- Unit Testing In Angular

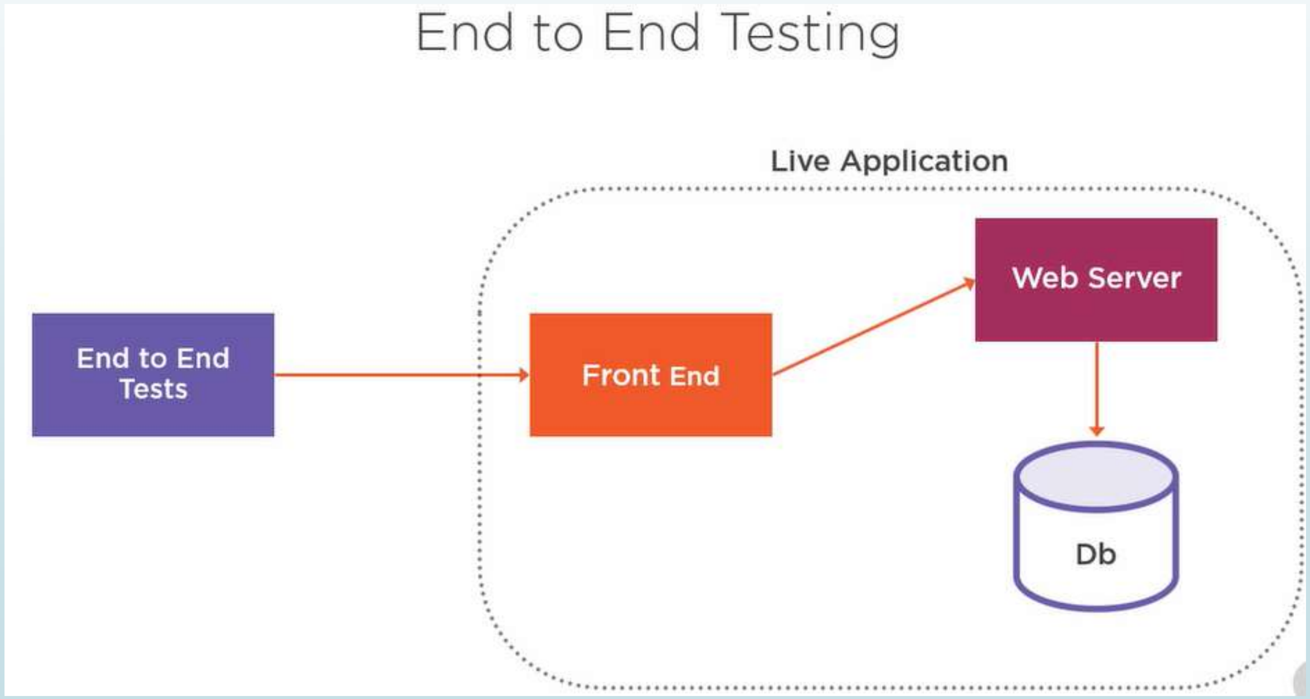
Automated Testing in General

- There are three types of automated tests.
- Unit testing, end-to-end testing, and integration, or functional, testing.
- Unit testing and end-to-end testing are fairly well defined, integration or functional testing tends to be a little bit more of a vague concept that can mean one thing to one person and something different to somebody else.
- Let's take a look at each of these types of tests.

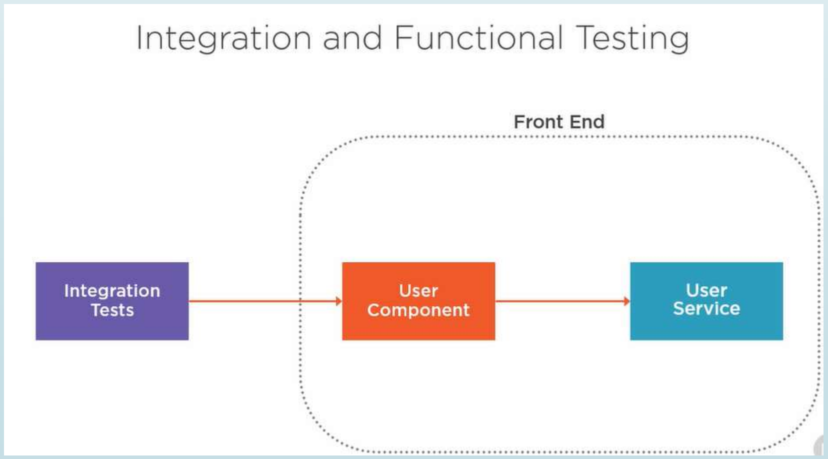
- <u>First we have end-to-end testing.</u>
- End-to-end testing is the kind of testing that is done against a live, running application.
- This means the full application with a live database, live server, live front-end.
- Then we write tests that exercise that live application.
- This is generally done through automating the browser.
- Tests are written to manipulate the browser in an automated way, to do things like click buttons, type values into forms, navigate around, and similar tasks.
- The benefit of end-to-end testing is you can validate that your application works as a whole.
- There are plenty of drawbacks to end-to-end testing, which have to do with speed and difficulty of writing tests.
- Generally end-to-end testing tends to be less reliable than other types of automated tests.

- Automated Testing in General
- Next we go to the other end of the spectrum, which is unit testing.
- Unit testing is done against a single unit of code.
- Generally, the accepted unit of code is a single class.
- Although, in some cases, we may define a unit as more than a single class.
- Next we have everything in between, integration and functional testing.
- Integration and functional testing is defined as more than a unit, but less than the complete application.
- So it's at least two units working together; oftentimes, these types of tests are used to check that a certain part of the application works with another part of the application.
- Let's look at some diagrams of these concepts.

- Automated Testing in General
- We've got our live application, which includes our front-end, our web server, and database, all working together.
- With end-to-end tests we create tests that automate that front-end.
- We write tests that manipulate the browser and exercise our application to prove that it's working.

