${searchTerm}&method=flickr.photos.search&format=json&nojsoncallback=1&
tag_mode=all&media=photos&per_page=15&extras=tags,date_taken,owner_name,url_
q,url_m&api_key=XYZ`;

describe('FlickrService', () => {
  let flickrService: FlickrService;
  let controller: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [FlickrService],
    });
  });
```

```
flickrService = TestBed.inject(FlickrService);
controller = TestBed.inject(HttpTestingController);
});

it('searches for public photos', () => {
  let actualPhotos: Photo[] | undefined;
  flickrService.searchPublicPhotos(searchTerm).subscribe(
    (otherPhotos) => {
      actualPhotos = otherPhotos;
    }
  );

  const request = controller.expectOne(expectedUrl);
  request.flush({ photos: { photo: photos } });
  controller.verify();

  expect(actualPhotos).toEqual(photos);
});
});
```

 [FlickrService: test code](#)

 [Photo spec helper](#)

Testing the error case

Are we done with testing `searchPublicPhotos`? We have tested the success case in which the server returns a `200 OK`. But we have not tested the error case yet!

UNHAPPY PATH

`searchPublicPhotos` passes through the error from `HttpClient`. If the Observable returned by `this.http.get` fails with an error, the Observable returned by `searchPublicPhotos` fails with the same error.

Whether there is custom error handling in the Service or not, the *unhappy path* should be tested.

Let us simulate a “500 Internal Server Error”. Instead of responding to the request with `flush`, we let it fail by calling `error`.

```
const status = 500;
const statusText = 'Internal Server Error';
const errorEvent = new ErrorEvent('API error');
/* ... */
const request = controller.expectOne(expectedUrl);
request.error(
  errorEvent,
  { status, statusText }
);
```

The `TestRequest`'s `error` method expects an `ErrorEvent`, and an optional options object.

`ErrorEvent` is a special type of `Error`. For testing purposes, we create an instance using `new ErrorEvent('...')`. The constructor parameter is a string message that describes the error case.

The second parameter, the options object, allows us to set the HTTP `status` (like `500`), the `statusText` (like `'Internal Server Error'`) and response headers. In the example above, we set `status` and `statusText`.

EXPECT OBSERVABLE ERROR

Now we check that the returned Observable behaves correctly. It must not emit a next value and must not complete. It must fail with an error.

We achieve that by subscribing to `next`, `error` and `complete` events:

```
flickrService.searchPublicPhotos(searchTerm).subscribe(  
  () => {  
    /* next handler must not be called! */  
  },  
  (error) => {  
    /*  
     error handler must be called!  
     Also, we need to inspect the error.  
    */  
  },  
  () => {  
    /* complete handler must not be called! */  
  },  
);
```

FAIL

When the `next` or `complete` handlers are called, the spec must fail immediately. There is a handy global Jasmine function for this purpose: `fail`.

For inspecting the error, we use the same pattern as above, saving the error in a variable in the outer scope.

```
let actualError: HttpResponse | undefined;

flickrService.searchPublicPhotos(searchTerm).subscribe(
  () => {
    fail('next handler must not be called');
  },
  (error) => {
    actualError = error;
  },
  () => {
    fail('complete handler must not be called');
  },
);
```

After answering the request with a server error, we check that the error is passed through. The `error` handler receives an `HttpResponse` object that reflects the `ErrorEvent` as well as the status information.

```
if (!actualError) {
  throw new Error('Error needs to be defined');
}
expect(actualError.error).toBe(errorEvent);
```

```
expect(actualError.status).toBe(status);
expect(actualError.statusText).toBe(statusText);
```

TYPE GUARD

Since `actualError` is defined as `HttpErrorResponse | undefined`, we need to rule out the `undefined` case first before accessing the properties.

`expect(actualError).toBeDefined()` would accomplish that. But the TypeScript compiler does not know that this rules out the `undefined` case. So we need to throw an exception manually.

This is the full spec for the error case:

```
it('passes through search errors', () => {
  const status = 500;
  const statusText = 'Server error';
  const errorEvent = new ErrorEvent('API error');

  let actualError: HttpErrorResponse | undefined;




  flickrService.searchPublicPhotos(searchTerm).subscribe(
    () => {
      fail('next handler must not be called');
    },
    (error) => {
      actualError = error;
    },
    () => {
      fail('complete handler must not be called');
    },
  );
});
```



```
controller.expectOne(expectedUrl).error(  
    errorEvent,  
    { status, statusText }  
);  
  
if (!actualError) {  
    throw new Error('Error needs to be defined');  
}  
expect(actualError.error).toBe(errorEvent);  
expect(actualError.status).toBe(status);  
expect(actualError.statusText).toBe(statusText);  
});
```

This example is deliberately verbose. It shows you how to test all details. It fails fast and provides helpful error messages.

This approach is recommended for Service methods that have a dedicated error handling. For example, a Service might distinguish between successful responses (like “200 OK”), client errors (like “404 Not Found”) and server errors (like “500 Server error”).

-  [FlickrService: implementation and test](#)
-  [Angular API reference: `HttpErrorResponse`](#)
-  [MDN reference: `ErrorEvent`](#)

Alternatives for finding pending requests

We have used `controller.expectOne` to find a request that matches the expected URL. Sometimes it is necessary to specify more criteria, like the method (`GET`, `POST`, etc.), headers or the request body.

`expectOne` has several signatures. We have used the simplest, a string that is interpreted as URL:

```
controller.expectOne('https://www.example.org')
```

To search for a request with a given method and url, pass an object with these properties:

```
controller.expectOne({
  method: 'GET',
  url: 'https://www.example.org'
})
```

If you need to find one request by looking at its details, you can pass a function:

```
controller.expectOne(
  (requestCandidate) =>
    requestCandidate.method === 'GET' &&
    requestCandidate.url === 'https://www.example.org' &&
    requestCandidate.headers.get('Accept') === 'application/json',
);
```

This predicate function is called for each request, decides whether the candidate matches and returns a boolean.

This lets you sift through all requests programmatically and check all criteria. The candidate is an [HttpRequest](#) instance with properties like `method`, `url`, `headers`, `body`, `params`, etc.

There are two possible approaches: Either you use `expectOne` with many criteria, like in the predicate example. If some request detail does not match, `expectOne` throws an exception and fails the test.

Or you use `expectOne` with few criteria, passing `{ method: '...', url: '...' }`. To check the request details, you can still use Jasmine expectations.

`expectOne` returns a `TestRequest` instance. This object only has methods to answer the request, but no direct information about the request. Use the `request` property to access the underlying [HttpRequest](#).

```
// Get the TestRequest.
const request = controller.expectOne({
  method: 'GET',
  url: 'https://www.example.org'
});
// Get the underlying HttpRequest. Yes, this is confusing.
const httpRequest = request.request;
expect(httpRequest.headers.get('Accept')).toBe('application/json');
request.flush({ success: true });
```

This is equivalent to the predicate example above, but gives a more specific error message if the header is incorrect.

MATCH

In addition to `expectOne`, there is the `match` method for finding multiple requests that match certain criteria. It returns an array of requests. If there are no matches, the array is empty, but the spec does not fail. Hence, you need to add Jasmine expectations to check the array and the requests therein.

Assume there is a `CommentService` with a method `postTwoComments`. The code under test makes two requests to the same URL, but with a different body.

```
@Injectable()
class CommentService() {
  constructor(private http: HttpClient) {}
  public postTwoComments(firstComment: string, secondComment: string) {
    return combineLatest([
      this.http.post('/comments/new', { comment: firstComment }),
      this.http.post('/comments/new', { comment: secondComment }),
    ]);
  }
}
```



The spec could contain:

```
const firstComment = 'First comment!';
const secondComment = 'Second comment!';
commentService
```

```
.postTwoComments(firstComment, secondComment)
.subscribe();

const requests = controller.match({
  method: 'POST',
  url: '/comments/new',
});
expect(requests.length).toBe(2);
expect(requests[0].request.body).toEqual({ comment: firstComment });
expect(requests[1].request.body).toEqual({ comment: secondComment });
requests[0].flush({ success: true });
requests[1].flush({ success: true });
```

We verify the number of requests and also the body of each request. If these checks pass, we answer each request.

-  [Angular API reference: HttpRequest](#)
-  [Angular API reference: TestRequest](#)

Testing Services: Summary

All in all, testing Services is easier than testing other Angular application parts. Most Services have a clear purpose and a well-defined public API.

If the Service under test depends on another Service, a unit test needs to fake the dependency. This is probably the hardest part, but takes the same effort as faking Services that are Component dependencies.

PREDEFINED TESTING MODULES

Angular ships with crucial Services that are commonly used in your own Services. Since Angular intends to be testable, Angular also offers tools to replace them with fakes.

We have used the [HttpClientTestingModule](#) for testing a Service that depends on [HttpClient](#). To name another example, there is the [RouterTestingModule](#) for testing Services that depend on [Router](#) and [Location](#).

Testing Pipes

LEARNING OBJECTIVES

- Verifying the output of synchronous, pure Pipes
- Testing asynchronous, impure Pipes that load data from a Service

An Angular Pipe is a special function that is called from a Component template. Its purpose is to transform a value: You pass a value to the Pipe, the Pipe computes a new value and returns it.

The name Pipe originates from the vertical bar “|” that sits between the value and the Pipe’s name. The concept as well as the “|” syntax originate from Unix pipes and Unix shells.

In this example, the value from `user.birthday` is transformed by the `date` Pipe:

```
{{ user.birthday | date }}
```

FORMATTING

Pipes are often used for internationalization, including translation of labels and messages, formatting of dates, times and various numbers. In these cases, the Pipe input value should not be shown to the user. The output value is user-readable.

Examples for built-in Pipes are [DatePipe](#), [CurrencyPipe](#) and [DecimalPipe](#). They format dates, amounts of money and numbers, respectively, according to the localization settings. Another well-known Pipe is the [AsyncPipe](#) which unwraps an Observable or Promise.

PURE PIPES

Most Pipes are *pure*, meaning they merely take a value and compute a new value. They do not have *side effects*: They do not change the input value and they do not change the state of other application parts. Like pure functions, pure Pipes are relatively easy to test.

GreetPipe

Let us study the structure of a Pipe first to find ways to test it. In essence, a Pipe is class with a public [transform](#) method. Here is a simple Pipe that expects a name and greets the user.

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({ name: 'greet' })
export class GreetPipe implements PipeTransform {
  transform(name: string): string {
    return `Hello, ${name}!`;
  }
}
```


In a Component template, we transform a value using the Pipe:

```
{{ 'Julie' | greet }}
```

The `GreetPipe` take the string `'Julie'` and computes a new string, `'Hello, Julie!'`.

SIMPLE VS. COMPLEX SETUP

There are two important ways to test a Pipe:

1. Create an instance of the Pipe class manually. Then call the `transform` method.

This way is fast and straight-forward. It requires minimal setup.

2. Set up a `TestBed`. Render a host Component that uses the Pipe. Then check the text content in the DOM.

This way closely mimics how the Pipe is used in practice. It also tests the name of the Pipe, as declared in the `@Pipe()` decorator.

Both ways allow to test Pipes that depend on Services. Either we provide the original dependencies, writing an integration test. Or we provide fake dependencies, writing a unit test.

GreetPipe test

The `GreetPipe` does not have any dependencies. We opt for the first way and write a unit test that examines the single instance.

First, we create a Jasmine test suite. In a `beforeEach` block, we create an instance of `GreetPipe`. In the specs, we scrutinize the `transform` method.

```
describe('GreetPipe', () => {  
  let greetPipe: GreetPipe;  
  
  beforeEach(() => {  
    greetPipe = new GreetPipe();  
  });  
  
  it('says Hello', () => {  
    expect(greetPipe.transform('Julie')).toBe('Hello, Julie!');  
  });  
});
```

We call the `transform` method with the string `'Julie'` and expect the output `'Hello, Julie!'`.

This is everything that needs to be tested in the `GreetPipe` example. If the `transform` method contains more logic that needs to be tested, we add more specs that call the method with different input.

Testing Pipes with dependencies

Many Pipes depend on local settings, including the user interface language, date and number formatting rules, as well as the selected country, region or currency.

We are introducing and testing the `TranslatePipe`, a complex Pipe with a Service dependency.

 [TranslatePipe: Source code](#)

 [TranslatePipe: Run the app](#)

The example application lets you change the user interface language during runtime. A popular solution for this task is the [ngx-translate](#) library. For the purpose of this guide, we will adopt ngx-translate's proven approach but implement and test the code ourselves.

TranslateService

The current language is stored in the `TranslateService`. This Service also loads and holds the translations for the current language.

The translations are stored in a map of keys and translation strings. For example, the key `greeting` translates to "Hello!" if the

current language is English.

The `TranslateService` looks like this:

```
import { HttpClient } from '@angular/common/http';
import { EventEmitter, Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';
import { map, take } from 'rxjs/operators';

export interface Translations {
  [key: string]: string;
}

@Injectable()
export class TranslateService {
  /** The current language */
  private currentLang = 'en';

  /** Translations for the current language */
  private translations: Translations | null = null;

  /** Emits when the language change */
  public onTranslationChange = new EventEmitter<Translations>();

  constructor(private http: HttpClient) {
    this.loadTranslations(this.currentLang);
  }

  /** Changes the language */
  public use(language: string): void {
    this.currentLang = language;
    this.loadTranslations(language);
  }
}
```

```

/** Translates a key asynchronously */
public get(key: string): Observable<string> {
    if (this.translations) {
        return of(this.translations[key]);
    }
    return this.onTranslationChange.pipe(
        take(1),
        map((translations) => translations[key])
    );
}

/** Loads the translations for the given language */
private loadTranslations(language: string): void {
    this.translations = null;
    this.http
        .get<Translations>(`assets/${language}.json`)
        .subscribe((translations) => {
            this.translations = translations;
            this.onTranslationChange.emit(translations);
        });
}
}

```

This is what the Service provides:

1. **use** method: Set the current language and load the translations as JSON via HTTP.
2. **get** method: Get the translation for a key.
3. **onTranslationChange EventEmitter**: Observing changes on the translations as a result of **use**.

In the example project, the `AppComponent` depends on the `TranslateService`. On creation, the Service loads the English translations. The `AppComponent` renders a select field allowing the user to change the language.

 [TranslateService: implementation code](#)

 [TranslateService: test code](#)

TranslatePipe

To show a translated label, a Component could call the Service's `get` method manually for each translation key. Instead, we introduce the `TranslatePipe` to do the heavy lifting. It lets us write:

```
{{ 'greeting' | translate }}
```

This translates the key `'greeting'`.

