# Types of Tests

[(https://github.com/learn-co-curriculum/js-tdd-types-of-tests)](https://github.com/learn-co-curriculum/js-tdd-types-of-tests) [(https://github.com/learn-co-curriculum/js-tdd-types-of-tests/issues/new)](https://github.com/learn-co-curriculum/js-tdd-types-of-tests/issues/new)

## Learning Goals

- Define different types of tests
- Understand how to determine what code to test

## Introduction

When building complex applications, one challenging decision you'll have to make is: "What should I test?"

In this lesson, we'll define some common categories of testing, and talk about where to devote your time when writing tests for your applications.

## Types of Tests

There are a number of different terms that roughly define the scope of what should be covered by a given test:

- **Static**: tests for typos in code and errors that can be detected without even running the application. You don't actually *write* static tests: there are tools, such as **ESLint** (https://eslint.org/) and **Typescript** (https://www.typescriptlang.org/), that help perform *static analysis* on your code to catch these types of errors.
- **Unit**: tests for individual "units" (which could be an individual function, class, or component) in isolation from the rest of the application. The units being tested shouldn't have any dependencies outside themselves. Unit tests are written in a framework like Jest.
- **Integration**: testing multiple related units of the application and the interactions between them. Integration tests are written in a framework like Jest.
- **End-to-end**: sometimes also called "functional testing", this type of test works by running your application in an environment as close as possible to how a user would interact with it, and verifying that the application functions as expected. End-to-end tests need to be run by a special program that can actually run your code in a browser. **Playwright** (https://playwright.dev/) and **Cypress** (https://www.cypress.io/) are two of the most popular library for end-to-end testing.

There are also other testing definitions of tests beyond these four categories, but in general, these four categories are good ways to roughly think about different levels at which to test your application.

As you start writing tests, it's good to consider what kind of test you're setting out to write. The boundaries between integration and unit tests in particular can be challenging to identify at first. A good rule of thumb when you're determining what kind of test you're writing is to ask *whether the code you're testing depends on other parts of your application*. If it does, can it be refactored to be tested in isolation so that it can unit tested? If not, you'll be writing an integration test. Integration tests can be more challenging to write, but they also give more confidence that the application works as intended.

## Testing Philosophies

For many years, the prevailing wisdom was to focus mostly on writing unit tests. That is demonstrated in Martin Fowler's **Test Pyramid (https://martinfowler.com/bliki/TestPyramid.html)** diagram below (where "Service" and "UI" correspond to our definitions of "Integration" and "End-to-end"):

One big reason for this is that unit tests are relatively fast and easy to write, and also that they can be run quickly. Being able to run tests quickly becomes important once your application has hundreds of tests! Since TDD dictates running tests each time you write new code, having a slow test suite can make for a frustrating developer experience.

In recent years (particularly in the JavaScript community), more emphasis has been placed on the value of writing integration tests and spending less time on unit testing. One philosophy for testing that has been adopted as a replacement for the Test Pyramid is the Kent C. Dodds' **Test Trophy (https://kentcdodds.com/blog/the-testing-trophy-and-testing-classifications)**:

The idea behind this approach that the higher up you move in the trophy, the more confidence you'll have that that your tests are accurately representing the feature you're trying to build. The downside is that the higher up you move in the trophy, the more expensive and time consuming the tests are to run. So integration tests should be a good middle ground between confidence in your code and time needed to run the tests.

## Determining What To Test

Knowing how to write tests is important, but knowing *what* to test is equally crucial! How can we decide what we should be testing in our applications?

The most important question to answer when determining what to test is: **"What feature am I trying to build?"**

Approaching tests with this question in mind puts us in a good state of mind when we start writing tests. Good tests should specify the **behavior** of an application, not the **implementation**. When writing a test for a function, we shouldn't care what the code inside the function looks like: we should only be concerned with the function's behavior, i.e. "when given X input, it returns Y."

A good example of this approach is the test we wrote in the previous lesson. We started by *describing the feature we were trying to build*:

> Your users will fill out their profile information with what year they were born. You will subsequently need to display how old they are on their profile page.

We then used that to *design a specification for our code*, based on a **use case** we expect for our users:

> I expect calling the function currentAgeForBirthYear(1984) will return 38

We then wrote the tests that gave us confidence that the code we had written successfully implemented this feature:

```
describe("currentAgeForBirthYear", () => {
  it("returns the age of a person based on the year of birth", () => {
    const birthYear = 1984;
    const ageOfPerson = currentAgeForBirthYear(birthYear);
    expect(ageOfPerson).toBe(38);
  });
});
```

In this case, our test would be considered a **unit test**, since the unit we're testing ( `currentAgeForBirthYear` ) doesn't depend on any other code in our application in order to perform its functionality.

## Conclusion

Understanding the names of different types of tests and having a philosophy of knowing what type of test to write for a given feature is helpful, but like anything, it takes time and practice to get the balance right. For now, we'll give you some guidance as to the kind of tests you should be writing so that when you start building your own applications you'll have a set of examples to draw from.

In the next lesson, you'll get a chance to design your first unit test from scratch (and also implement the code to pass the test) by following a test-driven development workflow.

## Resources

- Martin Fowler: Test Pyramid (https://martinfowler.com/bliki/TestPyramid.html)
- Kent C. Dodds: Write tests. Not too many. Mostly Integration (https://kentcdodds.com/blog/write-tests)
- Kent C. Dodds: Test Trophy (https://kentcdodds.com/blog/the-testing-trophy-and-testing-classifications)