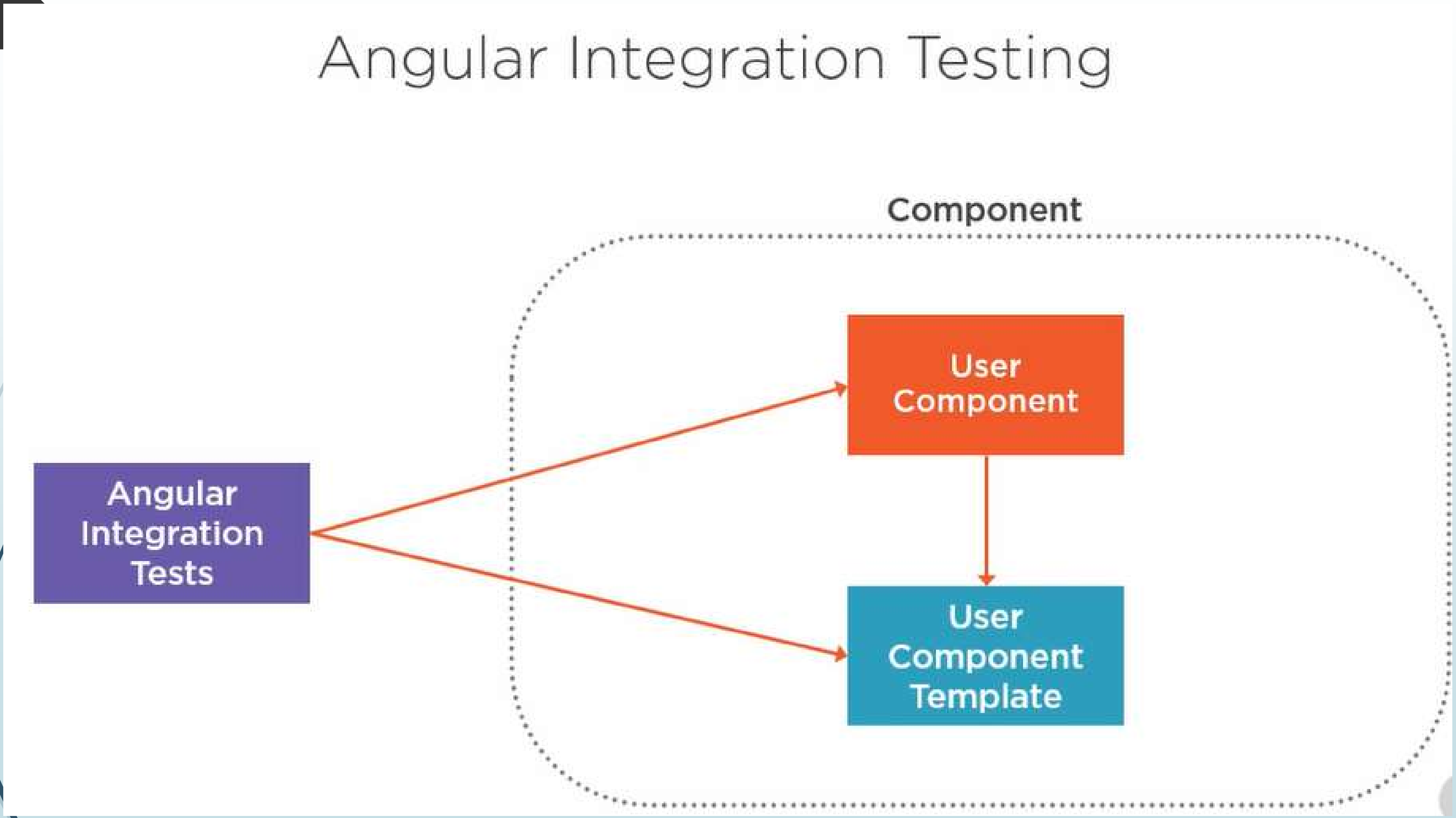


- Unit Testing
- With integrational and functional testing we've gone into our front-end and now we're using two different pieces of our front-end that are related and work together, but not necessarily the same unit.
- In the figure it's the user component and the user service.
- In an integration or functional test we might test that those two components work together correctly.
- To do that we write integration tests that use the user component, which also uses the live user service, and then we might check and see what kind of HTTP calls the user service is making or what values it returns back to the user component.

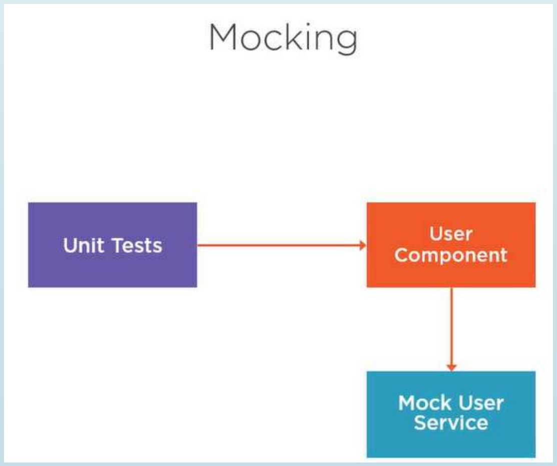


- Unit Testing
- In this case we just take a single unit of code, again our user component, and we write tests against that one unit of code.
- Unit testing is usually the most talked about kind of testing, and generally, in development, we write more unit tests than we write of any other kind of test.
- Again, there can be a little bit or argument about what is a unit.
- For example, if we had a user helper class we might consider that and the user component to be a single unit of code and we might test those together.
- Components in Angular have templates.
- So Angular has tooling that allows for a special kind of test, which they call an integration test.
- It uses the template and the component together to make sure that those two pieces are working correctly together.