Here is the code:

```
import {  
  ChangeDetectorRef,  
  OnDestroy,  
  Pipe,  
  PipeTransform,  
} from '@angular/core';  
import { Subscription } from 'rxjs';
```

```

import { TranslateService } from './translate.service';

@Pipe({
  name: 'translate',
  pure: false,
})
export class TranslatePipe implements PipeTransform, OnDestroy {
  private lastKey: string | null = null;
  private translation: string | null = null;

  private onTranslationChangeSubscription: Subscription;
  private getSubscription: Subscription | null = null;

  constructor(
    private changeDetectorRef: ChangeDetectorRef,
    private translateService: TranslateService
  ) {
    this.onTranslationChangeSubscription =
      this.translateService.onTranslationChange.subscribe(
        () => {
          if (this.lastKey) {
            this.getTranslation(this.lastKey);
          }
        }
      );
  }

  public transform(key: string): string | null {
    if (key !== this.lastKey) {
      this.lastKey = key;
      this.getTranslation(key);
    }
    return this.translation;
  }
}

```

```

private getTranslation(key: string): void {
  this.getSubscription?.unsubscribe();
  this.getSubscription = this.translateService
    .get(key)
    .subscribe((translation) => {
      this.translation = translation;
      this.changeDetectorRef.markForCheck();
      this.getSubscription = null;
    });
}

public ngOnDestroy(): void {
  this.onTranslationChangeSubscription.unsubscribe();
  this.getSubscription?.unsubscribe();
}
}

```

ASYNCR TRANSLATION

The [TranslatePipe](#) is *impure* because the translations are loaded asynchronously. When called the first time, the [transform](#) method cannot return the correct translation synchronously. It calls the [TranslateService](#)'s [get](#) method which returns an [Observable](#).

TRIGGER CHANGE DETECTION

Once the translation is loaded, the [TranslatePipe](#) saves it and notifies the Angular change detector. In particular, it marks the corresponding view as changed by calling [ChangeDetectorRef's markForCheck](#) method.

In turn, Angular re-evaluates every expression that uses the Pipe, like `'greeting' | translate`, and calls the `transform` method again. Finally, `transform` returns the right translation synchronously.

TRANSLATION CHANGES

The same process happens when the user changes the language and new translations are loaded. The Pipe subscribes to `TranslateService`'s `onTranslationChange` and calls the `TranslateService` again to get the new translation.

 [TranslatePipe: implementation code](#)

 [Angular API reference: ChangeDetectorRef](#)

TranslatePipe test

Now let us test the `TranslatePipe`! We can either write a test that integrates the `TranslateService` dependency. Or we write a unit test that replaces the dependency with a fake.

`TranslateService` performs HTTP requests to load the translations. We should avoid these side effects when testing `TranslatePipe`. So let us fake the Service to write a unit test.

```
let translateService: Pick<
  TranslateService, 'onTranslationChange' | 'get'
>;
```

```

/* ... */
translateService = {
  onTranslationChange: new EventEmitter<Translations>(),
  get(key: string): Observable<string> {
    return of(`Translation for ${key}`);
  },
};

```

The fake is a partial implementation of the original. The `TranslatePipe` under test only needs the `onTranslationChange` property and the `get` method. The latter returns a fake translation including the key so we can test that the key was passed correctly.

HOST COMPONENT

Now we need to decide whether to test the Pipe directly or within a host Component. Neither solution is significantly easier or more robust. You will find both solutions in the example project. In this guide, we will discuss the solution with `TestBed` and host Component.

Let us start with the host Component:

```

const key1 = 'key1';
const key2 = 'key2';

@Component({
  template: '{{ key | translate }}',
})
class HostComponent {

```

```
    public key = key1;
}
```

This Component uses the `TranslatePipe` to translate its `key` property. Per default, it is set to `key1`. There is also a second constant `key2` for testing the key change later.

Let us set up the test suite:

```
describe('TranslatePipe: with TestBed and HostComponent', () => {
    let fixture: ComponentFixture<HostComponent>;
    let translateService: Pick<
        TranslateService, 'onTranslationChange' | 'get'
    >;

    beforeEach(async () => {
        translateService = {
            onTranslationChange: new EventEmitter<Translations>(),
            get(key: string): Observable<string> {
                return of(`Translation for ${key}`);
            },
        };

        await TestBed.configureTestingModule({
            declarations: [TranslatePipe, HostComponent],
            providers: [
                { provide: TranslateService, useValue: translateService }
            ],
        }).compileComponents();

        translateService = TestBed.inject(TranslateService);
        fixture = TestBed.createComponent(HostComponent);
    });
```

```
/* ... */  
});
```

In the testing Module, we declare the Pipe under test and the `HostComponent`. For the `TranslateService`, we provide a fake object instead. Just like in a Component test, we create the Component and examine the rendered DOM.

SYNC AND ASYNC TRANSLATION

What needs to be tested? We need to check that `{{ key | translate }}` evaluates to `Translation for key1`. There are two cases that need to be tested though:

1. The translations are already loaded. The Pipe's `transform` method returns the correct translation synchronously. The Observable returned by `TranslateService`'s `get` emits the translation and completes immediately.
2. The translations are pending. `transform` returns `null` (or an outdated translation). The Observable completes at any time later. Then, the change detection is triggered, `transform` is called the second time and returns the correct translation.

In the test, we write specs for both scenarios:

```
it('translates the key, sync service response', /* ... */);  
it('translates the key, async service response', /* ... */);
```

Let us start with the first case. The spec is straight-forward.

```
it('translates the key, sync service response', () => {  
  fixture.detectChanges();  
  expectContent(fixture, 'Translation for key1');  
});
```

Remember, the `TranslateService` fake returns an Observable created with `of`.

```
return of(`Translation for ${key}`);
```

This Observable emits one value and completes immediately. This mimics the case in which the Service has already loaded the translations.

We merely need to call `detectChanges`. Angular calls the Pipe's `transform` method, which calls `TranslateService`'s `get`. The Observable emits the translation right away and `transform` passes it through.

Finally, we use the [expectContent Component helper](#) to test the DOM output.

SIMULATE DELAY

Testing the second case is trickier because the Observable needs to emit asynchronously. There are numerous ways to achieve this. We will use the [RxJS delay operator](#) for simplicity.

At the same time, we are writing an asynchronous spec. That is, Jasmine needs to wait for the Observable and the expectations before the spec is finished.

FAKEASYNC AND TICK

Again, there are several ways how to accomplish this. We are going to use Angular's `fakeAsync` and `tick` functions. We have introduced them when [testing a form with async validators](#).

A quick recap: `fakeAsync` freezes time and prevents asynchronous tasks from being executed. The `tick` function then simulates the passage of time, executing the scheduled tasks.

`fakeAsync` wraps the function passed to `it`:

```
it('translates the key, async service response', fakeAsync(() => {  
  /* ... */  
}));
```

Next, we need to change the `TranslateService`'s `get` method to make it asynchronous.

```
it('translates the key, async service response', fakeAsync(() => {  
  translateService.get = (key) =>  
    of(`Async translation for ${key}`).pipe(delay(100));  
  /* ... */  
}));
```

DELAY OBSERVABLE

We still use `of`, but we delay the output by 100 milliseconds. The exact number does not matter as long as there is *some* delay greater or equal 1.

Now, we can call `detectChanges` for the first time.

```
it('translates the key, async service response', fakeAsync(() => {
  translateService.get = (key) =>
    of(`Async translation for ${key}`).pipe(delay(100));
  fixture.detectChanges();
  /* ... */
}));
```

The Pipe's `transform` method is called for the first time and returns `null` since the Observable does not emit a value immediately.

So we expect that the output is empty:

```
it('translates the key, async service response', fakeAsync(() => {
  translateService.get = (key) =>
    of(`Async translation for ${key}`).pipe(delay(100));
  fixture.detectChanges();
  expectContent(fixture, '');
  /* ... */
}));
```

LET TIME PASS

Here comes the interesting part. We want the Observable to emit a value now. We simulate the passage of 100 milliseconds with `tick(100)`.

```

it('translates the key, async service response', fakeAsync(() => {
  translateService.get = (key) =>
    of(`Async translation for ${key}`).pipe(delay(100));
  fixture.detectChanges();
  expectContent(fixture, '');

  tick(100);
  /* ... */
}));

```

This causes the Observable to emit the translation and complete. The Pipe receives the translation and saves it.

To see a change in the DOM, we start a second change detection. The Pipe's `transform` method is called for the second time and returns the correct translation.

```

it('translates the key, async service response', fakeAsync(() => {
  translateService.get = (key) =>
    of(`Async translation for ${key}`).pipe(delay(100));
  fixture.detectChanges();
  expectContent(fixture, '');

  tick(100);
  fixture.detectChanges();
  expectContent(fixture, 'Async translation for key1');
}));

```

Testing these details may seem pedantic at first. But the logic in `TranslatePipe` exists for a reason.

There are two specs left to write:


```
it('translates a changed key', /* ... */);  
it('updates on translation change', /* ... */);
```

The `TranslatePipe` receives the translation asynchronously and stores both the key and the translation. When Angular calls `transform` with the *same key* again, the Pipe returns the translation synchronously. Since the Pipe is marked as *impure*, Angular does not cache the `transform` result.

DIFFERENT KEY

When `translate` is called with a *different key*, the Pipe needs to fetch the new translation. We simulate this case by changing the `HostComponent`'s `key` property from `key1` to `key2`.

```
it('translates a changed key', () => {  
  fixture.detectChanges();  
  fixture.componentInstance.key = key2;  
  fixture.detectChanges();  
  expectContent(fixture, 'Translation for key2');  
});
```

After a change detection, the DOM contains the updated translation for `key2`.

TRANSLATION CHANGE






Last but not least, the Pipe needs to fetch a new translation from the `TranslateService` when the user changes the language and new translations have been loaded. For this purpose, the Pipe subscribes to the Service's `onTranslationChange` emitter.

Our `TranslateService` fake supports `onTranslationChange` as well, hence we call the `emit` method to simulate a translation change. Before, we let the Service return a different translation in order to see a change in the DOM.

```
it('updates on translation change', () => {  
  fixture.detectChanges();  
  translateService.get = (key) =>  
    of(`New translation for ${key}`);  
  translateService.onTranslationChange.emit({});  
  fixture.detectChanges();  
  expectContent(fixture, 'New translation for key1');  
});
```

We made it! Writing these specs is challenging without doubt.

`TranslateService` and `TranslatePipe` are non-trivial examples with a proven API. The original classes from `ngx-translate` are more powerful. If you look for a robust and flexible solution, you should use the `ngx-translate` library directly.

-  [TranslatePipe: test code](#)
-  [Angular API reference: fakeAsync](#)
-  [Angular API reference: tick](#)
-  [RxJS: delay_operator](#)
-  [ngx-translate](#)

Testing Directives

LEARNING OBJECTIVES

- Testing the effect of an Attribute Directive
- Testing complex Structural Directives with Inputs and templates
- Providing a host Component for testing Attribute and Structural Directives

Angular beginners quickly encounter four core concepts: Modules, Components, Services and Pipes. A lesser known core concept are Directives. Without knowing, even beginners are using Directives, because Directives are everywhere.

In Angular, there are three types of Directives:

1. A **Component** is a Directive with a template. A Component typically uses an element type selector, like `app-counter`. Angular then looks for `app-counter` elements and renders the Component template into these host elements.
2. An **Attribute Directive** adds logic to an existing host element in the DOM. Examples for built-in Attribute Directives are `NgClass` and `NgStyle`.
3. A **Structural Directive** alters the structure of the DOM, meaning it adds and removes elements programmatically. Examples for built-in Structural Directives are `NgIf`, `NgFor` and `NgSwitch`.

We have already tested Components. We have yet to test the two other types of Directives.

Testing Attribute Directives

The name Attribute Directive comes from the attribute selector, for example `[ngModel]`. An Attribute Directive does not have a template and cannot alter the DOM structure.

We have already mentioned the built-in Attribute Directives `NgClass` and `NgStyle`. In addition, both Template-driven and Reactive Forms rely heavily on Attribute Directives: `NgForm`, `NgModel`, `FormGroupDirective`, `FormControlName`, etc.

STYLING LOGIC

Attributes Directives are often used for changing the style of an element, either directly with inline styles or indirectly with classes.

Most styling logic can be implemented using CSS alone, no JavaScript code is necessary. But sometimes JavaScript is required to set inline styles or add classes programmatically.

ThresholdWarningDirective

None of our [example applications](#) contain an Attribute Directive, so we are introducing and testing the **ThresholdWarningDirective**.

This Directive applies to `<input type="number">` elements. It toggles a class if the picked number exceeds a given threshold. If the number is higher than the threshold, the field should be marked visually.

Note that numbers above the threshold are valid input. The **ThresholdWarningDirective** does not add a form control validator. We merely want to warn the user so they check the input twice.

 [ThresholdWarningDirective: Source code](#)

 [ThresholdWarningDirective: Run the app](#)

Enter a number greater than 10 to see the effect.

This is the Directive's code:

```
import {
  Directive, ElementRef, HostBinding, HostListener, Input
} from '@angular/core';

@Directive({
  selector: '[appThresholdWarning]',
})
export class ThresholdWarningDirective {
  @Input()
```

```

public appThresholdWarning: number | null = null;

@HostBinding('class.overThreshold')
public overThreshold = false;

@HostListener('input')
public inputHandler(): void {
    this.overThreshold =
        this.appThresholdWarning !== null &&
        this.elementRef.nativeElement.valueAsNumber >
this.appThresholdWarning;
}

constructor(private elementRef: ElementRef<HTMLInputElement>) {}
}

```

This is how we apply the Directive to an element:

```
<input type="number" [appThresholdWarning]="10" />
```

This means: If the user enters a number that is greater than 10, mark the field with a visual warning.

One bit is missing: the styles for the visual warning.

```

input[type='number'].overThreshold {
    background-color: #fe9;
}

```

Before we write the test for the Directive, let us walk through the implementation parts.

INPUT OF THE SAME NAME

The `ThresholdWarningDirective` is applied with an attribute binding `[appThresholdWarning]="..."`. It receives the attribute value as an Input of the same name. This is how the threshold is configured.

```
@Input()  
public appThresholdWarning: number | null = null;
```

INPUT EVENT

Using `HostListener`, the Directive listens for `input` event on the host element. When the user changes the field value, the `inputHandler` method is called.

The `inputHandler` gets the field value and checks whether it is over the threshold. The result is stored in the `overThreshold` boolean property.

```
@HostListener('input')  
public inputHandler(): void {  
    this.overThreshold =  
        this.appThresholdWarning !== null &&  
        this.elementRef.nativeElement.valueAsNumber > this.appThresholdWarning;  
}
```

READ VALUE

To access the host element, we use the `ElementRef` dependency. `ElementRef` is a wrapper around the host element's DOM node.

`this.elementRef.nativeElement` yields the `input` element's DOM node. `valueAsNumber` contains the input value as a number.

TOGGLE CLASS

Last but not least, the `overThreshold` property is bound to a class of the same name using `HostBinding`. This is how the class is toggled.

```
@HostBinding('class.overThreshold')
public overThreshold = false;
```

ThresholdWarningDirective test

Now that we understand what is going on, we need to replicate the workflow in our test.

HOST COMPONENT

First of all, Attribute and Structural Directives need an existing host element they are applied to. When testing these Directives, we use a **host Component** that renders the host element. For example, the `ThresholdWarningDirective` needs an `<input type="number">` host element.

```
@Component({
  template: `
    <input type="number"
      [appThresholdWarning]="10" />
```



```
})  
class HostComponent {}
```

We are going to render this Component. We need a standard [Component test setup](#) using the TestBed.

```
describe('ThresholdWarningDirective', () => {  
  let fixture: ComponentFixture<HostComponent>;  
  
  beforeEach(async () => {  
    await TestBed.configureTestingModule({  
      declarations: [ThresholdWarningDirective, HostComponent],  
    }).compileComponents();  
  
    fixture = TestBed.createComponent(HostComponent);  
    fixture.detectChanges();  
  });  
  
  /* ... */  
});
```

When configuring the testing Module, we declare both the Directive under test and the host Component. Just like in a Component test, we render the Component and obtain a [ComponentFixture](#).

FIND INPUT ELEMENT

In the following specs, we need to access the input element. We use the standard approach: a [data-testid](#) attribute and the [findEl testing helper](#).

For convenience, we pick the input element in the `beforeEach` block. We save it in a shared variable named `input`.

```
@Component({
  template: `
    <input type="number"
      [appThresholdWarning]="10"
      data-testid="input" />
  `,
})
class HostComponent {}

describe('ThresholdWarningDirective', () => {
  let fixture: ComponentFixture<HostComponent>;
  let input: HTMLInputElement;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ThresholdWarningDirective, HostComponent],
    }).compileComponents();

    fixture = TestBed.createComponent(HostComponent);
    fixture.detectChanges();

    input = findEl(fixture, 'input').nativeElement;
  });

  /* ... */
});
```

CHECK CLASS

The first spec ensures that the Directive does nothing when the user has not touched the input. Using the element's `classList`, we expect the class `overThreshold` to be absent.

```
it('does not set the class initially', () => {  
  expect(input.classList.contains('overThreshold')).toBe(false);  
});
```

The next spec enters a number over the threshold. To simulate the user input, we use our handy testing helper `setFieldValue`. Then, the spec expects the class to be present.

```
it('adds the class if the number is over the threshold', () => {  
  setFieldValue(fixture, 'input', '11');  
  fixture.detectChanges();  
  expect(input.classList.contains('overThreshold')).toBe(true);  
});
```

`setFieldValue` triggers a fake `input` event. This triggers the Directive's event handler. `11` is greater than the threshold `10`, so the class is added. We still need to call `detectChanges` so the DOM is updated.

The last spec makes sure that the threshold is still considered as a safe value. No warning should be shown.

```
it('removes the class if the number is at the threshold', () => {  
  setFieldValue(fixture, 'input', '10');  
  fixture.detectChanges();  
  expect(input.classList.contains('overThreshold')).toBe(false);  
});
```

This is it! Testing the `ThresholdWarningDirective` is like testing a Component. The difference is that the Component serves as a host for the Directive.

The full spec for the `ThresholdWarningDirective` looks like this:

```
import { Component } from '@angular/core';
import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { findEl, setFieldValue } from './spec-helpers/element.spec-helper';
import { ThresholdWarningDirective } from './threshold-warning.directive';

@Component({
  template: `
    <input type="number"
      [appThresholdWarning]="10"
      data-testid="input" />
  `
})
class HostComponent {}

describe('ThresholdWarningDirective', () => {
  let fixture: ComponentFixture<HostComponent>;
  let input: HTMLInputElement;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ThresholdWarningDirective, HostComponent],
    }).compileComponents();

    fixture = TestBed.createComponent(HostComponent);
    fixture.detectChanges();
    input = findEl(fixture, 'input').nativeElement;
  });
});
```

```
});

it('does not set the class initially', () => {
  expect(input.classList.contains('overThreshold')).toBe(false);
});

it('adds the class if the number is over the threshold', () => {
  setFieldValue(fixture, 'input', '11');
  fixture.detectChanges();
  expect(input.classList.contains('overThreshold')).toBe(true);
});

it('removes the class if the number is at the threshold', () => {
  setFieldValue(fixture, 'input', '10');
  fixture.detectChanges();
  expect(input.classList.contains('overThreshold')).toBe(false);
});
});
```

 [ThresholdWarningDirective: implementation code](#)

 [ThresholdWarningDirective: test code](#)

Testing Structural Directives

A Structural Directive does not have a template like a Component, but operates on an internal `ng-template`. The Directive renders the template into the DOM programmatically, passing context data to the template.

RENDER TEMPLATE PROGRAMMATICALLY

The prime examples demonstrate what Structural Directives are capable of:

- The `NgIf` Directive decides whether the template is rendered or not.
- The `NgFor` Directive walks over a list of items and renders the template repeatedly for each item.

A Structural Directive uses an attribute selector, like `[ngIf]`. The attribute is applied to a host element with the special asterisk syntax, for example `*ngIf`. Internally, this is translated to `<ng-template [ngIf]="..."> ... </ng-template>`.

This guide assumes that you roughly understand how Structural Directives work and how the microsyntax translates to Directive Inputs. Please refer to the [comprehensive official guide on Structural Directives](#).

PaginateDirective

We are introducing and testing the `PaginateDirective`, a complex Structural Directive.

NGFOR WITH PAGINATION

`PaginateDirective` works similar to `NgFor`, but does not render all list items at once. It spreads the items over pages, usually called **pagination**.

Per default, only ten items are rendered. The user can turn the pages by clicking on “next” or “previous” buttons.

 [PaginateDirective: Source code](#)

 [PaginateDirective: Run the app](#)

Before writing the test, we need to understand the outer structure of `PaginateDirective` first.

The simplest use of the Directive looks like this:

```
<ul>
  <li *appPaginate="let item of items">
    {{ item }}
  </li>
</ul>
```

This is similar to the `NgFor` directive. Assuming that `items` is an array of numbers (`[1, 2, 3, ...]`), the example above renders the first 10 numbers in the array.

The asterisk syntax `*appPaginate` and the so-called *microsyntax* `let item of items` is *syntactic sugar*. This is a shorter and nicer way to write something complex. Internally, Angular translates the code to the following:

```
<ng-template appPaginate let-item [appPaginateOf]="items">
  <li>
    {{ item }}
  </li>
</ng-template>
```

There is an `ng-template` with an attribute `appPaginate` and an attribute binding `appPaginateOf`. Also there is a template input variable called `item`.

RENDER TEMPLATE FOR EACH ITEM

As mentioned, a Structural Directive does not have its own template, but operates on an `ng-template` and renders it programmatically. Our `PaginateDirective` works with the `ng-template` shown above. The Directive renders the template for each item on the current page.

Now that we have seen Angular's internal representation, we can understand the structure of the `PaginateDirective` class:

```
@Directive({
  selector: '[appPaginate]',
})
export class PaginateDirective<T> implements OnChanges {
  @Input()
  public appPaginateOf: T[] = [];

  /* ... */
}
```


The Directive uses the `[appPaginate]` attribute selector and has an Input called `appPaginateOf`. By writing the microsyntax `*appPaginate="let item of items"`, we actually set the `appPaginateOf` Input to the value `items`.

DIRECTIVE INPUTS

The `PaginateDirective` has a configuration option named `perPage`. It specifies how many items are visible per page.

Per default, there are ten items on a page. To change it, we set `perPage: ...` in the microsyntax:

```
<ul>
  <li *appPaginate="let item of items; perPage: 5">
    {{ item }}
  </li>
</ul>
```

This translates to:

```
<ng-template
  appPaginate
  let-item
  [appPaginateOf]="items"
  [appPaginatePerPage]="5">
  <li>
    {{ item }}
  </li>
</ng-template>
```

`perPage` translates to an Input named `appPaginatePerPage` in the Directive's code:

```
@Directive({
  selector: '[appPaginate]',
})
export class PaginateDirective<T> implements OnChanges {
  @Input()
  public appPaginateOf: T[] = [];

  @Input()
  public appPaginatePerPage = 10;

  /* ... */
}
```

This is how built-in Structural Directives like `NgIf` and `NgFor` work as well.

Now it gets more complicated. Since we want to paginate the items, we need user controls to turn the pages – in addition to rendering the items.

Again, a Structural Directive lacks a template. `PaginateDirective` cannot render the “next” and “previous” buttons itself. And to remain flexible, it should not render specific markup. The Component that uses the Directive should decide how the controls look.

PASS ANOTHER TEMPLATE

We solve this by passing the controls as a template to the Directive. In particular, we pass a reference to a separate `ng-template`. This will be the second template the Directive operates on.

This is how the controls template could look like:

```
<ng-template
  #controls
  let-previousPage="previousPage"
  let-page="page"
  let-pages="pages"
  let-nextPage="nextPage"
>
  <button (click)="previousPage()">
    Previous page
  </button>
  {{ page }} / {{ pages }}
  <button (click)="nextPage()">
    Next page
  </button>
</ng-template>
```

`#controls` sets a [template reference variable](#). This means we can further reference the template by the name `controls`.

CONTEXT OBJECT

The Directive renders the controls template with a *context* object that implements the following TypeScript interface:

```
interface ControlsContext {  
    page: number;  
    pages: number;  
    previousPage(): void;  
    nextPage(): void;  
}
```

`page` is the current page number. `pages` is the total number of pages. `previousPage` and `nextPage` are functions for turning the pages.

USE PROPERTIES FROM CONTEXT

The `ng-template` takes these properties from the context and saves them in local variables of the same name:

```
let-previousPage="previousPage"  
let-page="page"  
let-pages="pages"  
let-nextPage="nextPage"
```

This means: Take the context property `previousPage` and make it available in the template under the name `previousPage`. And so on.

The content of the template is rather simple. It renders two buttons for the page turning, using the functions as click handlers. It outputs the current page number and the number of total pages.

```
<button (click)="previousPage()">
  Previous page
</button>
{{ page }} / {{ pages }}
<button (click)="nextPage()">
  Next page
</button>
```

Last but not least, we pass the template to the **PaginateDirective** using the microsyntax:

```
<ul>
  <li *appPaginate="let item of items; perPage: 5; controls: controls">
    {{ item }}
  </li>
</ul>
```

This translates to:

```
<ng-template
  appPaginate
  let-item
  [appPaginateOf]="items"
  [appPaginatePerPage]="5"
  [appPaginateControls]="controls">
  <li>
    {{ item }}
  </li>
</ng-template>
```

controls: ... in the microsyntax translates to an Input named **appPaginateControls**. This concludes the Directive's outer structure:

```

@Directive({
  selector: '[appPaginate]',
})
export class PaginateDirective<T> implements OnChanges {
  @Input()
  public appPaginateOf: T[] = [];

  @Input()
  public appPaginatePerPage = 10;

  @Input()
  public appPaginateControls?: TemplateRef<ControlsContext>;

  /* ... */
}

```

The inner workings of the `PaginateDirective` are not relevant for testing, so we will not discuss them in detail here. Please refer to the Angular guide [Write a structural directive](#) for a general explanation.

 [PaginateDirective: implementation code](#)

PaginateDirective test

We have explored all features of `PaginateDirective` and are now ready to test them!

HOST COMPONENT

First, we need a host Component that applies the Structural Directive under test. We let it render a list of ten numbers, three numbers on each page.

```
const items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

@Component({
  template: `
    <ul>
      <li
        *appPaginate="let item of items; perPage: 3"
        data-testid="item"
      >
        {{ item }}
      </li>
    </ul>
  `,
})
class HostComponent {
  public items = items;
}
```

CONTROLS TEMPLATE

Since we also want to test the custom controls feature, we need to pass a controls template. We will use the simple controls discussed above.

```
const items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

@Component({
  template: `
    <ul>
```

```

    <li
      *appPaginate="let item of items; perPage: 3; controls: controls"
      data-testid="item"
    >
      {{ item }}
    </li>
</ul>
<ng-template
  #controls
  let-previousPage="previousPage"
  let-page="page"
  let-pages="pages"
  let-nextPage="nextPage"
>
  <button
    (click)="previousPage()"
    data-testid="previousPage">
    Previous page
  </button>
  <span data-testid="page">{{ page }}</span>
  /
  <span data-testid="pages">{{ pages }}</span>
  <button
    (click)="nextPage()"
    data-testid="nextPage">
    Next page
  </button>
</ng-template>
` ,
})
class HostComponent {
  public items = items;
}

```


The template code already contains `data-testid` attributes. This is how we find and examine the elements in the test (see [Querying the DOM with test ids](#)).

This is quite a setup, but after all, we want to test the `PaginateDirective` under realistic conditions.

The test suite configures a testing Module, declares both the `HostComponent` and the `PaginateDirective` and renders the `HostComponent`:

```
describe('PaginateDirective', () => {
  let fixture: ComponentFixture<HostComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [PaginateDirective, HostComponent],
    }).compileComponents();

    fixture = TestBed.createComponent(HostComponent);
    fixture.detectChanges();
  });

  /* ... */
});
```

This is a standard Component test setup – nothing special yet.

The first spec verifies that the Directive renders the items on the first page, in our case the numbers 1, 2 and 3.

We have marked the item element with `data-testid="item"`. We use the [findEls test helper](#) to find all elements with said test id.

EXPECT RENDERED ITEMS

We expect to find three items. Then we examine the text content of each item and expect that it matches the item in the number list, respectively.

```
it('renders the items of the first page', () => {
  const els = findEls(fixture, 'item');
  expect(els.length).toBe(3);

  expect(els[0].nativeElement.textContent.trim()).toBe('1');
  expect(els[1].nativeElement.textContent.trim()).toBe('2');
  expect(els[2].nativeElement.textContent.trim()).toBe('3');
});
```

Already, the expectations are repetitive and hard to read. So we introduce a little helper function.

```
function expectItems(
  elements: DebugElement[],
  expectedItems: number[],
): void {
  elements.forEach((element, index) => {
    const actualText = element.nativeElement.textContent.trim();
    expect(actualText).toBe(String(expectedItems[index]));
  });
}
```

This lets us rewrite the spec so it is easier to grasp:

```
it('renders the items of the first page', () => {
  const els = findEls(fixture, 'item');
  expect(els.length).toBe(3);
  expectItems(els, [1, 2, 3]);
});
```

CHECK CONTROLS

The next spec proves that the controls template is rendered passing the current page and the total number of pages.

The elements have have a `data-testid="page"` and `data-testid="pages"`, respectively. We use the [expectText testing helper](#) to check their text content.

```
it('renders the current page and total pages', () => {
  expectText(fixture, 'page', '1');
  expectText(fixture, 'pages', '4');
});
```

Three more specs deal with the controls for turning pages. Let us start with the “next” button.

```
it('shows the next page', () => {
  click(fixture, 'nextPage');
  fixture.detectChanges();

  const els = findEls(fixture, 'item');
  expect(els.length).toBe(3);
  expectItems(els, [4, 5, 6]);
});
```

TURN PAGES

We simulate a click on the “next” button using the `click` testing helper. Then we start Angular’s change detection so the Component together with the Directive are re-rendered.

Finally, we verify that the Directive has rendered the next three items, the numbers 4, 5 and 6.

The spec for the “previous” button looks similar. First, we jump to the second page, then back to the first page.

```
it('shows the previous page', () => {
  click(fixture, 'nextPage');
  click(fixture, 'previousPage');
  fixture.detectChanges();

  const els = findEls(fixture, 'item');
  expect(els.length).toBe(3);
  expectItems(els, [1, 2, 3]);
});
```

STRESS TEST

We have now covered the Directive’s important behavior. Time for testing edge cases! Does the Directive behave correctly if we click on the “previous” button on the first page and the “next” button on the last page?

```
it('checks the pages bounds', () => {
  click(fixture, 'nextPage'); // -> 2
  click(fixture, 'nextPage'); // -> 3
  click(fixture, 'nextPage'); // -> 4
```

```

    click(fixture, 'nextPage'); // -> 4
    click(fixture, 'previousPage'); // -> 3
    click(fixture, 'previousPage'); // -> 2
    click(fixture, 'previousPage'); // -> 1
    click(fixture, 'previousPage'); // -> 1
    fixture.detectChanges();

    // Expect that the first page is visible again
    const els = findEls(fixture, 'item');
    expect(els.length).toBe(3);
    expectItems(els, [1, 2, 3]);
  });

```

By clicking on the buttons, we jump forward to the last page and backward to the first page again.