- Unit Testing

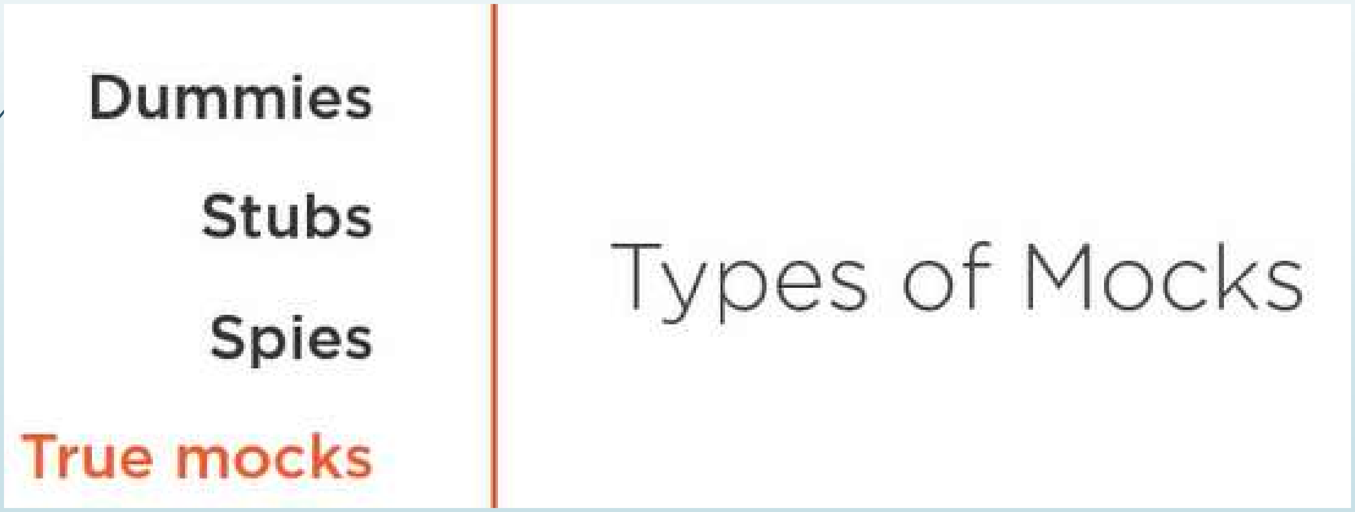


- Mocking
- One of the very important concepts in the unit testing is mocking.
- Mocking allows us to make sure that we are only testing a single unit of code at a time.
- For the most part, a class doesn't operate in isolation, most classes or components have dependencies.
- In this example a user component like this probably doesn't work by itself, it likely would use something like a user service, which is injected into the component.
- But when we write our unit tests we don't want to use the real user service, there are a lot of reasons for this.



- Mocking
- First, we're just trying to test the user component, it's a unit test.
- We don't want to test the user service as well, we'll write separate unit tests for that.
- Also, this user service might make HTTP calls, which we don't want in a unit test.
- So we need to draw a boundary around the unit that we're testing.
- Again, in this case, that's the user component.
- So we will do that with mocks.
- Instead of using the real user service we're going to provide a mock user service.

- Mocking
- A mock is a class that looks like the real class, but we can control what it does, what its methods return, and we can ask it questions about what methods were called during a test.
- Although most people just use the generic term mock, there are actually several types of objects that do various things related to mocking.



- Mocking
- A mock is a class that looks like the real class, but we can control what it does, what its methods return, and we can ask it questions about what methods were called during a test. Although most people just use the generic term mock, there are actually several types of objects that do various things related to mocking.
- The simplest kind is a dummy.
- **Dummies** are just objects that fill a place.
- They generally don't do much interesting, they're just used in place of a real object, like if a method call requires a parameter that's an object, but it doesn't care what that object is then a dummy is the perfect thing for that.
- A **stub** is an object that has controllable behavior.
- If we call a certain method on a stub we can decide in our test what value that method call will return.

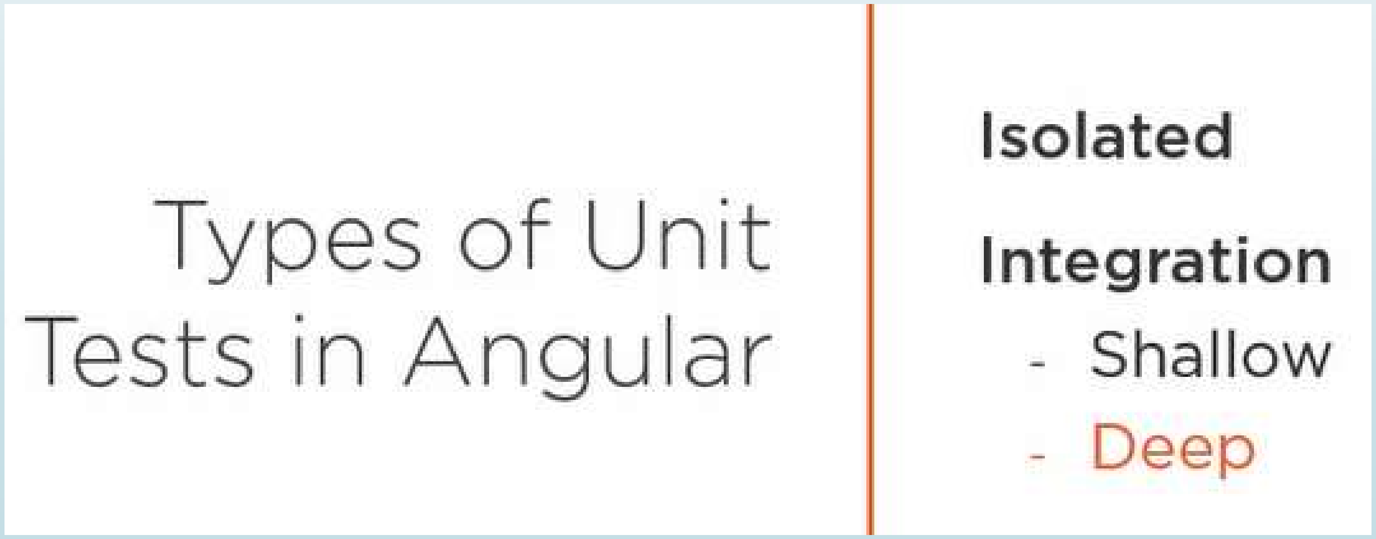
- Mocking

- A **spy** is an object that keeps track of which of its methods were called, and how many times they were called, and what parameters were used for each call.
- Most of the time, these are the types of objects that we use when we need a mock, and many times the boundaries between these three can be a little blurred.
- We might use objects that have the behavior of both stubs and spies for example.
- But there is one other type of object that we can use, a true mock.
- These are more complex objects that verify that they were used in exactly a specific way.
- For example, they can check that only a specific method was called, and that is was called only once, and it had some very specific parameters, and they're able to do this to themselves.

- Mocking
- True Mocks are a bit more difficult to work with and are usually overkill for what most unit tests need, but we will see an example of a kind of true mock when we get into testing components that use HTTP.

- Unit Testing In Angular

- Angular has several different kinds of unit tests, and it's important to know what each of them are and how they differ from each other.
- The basic unit test is an **<u>isolated test</u>**.
- This is what we think of when we think of a unit test.
- In an isolated test we simply exercise a single unit of code, either the class of a component, or the class of a service, or a pipe, we construct that class by hand, and we give it its construction parameters ourselves.


Types of Unit Tests in Angular

Isolated

Integration
- Shallow
- Deep

- Unit Testing In Angular
- An integration test is a bit more complex.
- In an Angular integration test we actually create a module in which we put just the code that we're going to test, generally just one component, but we actually test that in the context of an Angular module.
- This is used so that we can test the component with its template.
- There are two types of integration tests supported in Angular, shallow integration tests where we only test a single component, and deep integration tests, the difference being that many components actually have child components.
- Sometimes, we want to test both the parent component and the child component and how they work together and that is a deep integration test.

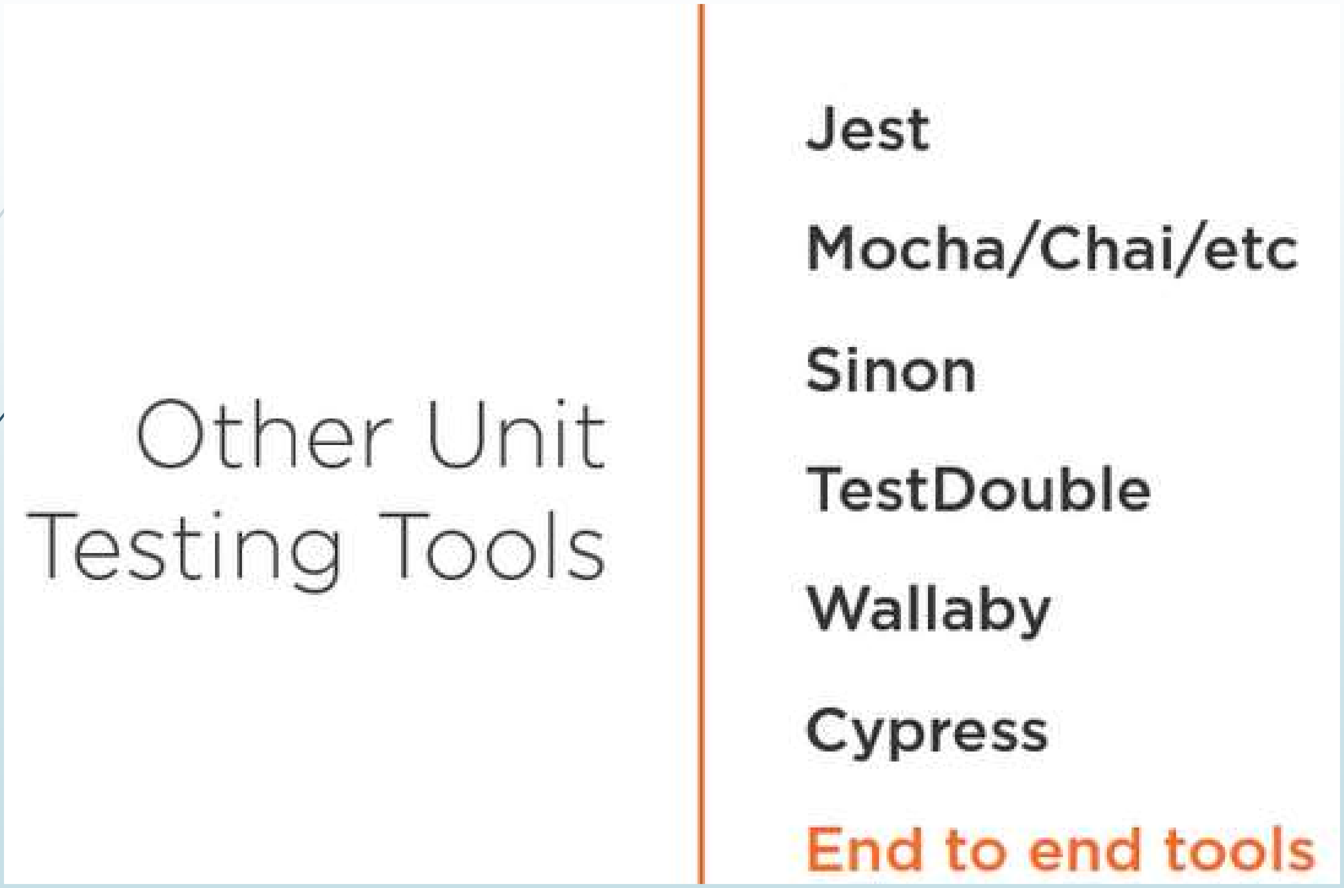
- Tools of Unit Testing with Angular
- the default tools that are used when testing an Angular application are the CLI sets up testing for us and it uses two different tools,
- **Karma**, which is the test runner, this is what actually executes our tests in a browser,
- and **Jasmine.**
- Jasmine is the tool we use to create mocks, and it's the tool that we use to make sure that the tests work the way that we want them to using expectations.