This is it! Here is the full test code:

```

import { Component, DebugElement } from '@angular/core';
import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import {
  findEls,
  expectText,
  click,
} from './spec-helpers/element.spec-helper';
import { PaginateDirective } from './paginate.directive';

const items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

@Component({
  template: `
    <ul>
      <li

```

```

        *appPaginate="let item of items; perPage: 3; controls: controls"
        data-testid="item"
    >
        {{ item }}
    </li>
</ul>
<ng-template
    #controls
    let-previousPage="previousPage"
    let-page="page"
    let-pages="pages"
    let-nextPage="nextPage"
>
    <button (click)="previousPage()" data-testid="previousPage">
        Previous page
    </button>
    <span data-testid="page">{{ page }}</span>
    /
    <span data-testid="pages">{{ pages }}</span>
    <button (click)="nextPage()" data-testid="nextPage">
        Next page
    </button>
</ng-template>
` ,
}))
class HostComponent {
    public items = items;
}

function expectItems(
    elements: DebugElement[],
    expectedItems: number[],
): void {
    elements.forEach((element, index) => {

```

```
    const actualText = element.nativeElement.textContent.trim();
    expect(actualText).toBe(String(expectedItems[index]));
  });
}
```

```
describe('PaginateDirective', () => {
  let fixture: ComponentFixture<HostComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [PaginateDirective, HostComponent],
    }).compileComponents();

    fixture = TestBed.createComponent(HostComponent);
    fixture.detectChanges();
  });

  it('renders the items of the first page', () => {
    const els = findEls(fixture, 'item');
    expect(els.length).toBe(3);
    expectItems(els, [1, 2, 3]);
  });

  it('renders the current page and total pages', () => {
    expectText(fixture, 'page', '1');
    expectText(fixture, 'pages', '4');
  });

  it('shows the next page', () => {
    click(fixture, 'nextPage');
    fixture.detectChanges();

    const els = findEls(fixture, 'item');
    expect(els.length).toBe(3);
  });
});
```

```

    expectItems(els, [4, 5, 6]);
  });

  it('shows the previous page', () => {
    click(fixture, 'nextPage');
    click(fixture, 'previousPage');
    fixture.detectChanges();

    const els = findEls(fixture, 'item');
    expect(els.length).toBe(3);
    expectItems(els, [1, 2, 3]);
  });


  it('checks the pages bounds', () => {
    click(fixture, 'nextPage'); // -> 2
    click(fixture, 'nextPage'); // -> 3
    click(fixture, 'nextPage'); // -> 4
    click(fixture, 'previousPage'); // -> 3
    click(fixture, 'previousPage'); // -> 2
    click(fixture, 'previousPage'); // -> 1
    fixture.detectChanges();

    // Expect that the first page is visible again
    const els = findEls(fixture, 'item');
    expect(els.length).toBe(3);
    expectItems(els, [1, 2, 3]);
  });
});

```

PaginateDirective is a complex Structural Directive that requires a complex test setup. Once we have created a suitable host Component, we can test it using our familiar testing helpers.

The fact that the logic resides in the Directive is not relevant for the specs.

 [PaginateDirective: implementation code](#)

 [PaginateDirective: test code](#)

Testing Modules

LEARNING OBJECTIVES

- Deciding whether and how to test Angular Modules
- Writing smoke tests to catch Module errors early

Modules are central parts of Angular applications. Often they contain important setup code. Yet they are hard to test since there is no typical logic, only sophisticated configuration.

ONLY METADATA

Angular Modules are classes, but most of the time, the class itself is empty. The essence lies in the metadata set with `@NgModule({ ... })`.

We could sneak into the metadata and check whether certain Services are provided, whether third-party Modules are imported and whether Components are exported.

But such a test would simply **mirror the implementation**. Code duplication does not give you more confidence, it only increases the cost of change.

Should we write tests for Modules at all? If there is a reference error in the Module, the compilation step (`ng build`) fails before the automated tests scrutinize the build. “Failing fast” is good from a software quality perspective.

SMOKE TEST

There are certain Module errors that only surface during runtime. These can be caught with a *smoke test*. Given this Module:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ExampleComponent } from './example.component';

@NgModule({
  declarations: [ExampleComponent],
  imports: [CommonModule],
})
export class FeatureModule {}
```

We write this smoke test:

```
import { TestBed } from '@angular/core/testing';
import { FeatureModule } from './example.module';

describe('FeatureModule', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [FeatureModule],
    });
  });

  it('initializes', () => {
    const module = TestBed.inject(FeatureModule);
    expect(module).toBeTruthy();
  });
});
```

The integration test uses the `TestBed` to import the Module under test. It verifies that no error occurs when importing the Module.

Measuring code coverage

LEARNING OBJECTIVES

- Understanding the code coverage metric
- Generating and inspecting a code coverage report
- Finding code that is not yet covered by automated tests
- Enforcing code coverage thresholds and improving code coverage

Code coverage, also called test coverage, tells you which parts of your code are executed by running the unit and integration tests. Code coverage is typically expressed as percent values, for example, 79% statements, 53% branches, 74% functions, 78% lines.

Statements are, broadly speaking, control structures like `if` and `for` as well as expressions separated by semicolon. Branches refers to the two branches of `if (...) {...} else {...}` and `... ? ... : ...` conditions. Functions and lines are self-explanatory.

Coverage report

In Angular's Karma and Jasmine setup, [Istanbul](#) is used for measuring test coverage. Istanbul rewrites the code under test to record whether a statement, branch, function and line was called. Then it produces a comprehensive test report.

To activate Istanbul when running the tests, add the `--code-coverage` parameter:

```
ng test --code-coverage
```

After the tests have completed, Istanbul saves the report in the `coverage` directory located in the Angular project directory.

The report is a bunch of HTML files you can open with a browser. Start by opening `coverage/index.html` in the browser of your choice.

The report for the Flickr search example looks like this:

All files

100% Statements 97/97 50% Branches 1/2 100% Functions 25/25 100% Lines 91/91

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File		Statements	Branches	Functions	Lines
src	<div></div>	100%	3/3	100%	0/0
src/app/actions	<div></div>	100%	3/3	100%	0/0
src/app/components/flickr-search	<div></div>	100%	14/14	100%	0/0
src/app/components/flickr-search-ng	<div></div>	100%	10/10	100%	0/0
src/app/components/full-photo	<div></div>	100%	3/3	100%	0/0
src/app/components/photo-item	<div></div>	100%	8/8	50%	1/2
src/app/components/photo-list	<div></div>	100%	7/7	100%	0/0
src/app/components/search-form	<div></div>	100%	6/6	100%	0/0
src/app/effects	<div></div>	100%	10/10	100%	0/0
src/app/reducers	<div></div>	100%	9/9	100%	0/0
src/app/selectors	<div></div>	100%	8/8	100%	0/0
src/app/services	<div></div>	100%	6/6	100%	0/0
src/app/spec-helpers	<div></div>	100%	10/10	100%	0/0

Istanbul creates an HTML page for every directory and every file. By following the links, you can descend to reports for the individual files.

For example, the coverage report for [photo-item.component.ts](#) of the Flickr search:

All files / src/app/components/photo-item photo-item.component.ts

100% Statements 8/8 50% Branches 1/2 100% Functions 1/1 100% Lines 8/8

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1  import { Component, EventEmitter, Input, Output } from '@angular/core';
2
3  import { Photo } from '../models/photo';
4
5  @Component({
6    selector: 'app-photo-item',
7    templateUrl: './photo-item.component.html',
8    styleUrls: ['./photo-item.component.css'],
9  })
10 1x export class PhotoItemComponent {
11    @Input()
12    6x public photo: Photo | null = null;
13
14    @Output()
15    6x public focusPhoto = new EventEmitter<Photo>();
16
17    1x public handleClick(event: MouseEvent): void {
18      2x   event.preventDefault();
19      2x   E if (this.photo) {
20      2x     this.focusPhoto.emit(this.photo);
21   }
22 }
23 1x }
24
```

The report renders the source code annotated with the information how many times a line was called. In the example

above, the code is fully covered except for an irrelevant `else` branch, marked with an “E”.

The spec `it('focusses a photo on click', () => {...})` clicks on the photo item to test whether the `focusPhoto` Output emits. Let us disable the spec on purpose to see the impact.

All files / src/app/components/photo-item photo-item.component.ts

62.5% Statements 5/8 0% Branches 0/2 0% Functions 0/1 62.5% Lines 5/8

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1  import { Component, EventEmitter, Input, Output } from '@angular/core';
2
3  import { Photo } from '../../models/photo';
4
5  @Component({
6    selector: 'app-photo-item',
7    templateUrl: './photo-item.component.html',
8    styleUrls: ['./photo-item.component.css'],
9  })
10 1x export class PhotoItemComponent {
11    @Input()
12    4x public photo: Photo | null = null;
13
14    @Output()
15    4x public focusPhoto = new EventEmitter<Photo>();
16
17    1x public handleClick(event: MouseEvent): void {
18      event.preventDefault();
19      if (this.photo) {
20        this.focusPhoto.emit(this.photo);
21      }
22    }
23    1x }
24
```

You can tell from the coverage report above that the `handleClick` method is never called. A key Component behavior is untested.

How to use the coverage report

Now that we know how to generate the report, what should we do with it?

In the chapter [The right amount of testing](#), we have identified code coverage as a useful, but flawed metric. As a quantitative measure, code coverage cannot assess the quality of your tests.

Software testing is not a competition. We should not try to reach a particular score just for the sake of it. For what purpose are we measuring code coverage then?

FIND UNCOVERED CODE

The coverage report is a valuable tool you should use while writing tests. It *reveals code behavior that is not yet tested*. The report not only guides your testing, it also deepens your understanding of how your tests work.

Whatever your current coverage score is, use the reporting to monitor and improve your testing practice. As described in [Tailoring your testing approach](#), testing should be part of the development routine. New features should include tests, bug fixes should include a test as proof and to prevent regressions.

IMPROVE COVERAGE

Writing new code and changing existing code should not lower the coverage score, but gradually increase it. This means if your existing tests cover 75% lines of code, new code needs to be at least 75% covered. Otherwise the score slowly deteriorates.

It is common practice to run the unit and integration tests in a continuous integration environment and measure the code coverage. To enforce a certain coverage score and to prevent decline, you can configure **thresholds** in the [Karma configuration](#).

In `karma.conf.js`, you can add global thresholds for statements, branches, functions and lines.

```
coverageReporter: {  
  /* ... */  
  check: {  
    global: {  
      statements: 75,  
      branches: 75,  
      functions: 75,  
      lines: 75,  
    },  
  },  
},
```

In the configuration above, all values are set to 75%. If the coverage drops below that number, the test execution fails even if all specs succeeded.

RAISE THE BAR

When new code is added to the project with a test coverage better than average, you can raise the thresholds slowly but steadily – for example, from 75 to 75.1, 75.2, 75.3 and so on. Soon these small improvements add up.

Test coverage should not be a pointless competition that puts developers under pressure and shames those that do not meet an arbitrary mark. Measuring coverage is a tool you should use for your benefit. Keep in mind that writing meaningful, spot-on tests does not necessarily increase the coverage score.

For beginners and experts alike, the coverage report helps to set up, debug and improve their tests. For experienced developers, the score helps to keep up a steady testing practice.

 [karma-coverage Configuration](#)

End-to-end testing

LEARNING OBJECTIVES

- Writing valuable tests that cover all parts of your application
- Understanding different approaches to end-to-end testing
- Setting up Cypress for testing your Angular project
- Orchestrating a web browser to load and inspect your application
- Intercepting API calls to return fixed data

We have successfully written unit and integration tests using Karma, Jasmine and Angular's own testing tools. These precise tests give confidence that a single application part – like a Component or Service - or a group of connected parts work as intended.

USER PERSPECTIVE

Karma and Jasmine tests take a technical perspective. They focus on the front-end JavaScript code alone and run it in a controlled and isolated test environment. What is really important though is whether the whole application works **for the user**.

The most effective and reliable way to ensure a working application is *manual testing*: A dedicated software tester walks through the application feature by feature, case by case according to a test plan.

Manual tests are slow, labor-intensive and cannot be repeated often. They are unspecific from a developer perspective: If the test fails, we cannot easily pin down which part of the application is responsible or which code change causes the regression.

We need automated tests that take the user's perspective. This is what **end-to-end (E2E) tests** do.

Strengths of end-to-end tests

As discussed in [distribution of testing efforts](#), all types of automated tests have pros and cons. Unit and integration tests are fast and reliable, but do not guarantee a working application. End-to-end test are slow and often fail incorrectly, but they assess the fitness of the application as a whole.

REAL CONDITIONS

When all parts of the application come together, a new type of bug appears. Often these bugs have to do with timing and order of events, like network latency and race conditions.

The unit and integration tests we wrote worked with a fake back-end. We send fake HTTP requests and respond with fake data. We made an effort to keep the originals and fakes on par.

FRONT-END AND BACK-END

It is much harder to keep the front-end code in sync with the actual API endpoints and responses from the back-end. Even if the front-end and the back-end share type information about the transferred data, there will be mismatches.

It is the goal of end-to-end tests to catch these bugs that cannot be caught by other automated tests.

Deployment for end-to-end tests

End-to-end tests require a testing environment that closely resembles the production environment. You need to deploy the full application, including the front-end and the relevant back-end parts. For that purpose, back-end frameworks typically support configurations for different environments, like development, testing and production.

DETERMINISTIC ENVIRONMENT

The database needs to be filled with pre-fabricated fake data. With each run of the end-to-end tests, you need to reset the database to a defined initial state.

The back-end services need to answer requests with deterministic responses. Third-party dependencies need to be set up so they return realistic data but do not compromise production data.

Since this guide is not about DevOps, we will not go into details here and focus on writing end-to-end tests.

How end-to-end tests work

An end-to-end test mimics how a user interacts with the application. Typically, the test engine launches an ordinary browser and controls it remotely.

SIMULATE USER ACTIONS

Once the browser is started, the end-to-end test navigates to the application's URL, reads the page content and makes keyboard and pointer input. For example, the test fills out a form and clicks on the submit button.

Just like unit and integration tests, the end-to-end test then makes expectations: Does the page include the right content? Did the URL change? This way, whole features and user interfaces are examined.

End-to-end testing frameworks

Frameworks for end-to-end tests allow navigating to URLs, simulating user input and inspecting the page content. Apart

from that, they have little in common. The test syntax and the way the tests are run differ widely.

There are two categories of end-to-end testing frameworks: Those that use WebDriver and those that do not.

BROWSER AUTOMATION

The **WebDriver protocol** allows to control a browser remotely with a set of commands. It originates from the Selenium browser automation project and is now developed at the World Wide Web Consortium (W3C).

All common browsers support the WebDriver protocol and can be controlled remotely. The most important WebDriver commands are:

- Navigate to a given URL
- Find one or more elements in the DOM
- Get information about a found element:
 - Get an element attribute or property
 - Get the element's text content
- Click on an element
- Send keyboard input to a form field

- Execute arbitrary JavaScript code

WebDriver is a high-level, generic, HTTP-based protocol. It connects the test running on one machine with a browser possibly running on another machine. The level of control over the browser is limited.

FLEXIBILITY VS. RELIABILITY

The main benefit of WebDriver is that tests can be run in different browsers, even simultaneously. Yet only some end-to-end testing frameworks build on WebDriver. Those who do not are more directly integrated into the browser – either via plugins or by patching the browser source code. This makes them more reliable, but also less flexible since they only support certain browsers or custom browser builds.

Up until version 12, Angular used **Protractor** as its default end-to-end testing framework. Protractor was based on WebDriver. Since Angular 12, Protractor is deprecated. In new CLI projects, there is no default end-to-end testing solution configured.

In this guide, we will learn to know **Cypress**, a mature end-to-end testing framework that does not use WebDriver.

 [WebDriver protocol](#)

 [Cypress: Official web site](#)

Introducing Cypress

Cypress is a testing framework that aims to improve the developer experience as well as the performance and reliability of end-to-end tests.

Cypress is the product of one company. The test runner we are going to use is open source and free of charge. The company generates revenue with an additional paid service. The Cypress cloud dashboard manages test runs recorded in a continuous integration environment. You do not have to subscribe to this service to write and run Cypress tests.

ARCHITECTURE

Since Cypress does not use WebDriver, it features a unique architecture. When starting Cypress, a Node.js application launches the browser. The browser is not controlled remotely, but the tests run directly inside the browser, supported by a browser plugin. The test runner provides a powerful user interface for inspecting and debugging tests right in the browser.

TRADE-OFFS

From our perspective, Cypress has a few drawbacks.

- In place of Jasmine, Cypress uses a combination of the libraries Mocha and Chai for writing tests. While both stacks

serve the same purpose, you have to learn the subtle differences. If you use Jasmine for unit and integration tests, your Cypress tests will look similar but work differently in detail.

- Cypress only supports Firefox as well as Chromium-based browsers like Google Chrome and Microsoft Edge. Cypress has experimental support for WebKit, the browser engine used by Safari. Cypress does not support legacy Edge or Internet Explorer.

Cypress is not simply better than WebDriver-based frameworks. It tries to solve their problems by narrowing the scope and by making trade-offs.

RECOMMENDED

That being said, this guide **recommends to use Cypress for testing Angular applications**. Cypress is well-maintained and well-documented. With Cypress, you can write valuable end-to-end tests with little effort.

In case you do need an up-to-date WebDriver-based framework, have a look at Webdriver.io instead.

 [Cypress: Trade-offs](#)

 [Cypress: Key differences](#)

 [Mocha – JavaScript testing framework](#)

 [Chai – assertion library](#)

Installing Cypress

An easy way to add Cypress to an existing Angular CLI project is the [Cypress Angular Schematic](#).

In your Angular project directory, run this shell command:

```
ng add @cypress/schematic
```

This command does four important things:

1. Add Cypress and auxiliary npm packages to `package.json`.
2. Add the Cypress configuration file `cypress.config.ts`.
3. Change the `angular.json` configuration file in order to add `ng run` commands.
4. Create a sub-directory named `cypress` with a scaffold for your tests.

The output looks like this:

```
i Using package manager: npm
✓ Found compatible package version: @cypress/schematic@2.5.0.
✓ Package information loaded.
```

```
The package @cypress/schematic@2.5.0 will be installed and executed.
Would you like to proceed? Yes
✓ Packages successfully installed.
? Would you like the default `ng e2e` command to use Cypress? [ Protractor
to Cypress Migration Guide: https://on.cypress.io/protractor-to-cypress?
cli=true ] Yes
? Would you like to add Cypress component testing? This will add all files
needed for Cypress component testing. No
CREATE cypress.config.ts (134 bytes)
CREATE cypress/tsconfig.json (139 bytes)
CREATE cypress/e2e/spec.cy.ts (143 bytes)
CREATE cypress/fixtures/example.json (85 bytes)
CREATE cypress/support/commands.ts (1377 bytes)
CREATE cypress/support/e2e.ts (649 bytes)
UPDATE package.json (1187 bytes)
UPDATE angular.json (3643 bytes)
✓ Packages installed successfully.
```

The installer asks if you would like the `ng e2e` command to start Cypress. If you are setting up a new project without end-to-end tests yet, it is safe to answer “Yes”.

In Angular CLI prior to version 12, `ng e2e` used to start Protractor. If you have any legacy Protractor tests in the project and want to continue to run them using `ng e2e`, answer “No” to the question.

Writing an end-to-end test with Cypress

In the project directory, you will find a sub-directory called [cypress](#). It contains:

- A [tsconfig.json](#) configuration for all TypeScript files specifically in this directory,
- an [e2e](#) directory for the end-to-end tests,
- a [support](#) directory for custom commands and other testing helpers,
- a [fixtures](#) directory for test data.

The test files reside in the [e2e](#) directory. Each test is TypeScript file with the extension [.cy.ts](#).

The tests itself are structured with the test framework **Mocha**. The assertions (also called expectations) are written using **Chai**.

Mocha and Chai is a popular combination. They roughly do the same as Jasmine, but are much more flexible and rich in features.

TEST SUITES

If you have [written unit tests with Jasmine](#) before, the Mocha structure will be familiar to you. A test file contains one or more suites declared with [describe\('...', \(\) => { /* ... */}\)](#). Typically, one file contains one [describe](#) block, possible with nested [describe](#) blocks.

Inside `describe`, the blocks `beforeEach`, `afterEach`, `beforeAll`, `afterAll` and `it` can be used similar to Jasmine tests.

This brings us to the following end-to-end test structure:

```
describe('... Feature description ...', () => {  
  beforeEach(() => {  
    // Navigate to the page  
  });  
  
  it('... User interaction description ...', () => {  
    // Interact with the page  
    // Assert something about the page content  
  });  
});
```

Testing the counter Component

Step by step, we are going to write end-to-end tests for the counter example application.

 [Counter Component: Source code](#)

 [Counter Component: Run the app](#)

As a start, let us write a minimal test that checks the document title. In the project directory, we create a file called `cypress/e2e/counter.cy.ts`. It looks like this:

```
describe('Counter', () => {  
  beforeEach(() => {
```

```
    cy.visit('/');  
  });  
  
  it('has the correct title', () => {  
    cy.title().should('equal', 'Angular Workshop: Counters');  
  });  
});
```

COMMANDS

Cypress commands are methods of the `cy` namespace object. Here, we are using two commands, `visit` and `title`.

`cy.visit` orders the browser to visit the given URL. Above, we use the path `/`. Cypress appends the path to the `baseUrl`. Per default, the `baseUrl` is set to `http://localhost:4200` in Cypress' configuration file, `cypress.config.ts`.

CHAINERS

`cy.title` returns the page title. To be specific, it returns a Cypress **Chainer**. This is an asynchronous wrapper around an arbitrary value. Most of the time, a Chainer wraps DOM elements. In the case, `cy.title` wraps a string.






ASSERTIONS

The Chainer has a `should` method for creating an assertion. Cypress relays the call to the Chai library to verify the assertion.