Tools We Will Use

Karma

Jasmine

- Other Testing tools



Other Unit Testing Tools

Jest

Mocha/Chai/etc

Sinon

TestDouble

Wallaby

Cypress

End to end tools

- Other Testing tools

- There are quite a few other unit testing tools that are available for unit testing with Angular.
- There's a very popular library called **Jest** that's been put out by facebook and it's really popular with other frameworks but can be used with Angular.
- There's **Mocha and Chai**, those are replacements for Jasmine.
- They are somewhat popular and they're easy to drop in and replace Jasmine with.
- There's **Sinon**, which is a specialized mocking library.
- If we find that the mocking capabilities in Jasmine aren't good enough then we can use something like Sinon.
- There's **TestDouble**, which is a competitor to Sinon, that's gaining some popularity, but is still far less popular than Sinon

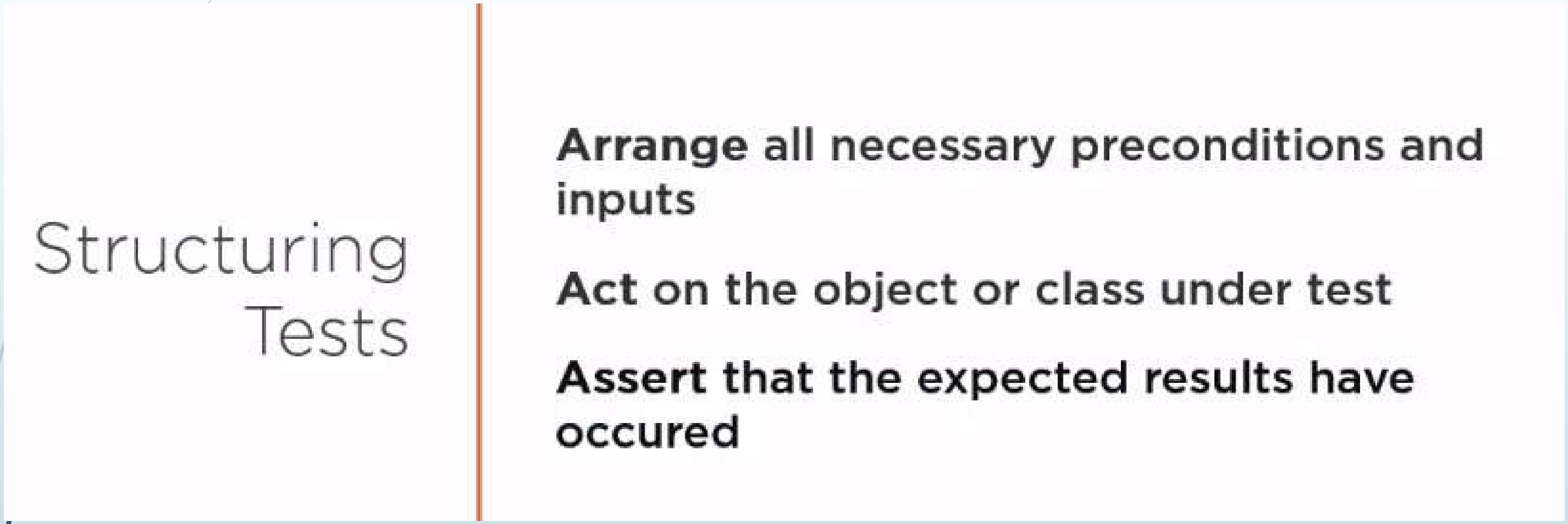
- Other Testing Tools
- .
- There's a tool called **Wallaby**, this is a paid tool that allows you to see the code coverage of your tests right in your IDE.
- It's very convenient and it's getting to be a popular tool.
- **Cypress** is traditionally considered to be an end-to-end testing tool, but they are developing capabilities to do more integration types of testing.
- In the future we may see Cypress become more popular with Angular.
- Finally, there are tons of end-to-end testing tools, again, this session is not about end-to-end tests, so we won't be dealing with them, but it is important to know that if we are going to write end-to-end tests with Angular we have a lot of choices.

- Isolated Unit Tests
-
- Testing a class as just a piece of JavaScript code is what isolated testing is.
- We're going to isolate a class, we're going to test it as a regular old piece of JavaScript, and the tools and techniques that we use are the same things that we would use to test plain old JavaScript code.
- It's also the same tools and techniques that are used in testing a lot of other JavaScript frameworks.

- Unit Testing – Writing Good Tests

- This is the type of subject that can be debated for hours, and hours, and there are volumes of blogs and books written on the subject.
- There are a few principles of writing good units tests that are generally agreed upon by the programming community at large.
- **First,** it's important to know how to structure a test.
- Structuring tests follows what's called the AAA pattern.
- First, we arrange all necessary preconditions and inputs, then, we act on the object or class under test, and finally, we assert that the expected results have occurred.
- In a good test we actually see the code doing these things in this order.
- First, we set up whatever we need to set up, then we make a change, and then we check that the change happened in the way that we expected.
- Another way that we look at this is that we set an initial state, we change the state, and then we check to make sure that the new state is correct.

- Unit Testing – Writing Good Tests



Structuring Tests

Arrange all necessary preconditions and inputs

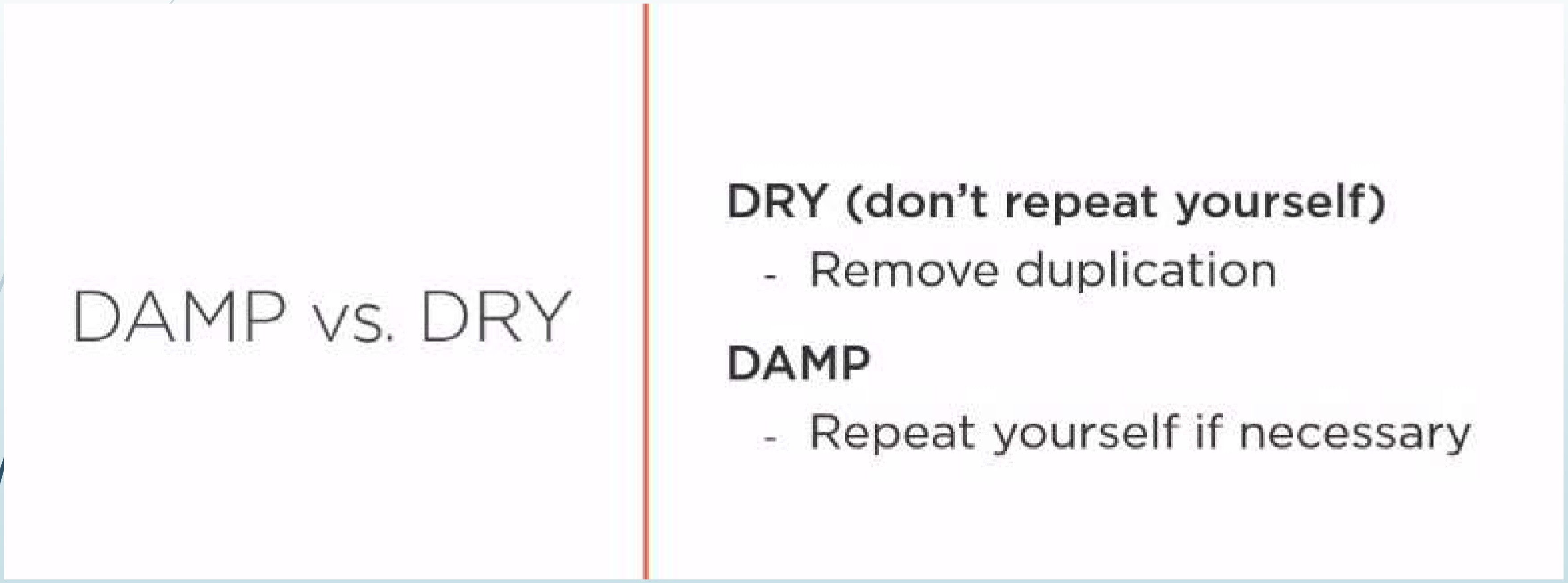
Act on the object or class under test

Assert that the expected results have occured

- Unit Testing – Writing Good Tests

- There's also a concept in testing of DAMP versus DRY.
- DRY, or don't repeat yourself, is a common concept used in programming.
- When we're following the DRY principle we remove duplication from our code, we don't want any duplication of code in our application.
- Good tests though, operate under a different principle called the DAMP principle.
- The reason we call it DAMP is because we still want to mostly follow the DRY principle, but we will repeat ourselves if necessary.
- This is good because a good test should tell a story.
- The story is we start at a given place, we make a change, and we check that we arrived where we got.
- That complete story should be within the it function.
- We shouldn't need to look around a whole lot in order to understand what's going on in the test, or in order to understand the story.

- Unit Testing



DAMP vs. DRY

DRY (don't repeat yourself)