```
cy.title().should('equal', 'Angular Workshop: Counters');
```


We pass two parameters, `'equal'` and the expected title string. `equal` creates an assertion that the subject value (the page title) equals to the given value (`'Angular Workshop: Counters'`). `equal` uses the familiar `===` comparison.

This `should` style of assertions differs from Jasmine expectations, which use the `expect(...).toBe(...)` style. In fact, Chai supports three different assertion styles: `should`, `assert`, but also `expect`. In Cypress you will typically use `should` on Chainers and `expect` on unwrapped values.

-  [Cypress API reference: cy.visit](#)
-  [Cypress API reference: cy.title](#)
-  [Cypress documentation: Assertions](#)
-  [Chai API reference: should style assertions](#)
-  [Chai API reference: equal](#)

Running the Cypress tests

Save the minimal test from the last chapter as `cypress/e2e/counter.cy.ts`.

Cypress has two shell commands to run the end-to-end tests:

TEST RUNNER

- **`npx cypress run` – Non-interactive test runner.** Runs the tests in a “headless” browser. This means the browser window is not visible.

The tests are run once, then the browser is closed and the shell command finishes. You can see the test results in the shell output.

This command is typically used in a continuous integration environment.

- **`npx cypress open` – Interactive test runner.** Opens a window where you can select which browser to use and which tests to run. The browser window is visible and it remains visible after completion.

You can see the test results the browser window. If you make changes on the test files, Cypress automatically re-runs the tests.

This command is typically used in the development environment.

SERVE AND RUN TESTS

The Cypress schematic that we have installed wraps these commands so they integrate with Angular.

- `ng run $project-name$:cypress-run` – Starts the non-interactive test runner.
- `ng run $project-name$:cypress-open` – Starts the interactive test runner.

`$project-name$` is a placeholder. Insert the name of the respective Angular project. This is typically the same as the directory name. If not, it can be found in `angular.json` in the `projects` object.

For example, the Counter example has the project name `angular-workshop`. Hence, the commands read:

- `ng run angular-workshop:cypress-run`
- `ng run angular-workshop:cypress-open`

DEVELOPMENT SERVER

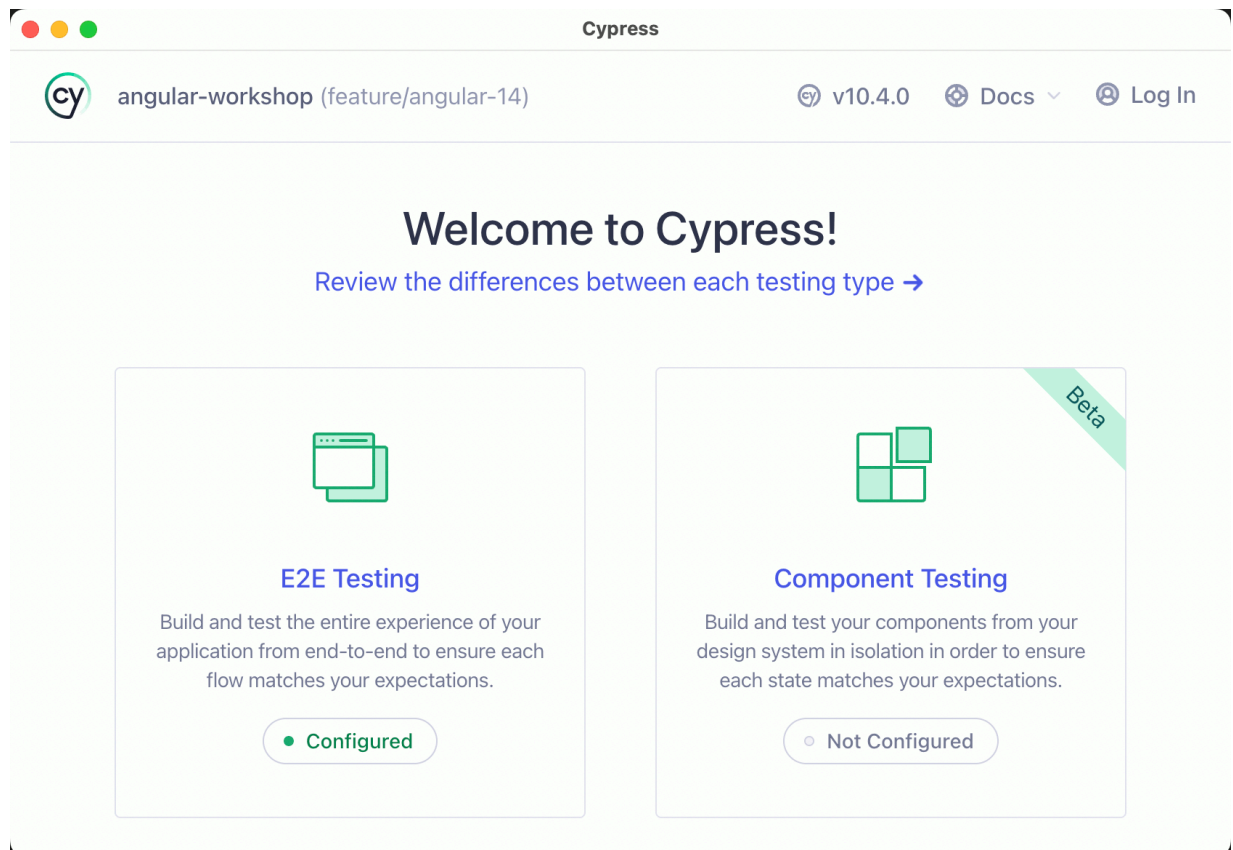
The commands `npx cypress run`, `npx cypress open` and `ng run $project-name$:cypress-open` require you to start Angular's development server with `ng serve` in a separate shell first. Cypress connects to the `baseUrl` (`http://localhost:4200`) and will let you know if the server is not reachable.

The command `ng run $project-name$:cypress-run` starts the development server, runs the tests and stops the server once the

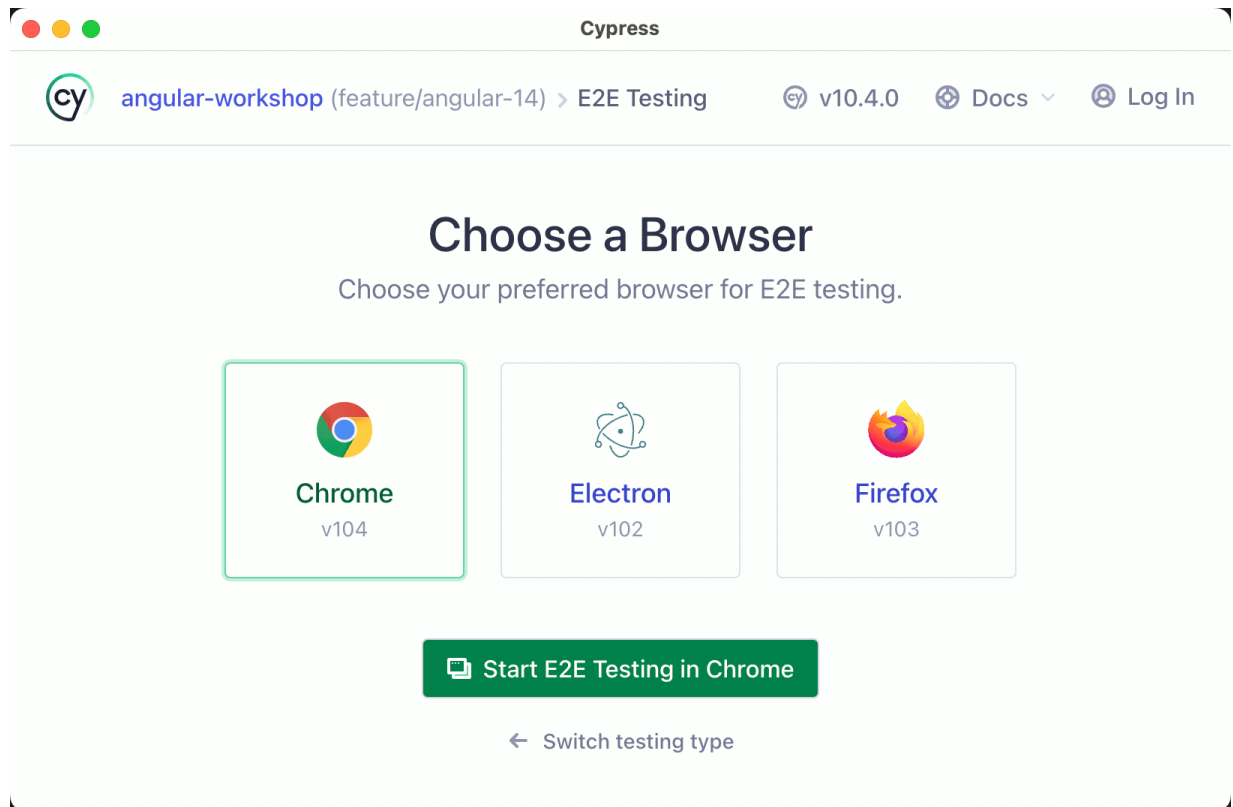
tests have completed.

LAUNCH WINDOW

The `npx cypress open` command will open the test runner. First, you need to choose the type of testing, which is “E2E testing” in our case.

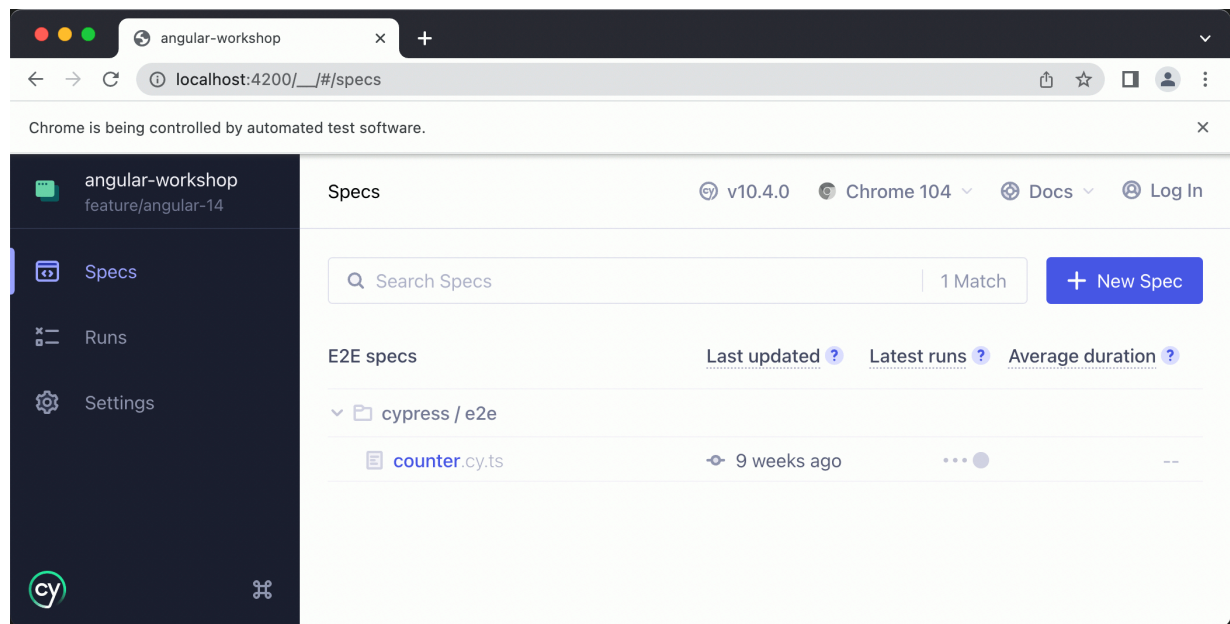


On the next screen, you need to choose the browser for running the tests.



Cypress automatically lists all browsers it finds on your system. In addition, you can run your tests in Electron. Cypress' user interface is an Electron application. Electron is based on Chromium, the open source foundation of the Chrome browser.

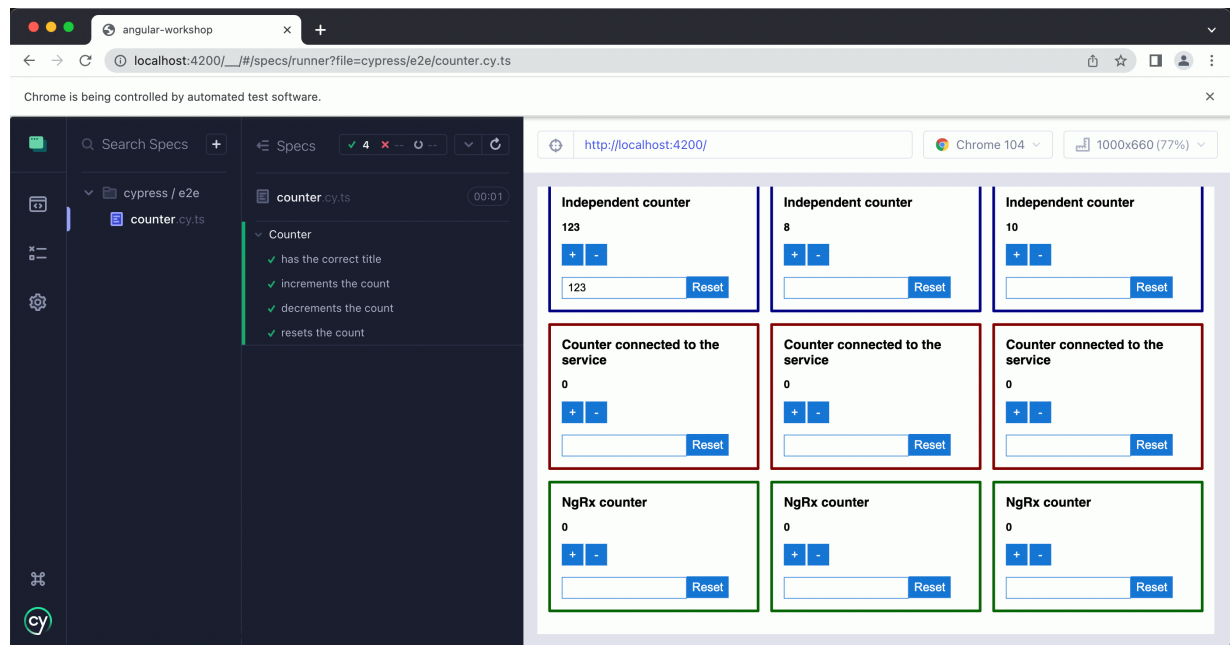
Select a browser and click on the "Start E2E Testing" button. This launches the browser and opens the test runner, Cypress' primary user interface. (The screenshot shows Chrome.)



In the main window pane, all tests are listed. To run a single test, click on it.

TEST RUNNER

Suppose you run the tests in Chrome and run the test `counter.cy.ts`, the in-browser test runner looks like this:

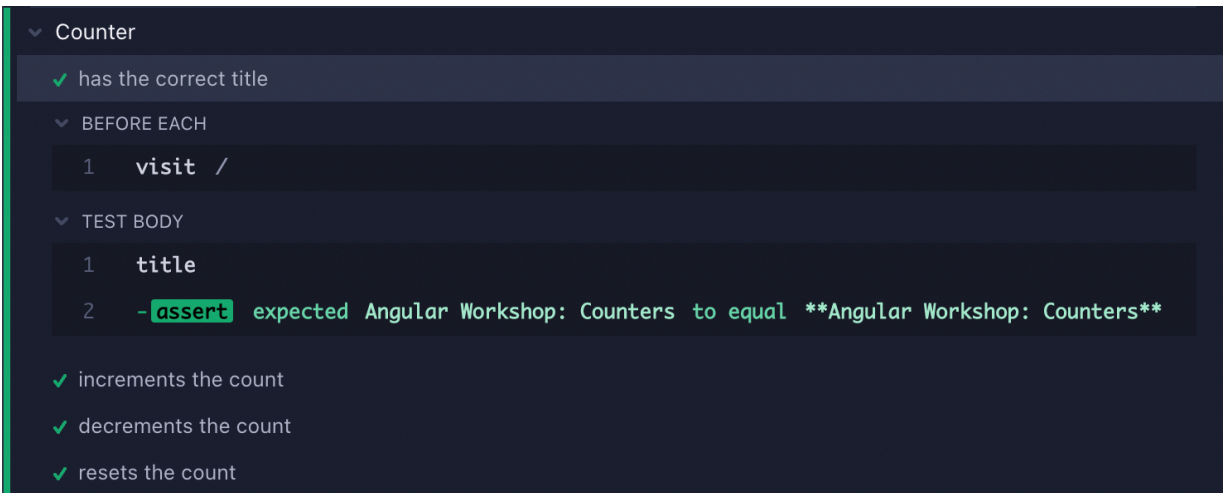


In the “Specs” column, the tests of this test run are listed. For each test, you can see the specs.

On the right side, the web page under test is seen. The web page is scaled to fit into the window, but uses a default viewport width of 1000 pixels.

SPEC LOG

By clicking on a spec name, you can see all commands and assertions in the spec.



You can watch Cypress running the specs command by command. This is especially useful when a spec fails. Let us break the spec on purpose to see Cypress' output.

```
cy.title().should('equal', 'Fluffy Golden Retrievers');
```

This change leads to a failing spec:


```
Counter
✖ has the correct title
BEFORE EACH
1 visit /
TEST BODY
1 title
2 -assert expected Angular Workshop: Counters to equal **Fluffy Golden Retrievers**

! AssertionError
Timed out retrying after 4000ms: expected 'Angular Workshop: Counters' to equal 'Fluffy Golden Retrievers'
cypress/e2e/counter.cy.ts:7:16

5 |
6 | it('has the correct title', () => {
> 7 |   cy.title().should('equal', 'Fluffy Golden Retrievers');
    |                   ^
8 | });
9 |
10 | it('increments the count', () => {

> View stack trace
Print to console
```

Cypress provides a helpful error message, pointing to the assertion that failed. You can click on the file name with line and column, `cypress/e2e/counter.cy.ts:7:16` in the example, to jump right to the assertion in your code editor.

TIME TRAVEL

A unique feature of the in-browser test runner is the ability to see the state of the page at a certain point in time. Cypress creates DOM snapshot whenever a command is run or an assertion verified.

By hovering over a command or assertion, you can travel back in time. The page on the right side then reflects the page when the

command or assertion was processed.

The time travel feature is invaluable when writing and debugging end-to-end tests. Use it to understand how your test interacts with the application and how the application reacts. When a test fails, use it to reconstruct what circumstances lead to the failure.

 [Cypress documentation: The Test Runner](#)

Asynchronous tests

Every Cypress command takes some time to execute. But from the spec point of view, the execution happens instantly.

COMMAND QUEUE

In fact, Cypress commands are merely declarative. The execution happens asynchronously. By calling `cy.visit` and `cy.title`, we add commands to a queue. The queue is processed later, command after command.

As a consequence, we do not need to wait for the result of `cy.visit`. Cypress automatically waits for the page to load before proceeding with the next command.

For the same reason, `cy.title` does not immediately return a string, but a Chainer that allows more declarations.


In the Jasmine unit and integration tests that we wrote before, we had to manage time ourselves. When dealing with asynchronous commands and values, we had to use `async` / `await`, `fakeAsync` and other means explicitly.

This is not necessary when writing Cypress tests. The Cypress API is designed for expressiveness and readability. Cypress hides the fact that all commands take time.

SYNCHRONOUS ASSERTIONS

Sometimes it is necessary to access and inspect a value synchronously. Cypress allows this in form of callback functions that are executed after a certain command was processed. You can pass a callback function to the `should` command or the more general `then` command.

Inside these callbacks, assertions on plain, unwrapped values are written using Chai's `expect` function. We will get to know this practise later.

 [Cypress API reference: should with callback function](#)

 [Cypress API reference: then command](#)

Automatic retries and waiting

A key feature of Cypress is that it retries certain commands and assertions.

For example, Cypress queries the document title and compares it with the expected title. If the title does not match straight away, Cypress will retry the `cy.title` command and the `should` assertion for four seconds. When the timeout is reached, the spec fails.

WAIT AUTOMATICALLY

Other commands are not retried, but have a built-in waiting logic. For example, we are going to use Cypress' `click` method to click on an element.

Cypress automatically waits for four seconds for the element to be clickable. Cypress scrolls the element into view and checks if it is visible and not disabled. After several other checks, the Cypress performs the click.

The retry and waiting timeout can be configured for all tests or individual commands.

RETRY SPECS

If a spec fails despite these retries and waiting, Cypress can be configured to retry the whole spec. This is the last resort if a particular spec produces inconsistent results.

These features makes end-to-end tests more reliable, but also easier to write. In other frameworks, you have to wait manually and there is no automatic retry of commands, assertions or specs.

 [Cypress introduction: Commands are asynchronous](#)

 [Cypress documentation: Interacting with Elements](#)

 [Cypress documentation: Retry-ability](#)

 [Cypress documentation: Test Retries](#)

Testing the counter increment

In our first Cypress test, we have checked the page title successfully. Let us test the counter's increment feature.

The test needs to perform the following steps:

1. Navigate to `"/`.
2. Find the element with the current count and read its text content.
3. Expect that the text is `"5"`, since this is the start count for the first counter.
4. Find the increment button and click it.

5. Find the element with the current count and read its text content (again).
6. Expect that the text now reads "6".

We have used `cy.visit('/')` to navigate to an address. The path `"/"` translates to `http://localhost:4200/` since this is the configured `baseUrl`.

Finding elements

The next step is to find an element in the current page. Cypress provides several ways to find elements. We are going to use the `cy.get` method to find an element by CSS selector.

```
cy.get('.example')
```

`cy.get` returns a Chainer, an asynchronous wrapper around the found elements, enriched with useful methods.

Just like with unit and integration test, the immediate question is: How should we find an element – by id, name, class or by other means?

FIND BY TEST ID

As discussed in [querying the DOM with test ids](#), this guide recommends to mark elements with **test ids**.

These are data attributes like `data-testid="example"`. In the test, we use a corresponding attribute selector to find the elements, for example:

```
cy.get('[data-testid="example"]')
```

FIND BY TYPE

Test ids are recommended, but other ways to find elements are still useful in some cases. For example, you might want to check the presence and the content of an `h1` element. This element has a special meaning and you should not find it with an arbitrary test id.

The benefit of a test id is that it can be used on any element. Using a test id means ignoring the element type (like `h1`) and other attributes. The test does not fail if those change.

But if there is a reason for this particular element type or attribute, your test should verify the usage.

 [Cypress API reference: cy.get](#)

 [Cypress Best Practices: Selecting Elements](#)

Interacting with elements

To test the counter Component, we want to verify that the start count for the first counter is “5”. The current count lives in an element with the test id `count`. So the element finder is:

```
cy.get('[data-testid="count"]')
```

PRESENCE AND CONTENT

The `cy.get` command already has an assertion built-in: It expects to find at least one element matching the selector. Otherwise, the spec fails.

Next, we check the element’s text content to verify the start count. Again, we use the `should` method to create an assertion.

```
cy.get('[data-testid="count"]').should('have.text', '5');
```

The `have.text` assertion compares the text content with the given string.

We did it! We have found an element and checked its content.

CLICK

Now let us increment the count. We find and click on the increment button (test id `increment-button`). Cypress offers the `cy.click` method for this purpose.