- Remove duplication

DAMP

- Repeat yourself if necessary

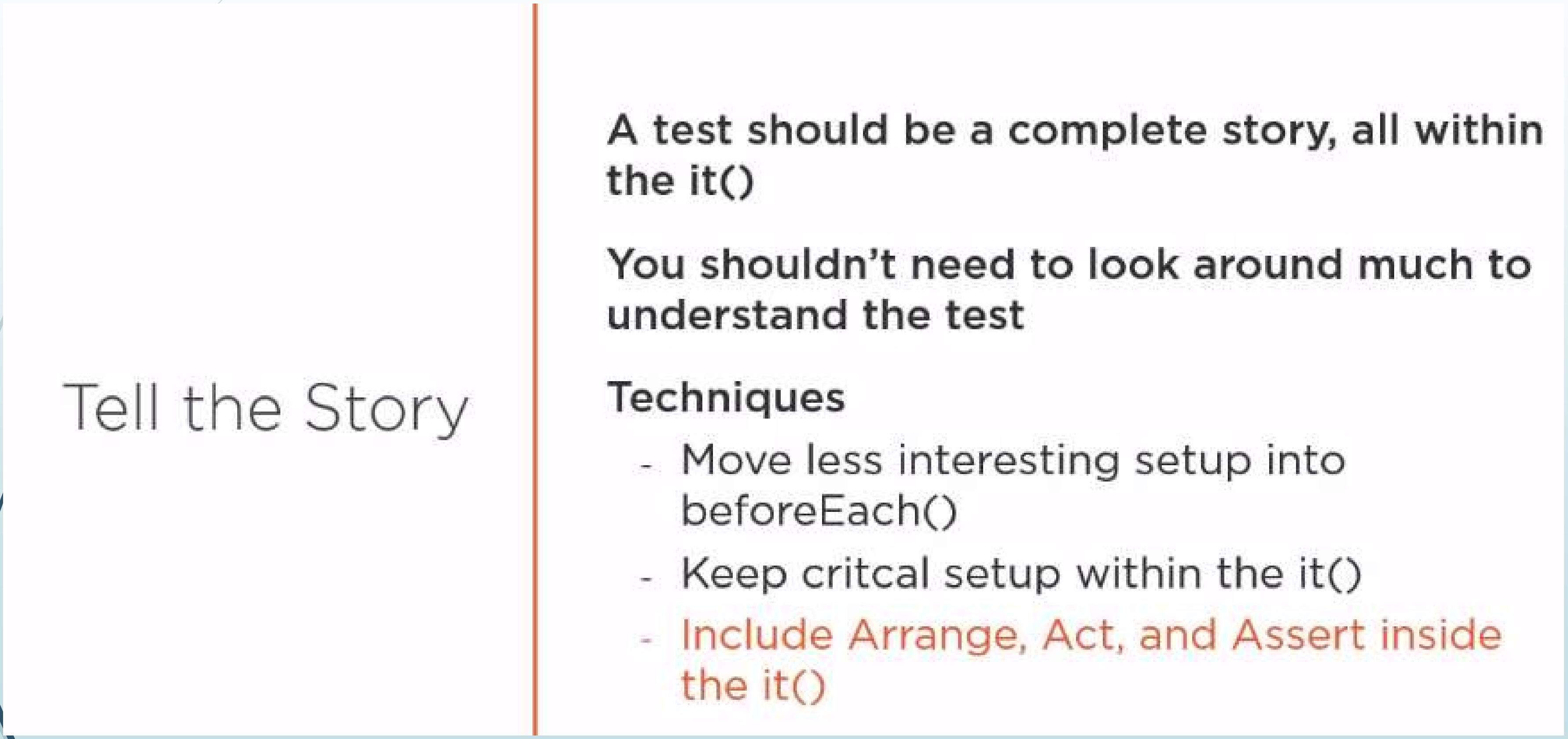
- Unit Testing – Writing Good Tests

- So there are some techniques that we can use in order to tell our story effectively.
- First, we should move less-interesting setup into our beforeEach.
- If there's some setup or initial state that needs to be there, but isn't critical for the test that we're creating, then we can move that setup into our beforeEach function.
- Critical setup though should be within the it function.
- So if we have two different tests that use the same piece of setup, but that setup is important to the story of what the test is, then we will duplicate that setup within the it block, rather than extracting them both out into the beforeEach, which would remove duplication.
- This is why we call our tests DAMP and not DRY.

- Unit Testing – Writing Good Tests

- Finally, we want to make sure that we include the arrange, the act, and the assert inside of the it function as they are possible to do.
- Sometime we test the initial state of our code, so sometimes the arrange might be missing, but in general, we try to include all three pieces inside of our it function and not move those out into a common beforeEach.
- Following these principles will help us write better tests, but of course, there is plenty of art to writing good tests and there is no replacement for experience.

- Unit Testing – Writing Good Tests



Tell the Story

A test should be a complete story, all within the it()

You shouldn't need to look around much to understand the test

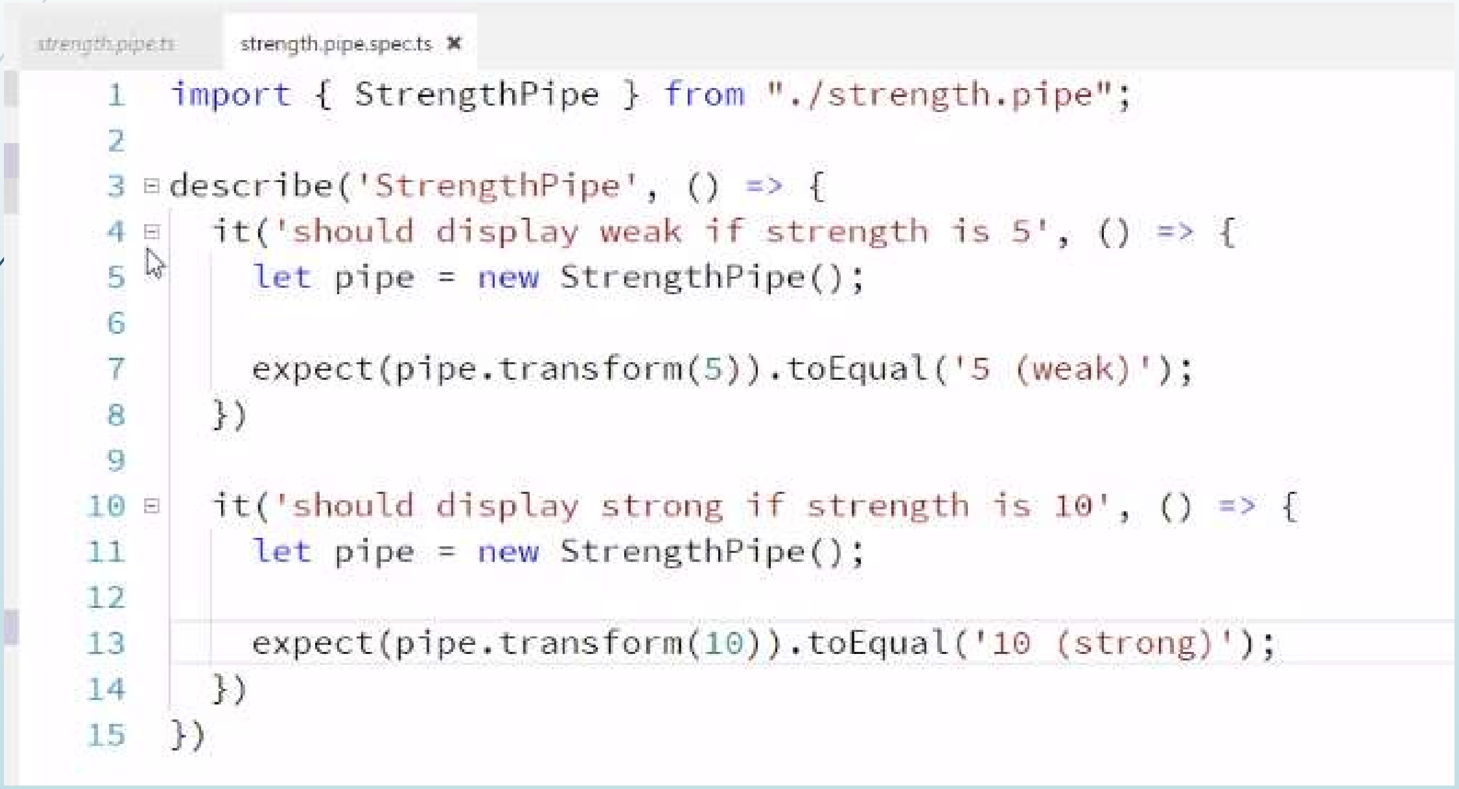
Techniques

- Move less interesting setup into beforeEach()
- Keep critcal setup within the it()
- Include Arrange, Act, and Assert inside the it()

- Unit Testing – Testing a Pipe
- We are going to start by testing this StrengthPipe.
- The reason we're going to do this first is because this is an extremely simple piece of code.
- The StrengthPipe class has no dependencies, it doesn't have a constructor that receives any injected parameters, and it's only got one method, the transform method.
- The transform method is a simple, stateless method, it takes in a number and returns a string.
- In this case, if the number is less than 10 it returns the value as a string plus the word weak.
- If it's between 10 and less than 20, it's the same thing, but the word strong in parentheses, and if it's greater than 20 it's the word unbelievable.
- To create a test for this we'll create a new file, and we're going to create it right alongside that StrengthPipe, and we'll call this strength. pipe. spec. ts.
- It's customary with tests to write the spec file using the same exact name, but just adding the word s-p-e-c as another piece of the test file.

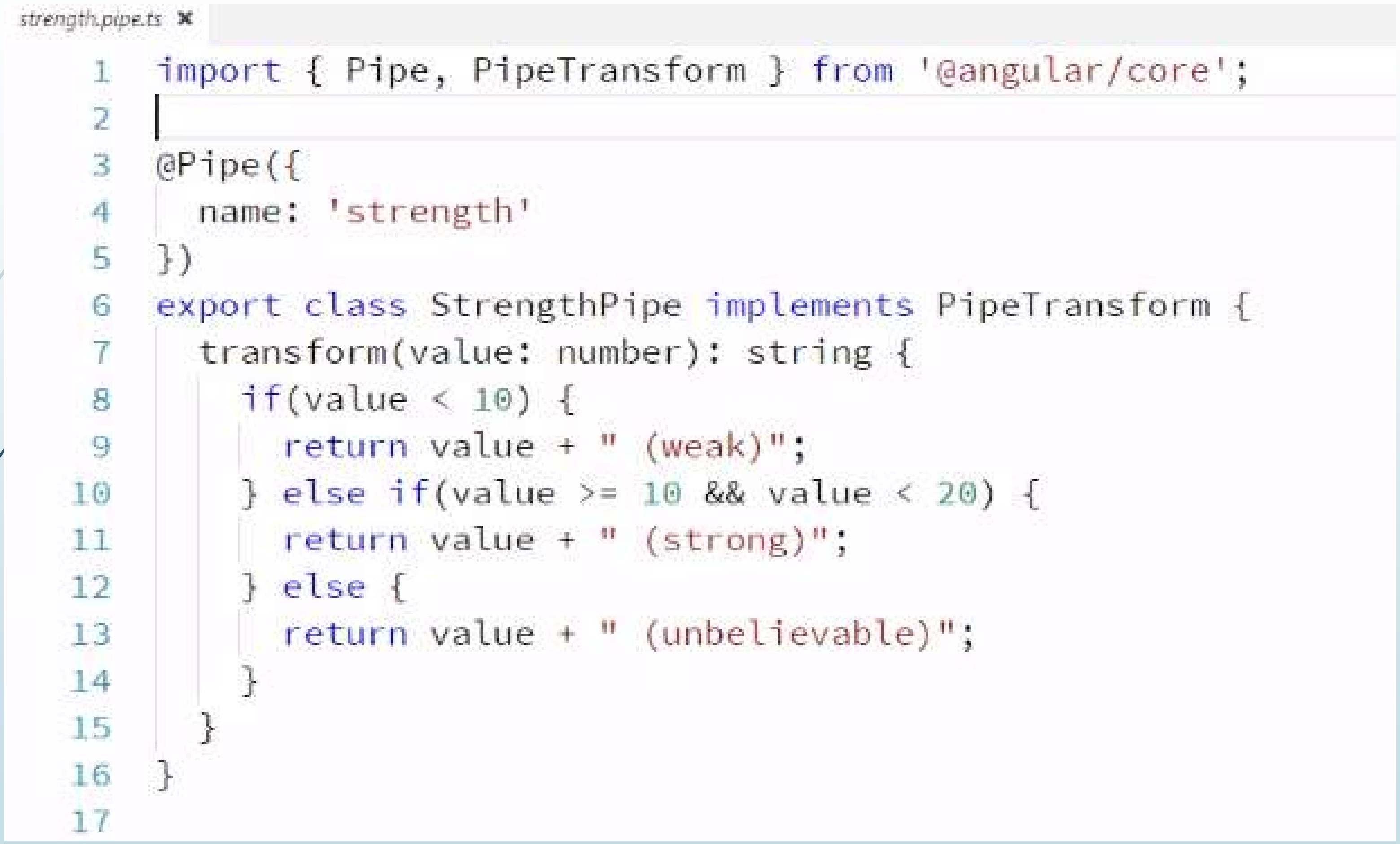
- Unit Testing – Testing a Pipe

- It's also customary to put it into the same directory.
- That way it's easy to see whether or not a piece of code has a test for it, and if not we can then write tests for it if we wish.



```typescript
import { StrengthPipe } from "./strength.pipe";

describe('StrengthPipe', () => {
  it('should display weak if strength is 5', () => {
    let pipe = new StrengthPipe();

    expect(pipe.transform(5)).toEqual('5 (weak)');
  })

  it('should display strong if strength is 10', () => {
    let pipe = new StrengthPipe();

    expect(pipe.transform(10)).toEqual('10 (strong)');
  })
})
```

• Unit Testing – Testing a Pipe


strength.pipe.ts ✖

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'strength'
})
export class StrengthPipe implements PipeTransform {
  transform(value: number): string {
    if(value < 10) {
      return value + " (weak)";
    } else if(value >= 10 && value < 20) {
      return value + " (strong)";
    } else {
      return value + " (unbelievable)";
    }
  }
}
```