```
cy.get('[data-testid="increment-button"]').click();
```


The Angular code under test handles the click event. Finally, we verify that the visible count has increased by one. We repeat the `should('have.text', ...)` command, but expect a higher number.

The test suite now looks like this:

```
describe('Counter', () => {
  beforeEach(() => {
    cy.visit('/');
  });

  it('only has the correct title', () => {
    cy.title().should('equal', 'Angular Workshop: Counters');
  });

  it('increments the count', () => {
    cy.get('[data-testid="count"]').should('have.text', '5');
    cy.get('[data-testid="increment-button"]').click();
    cy.get('[data-testid="count"]').should('have.text', '6');
  });
});
```

The next feature we need to test is the decrement button. The spec works similar to the increment spec. It clicks on the decrement button (test id `decrement-button`) and checks that the count has decreased.

```
it('decrements the count', () => {
  cy.get('[data-testid="decrement-button"]').click();
  cy.get('[data-testid="count"]').should('have.text', '4');
});
```

Last but not least, we test the reset feature. The user can enter a new count into a form field (test id `reset-input`) and click on the reset button (test id `reset-button`) to set the new count.

FILL OUT FORM

The Cypress Chainer has a generic method for sending keys to an element that the keyboard can interact with: `type`.

To enter text into the form field, we pass a string to the `type` method.

```
cy.get('[data-testid="reset-input"]').type('123');
```

Next, we click on the reset button and finally expect the change.

```
it('resets the count', () => {  
  cy.get('[data-testid="reset-input"]').type('123');  
  cy.get('[data-testid="reset-button"]').click();  
  cy.get('[data-testid="count"]').should('have.text', '123');  
});
```

This is the full test suite:

```
describe('Counter', () => {  
  beforeEach(() => {  
    cy.visit('/');  
  });  
  
  it('has the correct title', () => {  
    cy.title().should('equal', 'Angular Workshop: Counters');  
  });  
});
```

```

it('increments the count', () => {
  cy.get('[data-testid="count"]').should('have.text', '5');
  cy.get('[data-testid="increment-button"]').click();
  cy.get('[data-testid="count"]').should('have.text', '6');
});

it('decrements the count', () => {
  cy.get('[data-testid="decrement-button"]').click();
  cy.get('[data-testid="count"]').should('have.text', '4');
});

it('resets the count', () => {
  cy.get('[data-testid="reset-input"]').type('123');
  cy.get('[data-testid="reset-button"]').click();
  cy.get('[data-testid="count"]').should('have.text', '123');
});
});

```

On the start page of the counter project, there are in fact nine counter instances. The `cy.get` command therefore returns nine elements instead of one.

FIRST MATCH

Commands like `type` and `click` can only operate on one element, so we need to reduce the element list to the first result. This is achieved by Cypress' `first` command inserted in the chain.

```

it('increments the count', () => {
  cy.get('[data-testid="count"]').first().should('have.text', '5');
  cy.get('[data-testid="increment-button"]').first().click();

```

```
cy.get('[data-testid="count"]').first().should('have.text', '6');
});
```

This also applies to the other specs. If the element under test only appears once, the `first` command is not necessary, of course.

All counter features are now tested. In the next chapters, we will refactor the code to improve its readability and maintainability.

 [Counter E2E test code](#)

 [Cypress API reference: click](#)

 [Cypress API reference: type](#)

 [Cypress API reference: first](#)

 [Cypress FAQ: How do I get an element's text contents?](#)

Custom Cypress commands

The test we wrote is quite repetitive. The pattern `cy.get('[data-testid="..."]')` is repeated over and over.

The first improvement is to write a helper that hides this detail.

We have already written two similar functions as [unit testing helpers](#), `findEl` and `findEls`.

FIND BY TEST ID

The easiest way to create a Cypress helper for finding elements is a function.

```
function findEl(testId: string): Cypress.Chainable<JQuery<HTMLElement>> {  
  return cy.get(`[data-testid="${testId}"]`);  
}
```

This would allow us to write `findEl('count')` instead of `cy.get('[data-testid="count"]')`.

CUSTOM COMMANDS

This works fine, but we opt for a another way. Cypress supports adding **custom commands** to the `cy` namespace. We are going to add the command `byTestId` so we can write `cy.byTestId('count')`.

Custom commands are placed in `cypress/support/commands.ts`. This file is automatically created by the Angular schematic. Using `Cypress.Commands.add`, we add our own command as a method of `cy`. The first parameter is the command name, the second is the implementation as a function.

CY.BYTESTID

The simplest version looks like this:

```
Cypress.Commands.add(  
  'byTestId',  
  (id: string) =>  
    cy.get(`[data-testid="${id}"]`)  
);
```

Now we can write `cy.byId('count')`. We can still fall back to `cy.get` if we want to find an element by other means.

`cy.byId` should have the same flexibility as the generic `cy.get`. So we add the second `options` parameter as well. We borrow the function signature from the official `cy.get` typings.

```
Cypress.Commands.add(
  'byTestId',
  // Borrow the signature from cy.get
  <E extends Node = HTMLElement>(
    id: string,
    options?: Partial<
      Cypress.Loggable & Cypress.Timeoutable & Cypress.Withinable &
      Cypress.Shadow
    >,
  ): Cypress.Chainable<jQuery<E>> =
    cy.get(`[data-testid="${id}"]`, options),
);
```

For proper type checking, we need to tell the TypeScript compiler that we have extended the `cy` namespace. In `commands.ts`, we extend the `Chainable` interface with a method declaration for `byTestId`.

```
declare namespace Cypress {
  interface Chainable {
    /**
     * Get one or more DOM elements by test id.
     *
     * @param id The test id
     */
  }
}
```

```

    * @param options The same options as cy.get
    */
    byTestId<E extends Node = HTMLElement>(
      id: string,
      options?: Partial<
        Cypress.Loggable & Cypress.Timeoutable & Cypress.Withinable &
        Cypress.Shadow
      >,
    ): Cypress.Chainable<jQuery<E>>;
  }
}

```

You do not have to understand these type definitions in detail. They simply make sure that you can pass the same `options` to `cy.byTestId` that you can pass to `cy.get`.

Save `commands.ts`, then open `cypress/support/e2e.ts` and activate the line that imports `command.ts`.

```
import './commands';
```

This is it! We now have a strictly typed command `cy.byTestId`. Using the command, we can declutter the test.

```

describe('Counter (with helpers)', () => {
  beforeEach(() => {
    cy.visit('/');
  });

  it('has the correct title', () => {
    cy.title().should('equal', 'Angular Workshop: Counters');
  });
}





```

```
it('increments the count', () => {
  cy.byId('count').first().should('have.text', '5');
  cy.byId('increment-button').first().click();
  cy.byId('count').first().should('have.text', '6');
});

it('decrements the count', () => {
  cy.byId('decrement-button').first().click();
  cy.byId('count').first().should('have.text', '4');
});

it('resets the count', () => {
  cy.byId('reset-input').first().type('123');
  cy.byId('reset-button').first().click();
  cy.byId('count').first().should('have.text', '123');
});
});
```

Keep in mind that all these `first` calls are only necessary since there are multiple counters on the example page under test. If there is only one element with the given test id on the page, you do not need them.

-  [Counter E2E test with helpers](#)
-  [Full code: commands.ts](#)
-  [Cypress documentation: Custom commands](#)
-  [Cypress documentation: Types for custom commands](#)

Testing the Flickr search

We have learned the basics of Cypress by testing the counter app. Let us delve into end-to-end testing with Cypress by testing a more complex app, the Flickr search.

 [Flickr photo search: Source code](#)

 [Flickr photo search: Run the app](#)

Before writing any code, let us plan what the end-to-end test needs to do:

1. Navigate to “/”.
2. Find the search input field and enter a search term, e.g. “flower”.
3. Find the submit button and click on it.
4. Expect photo item links to flickr.com to appear.
5. Click on a photo item.
6. Expect the full photo details to appear.

NONDETERMINISTIC API

The application under test queries a third-party API with production data. The test searches for “flower” and with each test run, Flickr returns potentially different results.

There are two ways to deal with this dependency during testing:

1. Test against the *real* Flickr API.
2. *Fake* the Flickr API and return a fixed response.

If we test against the real Flickr API, we cannot be specific in our expectations due to changing search results. We can superficially test the search results and the full photo. We merely know that the clicked photo has “flower” in its title or tags.

REAL VS. FAKE API

This has pros and cons. Testing against the real Flickr API makes the test realistic, but less reliable. If the Flickr API has a short hiccup, the test fails although there is no bug in our code.

Running the test against a fake API allows us to inspect the application deeply. Did the application render the photos the API returned? Are the photo details shown correctly?

Keep in mind that unit, integration and end-to-end tests complement each other. The Flickr search is also tested extensively using unit and integration tests.

Each type of test should do what it does best. The unit tests already put the different photo Components through their paces. The end-to-end test does not need to achieve that level of detail.

With Cypress, both type of tests are possible. For a start, we will test against the real Flickr API. Then, we will fake the API.

Testing the search form

We create a file called `cypress/e2e/flickr-search.cy.ts`. We start with a test suite.

```
describe('Flickr search', () => {
  const searchTerm = 'flower';

  beforeEach(() => {
    cy.visit('/');
  });

  it('searches for a term', () => {
    /* ... */
  });
});
```

We instruct the browser to enter “flower” into the search field (test id `search-term-input`). Then we click on the submit button (test id `submit-search`).

```
it('searches for a term', () => {
  cy.byId('search-term-input')
    .first()
    .clear()
    .type(searchTerm);
  cy.byId('submit-search').first().click();
  /* ... */
});
```

CLEAR, THEN TYPE

The `type` command does not overwrite the form value with a new value, but sends keyboard input, key by key.

Before entering “flower”, we need to clear the field since it already has a pre-filled value. Otherwise we would append “flower” to the existing value. We use Cypress’ `clear` method for that purpose.

Clicking on the submit button starts the search. When the Flickr API has responded, we expect the search results to be appear.

EXPECT SEARCH RESULTS

A search result consists of a link (`a` element, test id `photo-item-link`) and an image (`img` element, test id `photo-item-image`).

We expect 15 links to appear since this is amount of results requested from Flickr.

```
cy.byTestId('photo-item-link')  
  .should('have.length', 15)
```

By writing `should('have.length', 15)`, we assert that there are 15 elements.

Each link needs to have an `href` containing `https://www.flickr.com/photos/`. We cannot check for an exact URL since results are the dynamic. But we know that all Flickr photo URLs have the same structure.

There is no direct Chai assertion for checking that each link in the list has an `href` attribute containing `https://www.flickr.com/photos/`. We need to check each link in the list individually.

The Chainer has an `each` method to call a function for each element. This works similar to JavaScript's `forEach` array method.

```
cy.byId('photo-item-link')
  .should('have.length', 15)
  .each((link) => {
    /* Check the link */
  });
```

Cypress has three surprises for us.

SYNCHRONOUS JQUERY OBJECT

1. `link` is a synchronous value. Inside the `each` callback, we are in synchronous JavaScript land. (We could do asynchronous operations here, but there is no need.)
2. `link` has the type `JQuery<HTMLElement>`. This is an element wrapped with the popular jQuery library. Cypress chose jQuery because many JavaScript developers are already familiar with it. To read the `href` attribute, we use `link.attr('href')`.
3. We cannot use Cypress' `should` method since it only exists on Cypress Chainers. But we are dealing with a jQuery object

here. We have to use a standard Chai assertion. We use `expect` together with `to.contain`.

This brings us to:

```
cy.byId('photo-item-link')
  .should('have.length', 15)
  .each((link) => {
    expect(link.attr('href')).to.contain(
      'https://www.flickr.com/photos/'
    );
  });
```

The test now looks like this:

```
describe('Flickr search', () => {
  const searchTerm = 'flower';

  beforeEach(() => {
    cy.visit('/');
  });

  it('searches for a term', () => {
    cy.byId('search-term-input')
      .first()
      .clear()
      .type(searchTerm);
    cy.byId('submit-search').first().click();






    cy.byId('photo-item-link')
      .should('have.length', 15)
      .each((link) => {
        expect(link.attr('href')).to.contain(
```

```
        'https://www.flickr.com/photos/'
    );
  });
  cy.byId('photo-item-image').should('have.length', 15);
});
});
```

To start the tests, we first start the development server with `ng serve` and then start Cypress:

```
ng run flickr-search:cypress-open
```

This opens the test runner where we click on `flickr-search.cy.ts`.

-  [Flickr search E2E test code](#)
-  [Cypress API reference: clear](#)
-  [Cypress API reference: each](#)
-  [jQuery API reference: attr](#)
-  [Chai API reference: include \(contain\)](#)

Testing the full photo

When the user clicks on a link in the result list, the click event is caught and the full photo details are shown next to the list. (If the user clicks with the control/command key pressed or right-clicks, they can follow the link to flickr.com.)

In the end-to-end test, we add a spec to verify this behavior.

```
it('shows the full photo', () => {  
  /* ... */  
});
```

First, it searches for “flower”, just like the spec before.

```
cy.byId('search-term-input').first().clear().type(searchTerm);  
cy.byId('submit-search').first().click();
```

Then we find all photo item links, but not to inspect them, but to click on the first one:

```
cy.byId('photo-item-link').first().click();
```

The click lets the photo details appear. As mentioned above, we cannot check for a specific title, a specific photo URL or specific tags. The clicked photo might be a different one with each test run.

Since we have searched for “flower”, the term is either in the photo title or tags. We check the text content of the wrapper element with the test id `full-photo`.

```
cy.byId('full-photo').should('contain', searchTerm);
```

CONTAIN VS. HAVE TEXT

The `contain` assertion checks whether the given string is somewhere in the element’s text content. (In contrast, the `have.text` assertion checks whether the content equals the given string. It does not allow additional content.)

Next, we check that a title and some tags are present and not empty.

```
cy.byId('full-photo-title').should('not.have.text', '');  
cy.byId('full-photo-tags').should('not.have.text', '');
```

The image itself needs to be present. We cannot check the `src` attribute in detail.

```
cy.byId('full-photo-image').should('exist');
```

The spec now looks like this:

```
it('shows the full photo', () => {  
  cy.byId('search-term-input').first().clear().type(searchTerm);  
  cy.byId('submit-search').first().click();  
  
  cy.byId('photo-item-link').first().click();  
  cy.byId('full-photo').should('contain', searchTerm);  
  cy.byId('full-photo-title').should('not.have.text', '');  
  cy.byId('full-photo-tags').should('not.have.text', '');  
  cy.byId('full-photo-image').should('exist');  
});
```

The assertions `contain`, `text` and `exist` are defined by Chai-jQuery, an assertion library for checking jQuery element lists.

Congratulations, we have successfully tested the Flickr search! This example demonstrates several Cypress commands and assertions. We also caught a glimpse of Cypress internals.

 [Flickr search E2E test code](#)

Page objects

The Flickr search end-to-end test we have written is fully functional. We can improve the code further to increase clarity and maintainability.

We introduce a design pattern called **page object**. A design pattern is a proven code structure, a best practice to solve a common problem.

HIGH-LEVEL INTERACTIONS

A page object represents the web page that is scrutinized by an end-to-end test. The page object provides a high-level interface for interacting with the page.

So far, we have written low-level end-to-end tests. They find individual elements by hard-coded test id, check their content and click on them. This is fine for small tests.

But if the page logic is complex and there are diverse cases to test, the test becomes an unmanageable pile of low-level instructions. It is hard to find the gist of these tests and they are hard to change.

A page object organizes numerous low-level instructions into a few high-level interactions. What are the high-level interactions in the Flickr search app?

1. Search photos using a search term
2. Read the photo list and interact with the items
3. Read the photo details

Where possible, we group these interactions into methods of the page object.

PLAIN CLASS

A page object is merely an abstract pattern – the exact implementation is up to you. Typically, the page object is declared as a class that is instantiated when the test starts.

Let us call the class `FlickrSearch` and save it in a separate file, `cypress/pages/flickr-search.page.ts`. The directory `pages` is reserved for page objects, and the `.page.ts` suffix marks the page object.

```
export class FlickrSearch {  
  public visit(): void {  
    cy.visit('/');  
  }  
}
```

The class has a `visit` method that opens the page that the page object represents.

In the test, we import the class and create an instance in a `beforeEach` block.

```
import { FlickrSearch } from '../pages/flickr-search.page';

describe('Flickr search (with page object)', () => {
  const searchTerm = 'flower';

  let page: FlickrSearch;

  beforeEach(() => {
    page = new FlickrSearch();
    page.visit();
  });

  /* ... */
});
```

The `FlickrSearch` instance is stored in a variable declared in the `describe` scope. This way, all specs can access the page object.

SEARCH

Let us implement the first high-level interaction on the page object: searching for photos. We move the relevant code from the test into a method of the page object.

```
public searchFor(term: string): void {
  cy.byTestId('search-term-input').first().clear().type(term);
```

```
    cy.byId('submit-search').first().click();  
}
```

The `searchFor` method expects a search term and performs all necessary steps.

ELEMENT QUERIES

Other high-level interactions, like reading the photo list and the photo details, cannot be translated into page object methods. But we can move the test ids and element queries to the page object.

```
public photoItemLinks(): Cypress.Chainable<JQuery<HTMLElement>> {  
    return cy.byId('photo-item-link');  
}
```

```
public photoItemImages(): Cypress.Chainable<JQuery<HTMLElement>> {  
    return cy.byId('photo-item-image');  
}
```

```
public fullPhoto(): Cypress.Chainable<JQuery<HTMLElement>> {  
    return cy.byId('full-photo');  
}
```

```
public fullPhotoTitle(): Cypress.Chainable<JQuery<HTMLElement>> {  
    return cy.byId('full-photo-title');  
}
```

```
public fullPhotoTags(): Cypress.Chainable<JQuery<HTMLElement>> {  
    return cy.byId('full-photo-tags');  
}
```

```
public fullPhotoImage(): Cypress.Chainable<JQuery<HTMLElement>> {
```

```
    return cy.byTestId('full-photo-image');  
  }  
}
```

These methods return element Chainers.

Next, we rewrite the end-to-end test to use the page object methods.

```
import { FlickrSearch } from '../pages/flickr-search.page';
```

```
describe('Flickr search (with page object)', () => {  
  const searchTerm = 'flower';
```

```
  let page: FlickrSearch;
```

```
  beforeEach(() => {  
    page = new FlickrSearch();  
    page.visit();  
  });
```

```
  it('searches for a term', () => {  
    page.searchFor(searchTerm);  
    page  
      .photoItemLinks()  
      .should('have.length', 15)  
      .each((link) => {  
        expect(link.attr('href')).to.contain(  
          'https://www.flickr.com/photos/'  
        );  
      });  
    page.photoItemImages().should('have.length', 15);  
  });
```

```
  it('shows the full photo', () => {
```

```
page.searchFor(searchTerm);
page.photoItemLinks().first().click();
page.fullPhoto().should('contain', searchTerm);
page.fullPhotoTitle().should('not.have.text', '');
page.fullPhotoTags().should('not.have.text', '');
page.fullPhotoImage().should('exist');
});
});
```

For the Flickr search above, a page object is probably too much of a good thing. Still, the example demonstrates the key ideas of page objects:

- Identify repetitive high-level interactions and map them to methods of the page object.
- Move the finding of elements into the page object. The test ids, tag names, etc. used for finding should live in a central place.

When the markup of a page under test changes, the page object needs an update, but the test should remain unchanged.

- Leave all assertions ([should](#) and [expect](#)) in the specs. Do not move them to the page object.

HIGH-LEVEL TESTS

When writing end-to-end tests, you get lost in technical details quickly: finding elements, clicking them, filling out form fields, checking fields values and text content. But end-to-end tests should not revolve around these low-level details. They should describe the user journey on a high level.

The goal of this refactoring is not brevity. Using page objects does not necessarily lead to less code. The purpose of page objects is to separate low-level details – like finding elements by test ids – from the high-level user journey through the application. This makes the specs easier to read and the easier to maintain.

You can use the page object pattern when you feel the need to tidy up complex, repetitive tests. Once you are familiar with the pattern, it also helps you to avoid writing such tests in the first place.

 [Flickr search E2E test with page object](#)

 [Flickr search page object](#)

Faking the Flickr API

The end-to-end test we wrote for the Flickr search uses the real Flickr API. As discussed, this makes the test realistic.

The test provides confidence that the application works hand in hand with the third-party API. But it makes the test slower and only allows unspecific assertions.

INTERCEPT HTTP REQUESTS

With Cypress, we can uncouple the dependency. Cypress allows us to intercept HTTP requests and respond with fake data.

First of all, we need to set up the fake data. We have already created fake photo objects for the [FlickrService unit test](#). For simplicity, we just import them:

```
import {
  photo1,
  photo1Link,
  photos,
  searchTerm,
} from '../src/app/spec-helpers/photo.spec-helper';
```

Using the fake photos, we create a fake response object that mimics the relevant part of the Flickr response.

```
const flickrResponse = {
  photos: {
    photo: photos,
  },
};
```

FAKE SERVER WITH ROUTE

Now we instruct Cypress to intercept the Flickr API request and answer it with fake data. This setup happens in the test's `beforeEach` block. The corresponding Cypress command is `cy.intercept`.

```
beforeEach(() => {
  cy.intercept(
    {
      method: 'GET',
      url: 'https://www.flickr.com/services/rest/*',
      query: {
        tags: searchTerm,
        method: 'flickr.photos.search',
        format: 'json',
        nojsoncallback: '1',
        tag_mode: 'all',
        media: 'photos',
        per_page: '15',
        extras: 'tags,date_taken,owner_name,url_q,url_m',
        api_key: '*',
      },
    },
    {
      body: flickrResponse,
      headers: {
        'Access-Control-Allow-Origin': '*',
      },
    },
  ).as('flickrSearchRequest');

  cy.visit('/');
});
```

`cy.intercept` can be called in different ways. Here, we pass two objects:

1. A *route matcher* describing the requests to intercept. It contains the HTTP GET method, the base URL and a whole bunch of query string parameters. In the URL and the `api_key` query parameter, the `*` character is a wildcard that matches any string.
2. A *route handler* describing the response Cypress should send. As JSON response body, we pass the `flickrResponse` fake object.

Since the request to Flickr is cross-origin, we need to set the `Access-Control-Allow-Origin: *` header. This allows our Angular application at the origin `http://localhost:4200` to read the response from the origin `https://www.flickr.com/`.

ALIAS

Finally, we give the request an *alias* by calling `.as('flickrSearchRequest')`. This makes it possible to refer to the request later using the `@flickrSearchRequest` alias.

After this setup, Cypress intercepts the request to Flickr and handles it by itself. The original Flickr API is not reached.

The existing, rather generic specs still pass. Before we make them more specific, we need to verify that Cypress found a match and intercepted the HTTP request. Because if it did not, the test would still pass.

WAIT FOR REQUEST

We can achieve this by explicitly waiting for the request after starting the search.

```
it('searches for a term', () => {  
  cy.byId('search-term-input').first().clear().type(searchTerm);  
  cy.byId('submit-search').first().click();  
  
  cy.wait('@flickrSearchRequest');  
  
  /* ... */  
});
```

`cy.wait('@flickrSearchRequest')` tells Cypress to wait for a request that matches the specified criteria.

`@flickrSearchRequest` refers to the alias we have defined above.

If Cypress does not find a matching request until a timeout, the test fails. If Cypress caught the request, we know that the Angular application received the photos specified in the `photos` array.

SPECIFIC ASSERTIONS

By faking the Flickr API, we gain complete control over the response. We chose to return fixed data. The application under

test processes the data deterministically. As discussed, this allows us to verify that the application correctly renders the photos the API returned.

Let us write specific assertions that compare the photos in the result list with those in the `photos` array.

```
it('searches for a term', () => {
  cy.byId('search-term-input').first().clear().type(searchTerm);
  cy.byId('submit-search').first().click();

  cy.wait('@flickrSearchRequest');

  cy.byId('photo-item-link')
    .should('have.length', 2)
    .each((link, index) => {
      expect(link.attr('href')).to.equal(
        `https://www.flickr.com/photos/${photos[index].owner}/${photos[index].id}`,
      );
    });
  cy.byId('photo-item-image')
    .should('have.length', 2)
    .each((image, index) => {
      expect(image.attr('src')).to.equal(photos[index].url_q);
    });
});
```

Here, we walk through the links and images to ensure that the URLs originate from the fake data. Previously, when testing

against the real API, we tested the links only superficially. We could not test the image URLs at all.

Likewise, for the full photo spec, we make the assertions more specific.

```
it('shows the full photo', () => {
  cy.byId('search-term-input').first().clear().type(searchTerm);
  cy.byId('submit-search').first().click();

  cy.wait('@flickrSearchRequest');

  cy.byId('photo-item-link').first().click();
  cy.byId('full-photo').should('contain', searchTerm);
  cy.byId('full-photo-title').should('have.text', photo1.title);
  cy.byId('full-photo-tags').should('have.text', photo1.tags);
  cy.byId('full-photo-image').should('have.attr', 'src', photo1.url_m);
  cy.byId('full-photo-link').should('have.attr', 'href', photo1Link);
});
```

The specs now ensure that the application under test outputs the data from the Flickr API. `have.text` checks an element's text content, whereas `have.attr` checks the `src` and `href` attributes.

We are done! Our end-to-end test intercepts an API request and responds with fake data in order to inspect the application deeply.

INTERCEPT ALL REQUESTS

In the case of the Flickr search, we have intercepted an HTTP request to a third-party API. Cypress allows to fake any request,

including those to your own HTTP APIs.

This is useful for returning deterministic responses crucial for the feature under test. But it is also useful for suppressing requests that are irrelevant for your test, like marginal images and web analytics.

- 🔗 [Flickr search E2E test with cy.intercept](#)
- 🔗 [Photo spec helper](#)
- 🔗 [Cypress documentation: Network Requests](#)
- 🔗 [Cypress API reference: intercept](#)
- 🔗 [Cypress API reference: wait](#)

End-to-end testing: Summary

End-to-end tests used to be expensive while the outcome was poor. It was hard to write tests that reliably pass even when the application is working correctly. This time could not be invested in writing useful tests that uncover bugs and regressions.

For years, Protractor was the end-to-end testing framework many Angular developers relied on. With Cypress, a framework arose that sets new standards.

This guide recommends to start with Cypress because it excels in developer experience and cost-effectiveness. Still, WebDriver-

based frameworks like Webdriver.io are useful if you need to test a broad range of browsers.

Even with Cypress, end-to-end tests are much more complex and error-prone than unit and integration tests with Jasmine and Karma. Then again, end-to-end tests are highly effective to test a feature under realistic circumstances.

 [Counter: Cypress tests](#)

 [Flickr search: Cypress tests](#)

Summary

LEARNING OBJECTIVES

- Reducing the frustration from implementing and testing an application
- Taking small steps when learning testing techniques and expanding test coverage
- Growing as a developer and as a team by practicing automated testing

Writing tests is often a frustrating experience that comes on top of implementation troubles. The logical and technical complexity is overwhelming and intimidating. All this drains motivation.

The goal of this guide is to dispel the fear of testing. While testing Angular applications is a complex matter, this guide breaks it down into small, approachable steps and aims to give a well-balanced overview.

Find a testing strategy that reduces the frustration and benefits the quality of your software. Once you have written tests for a couple of features, you will learn which tests are worthwhile – tests that uncover bugs and prevent regressions. Continue with these successes, then slowly explore other kind of tests.







Luckily, the Angular community works steadily to make testing accessible. Angular's architecture facilitates the testing of all relevant parts. The framework ships with robust testing tools. If they do not fit your needs, there are mature community projects with alternative workflows.

Testing does not only make your software more reliable, but also evolves your coding practice in the long run. It requires to write testable code, and testable code is usually simpler.

Automated software testing is challenging and rewarding for various reasons. Despite all troubles, that makes it fascinating.

Index of example applications

All example applications are repositories on GitHub:

-  [Counter Component](#)
-  [Flickr photo search](#)
-  [Sign-up form](#)
-  [TranslatePipe](#)
-  [ThresholdWarningDirective](#)
-  [PaginateDirective](#)

References

- [Angular: Grundlagen, fortgeschrittene Themen und Best Practices](#), Second Edition, Ferdinand Malcher, Johannes Hoppe, Danny Koppenhagen. dpunkt.verlag, 2019. ISBN 978-3-86490-646-6
- [Testing Angular Applications](#), Jesse Palmer, Corinna Cohn, Mike Giambalvo, Craig Nishina. Manning Publications, 2018. ISBN 978-1-61729-364-1
- [Testing JavaScript Applications](#), Lucas da Costa. Manning Publications, 2021. ISBN 978-1-61729-791-5
- [JavaScript Testing Recipes](#), James Coglan, 2016.

Acknowledgements

This book would not be possible without the support and input from many individuals. I would like to thank everyone in the Angular, JavaScript and web development communities who not only shares knowledge and works on open source tools, but also advocates for inclusive communities and software.

Thanks to the teams at 9elements, Diebold Nixdorf and Keysight Technologies for the opportunity to work on first-class Angular applications. Thanks for the challenges, resources and patience that allowed me to research automated testing in detail.

Thanks to [Netanel Basal](#) for valuable feedback on the book, for creating Spectator and for many helpful articles on Angular and testing.

Thanks to [Nils Binder](#) for helping with the design, including the dark color scheme. Thanks to Melina Jacob for designing the e-book cover.

Thanks to [Tim Deschryver](#), [Kent C. Dodds](#), [Kara Erickson](#), [Asim Hussain](#), [Tracy Lee](#), [Brandon Roberts](#), [Jesse Palmer](#), Corinna Cohn, [Mike Giambalvo](#), [Craig Nishina](#), [Lucas F. Costa](#) and [Jessica Sachs](#) for insights on Angular, RxJS and automated testing.

About

Author: [Mathias Schäfer \(molily\)](#).

Mathias is a software developer with a focus on web development and JavaScript web applications.

Mathias is working for [9elements](#), a software and design agency.

[@molily on Mastodon](#), [@molily on Twitter](#).

Please send feedback and corrections to molily@mailbox.org or [file an issue on GitHub](#).

First published on February 17th, 2021.

License

License: [Creative Commons Attribution-ShareAlike \(CC BY-SA 4.0\)](#)

The example code is released into the public domain. See [Unlicense](#).

The [Flickr search example application](#) uses the [Flickr API](#) but is not endorsed or certified by Flickr, Inc. or SmugMug, Inc. Flickr is a trademark of Flickr, Inc. The displayed photos are property of their respective owners.

Online book cover photo: Flying probes testing a printed circuit board by genkur, [licensed from iStock](#).

E-book cover and favicon: [Official Angular logo](#) by Google, licensed under [Creative Commons Attribution \(CC BY 4.0\)](#).

Legal notice: [Impressum und Datenschutz](#)