Intro to Test-Driven Development

 (https://github.com/learn-co-curriculum/js-tdd-intro-to-test-driven-development)  (https://github.com/learn-co-curriculum/js-tdd-intro-to-test-driven-development/issues/new)

Learning Goals

- Define test-driven development and explain its benefits
- Explain the steps to follow when writing code using test-driven development

Introduction

In this section, we'll be learning how to write tests and how to follow a practice known as **Test-Driven Development** when building applications. We'll start by learning the basic workflows for writing tests, and understanding some of the tools we JavaScript developers have at our disposal. Then we'll move onto more advanced concepts like testing React components and writing tests that work with network requests and APIs.

To get started, let's talk about the philosophy behind **Test-Driven Development**.

Define Test-Driven Development

Test-Driven Development (or **TDD**) is *a practice that developers use to build high quality programs and prevent errors from developing in their programs*.

Following test-driven development means that *before a single line of code gets written, developers write tests first*.

The number one reason this is an important practice is because tests minimize the amount of errors and bugs in the code. As much as we'd like to be perfect developers and write bug-free code, the reality is that programming is hard, and even the most experienced programmers don't write flawless code on their first attempt.

Why Use Test-Driven Development?

It seems counterintuitive to write tests first, but there are a few reasons why developers work this way:

1. **Design:** TDD force you to think about the design of your code before you write it. It's easy to jump right into implementing the features of your application, but stopping to think about how exactly you want your code to function makes for well-designed and thoughtful programs.
2. **Discipline:** If you don't write tests first, you might never get around to writing them period. It might feel like a chore to write tests first, but you'll be glad at the end of the day that you have a full suite of tests to keep your code running smoothly and bug-free.
3. **Less Work:** By repeating the process of 1: write test, 2: write code to make the test pass, developers end up writing only the code they need. It's an arduous process, so superfluous code comes with a lot of baggage (more tests!). Thus, developers ultimately only write the code that's wholly necessary.
4. **Confidence in Code:** Having test coverage for your application means that developers can have more confidence when making changes to the application, since they'll be able to run tests and validate that any newly added code doesn't break existing tests.
5. **Documentation:** Well-written tests can serve as a form of documentation as to how code is intended to work. If your tests are written like "stories" that describe the behavior of your code, the tests themselves can be a helpful resource for new developers to understand an application's code base.

How Test-Driven Development Works

The typical test development cycle can be summed up with "red, green, refactor."

1. Red: write a test to describe a small feature you want to build. When you run the test, it'll fail. Having a failing test at first is an important step in the process, since it will force you to **think** about the feature you're trying to build before writing the code for it.
2. Green: Write *just enough* code to make the test pass. Now when you run the test, it should pass. Don't worry about writing perfect code at this stage. Just think about the least amount of code required to make the tests pass.
3. Refactor: Make that code as succinct, clear, and DRY as possible. Refactor as necessary and re-test your code.

Then, rinse and repeat for each new feature. That way if new code fails some tests, you can either undo the most recent change or debug your code by narrowing down the surface area in your code base where the error occurred based on your tests.

Conclusion

In the next lesson, we'll break down an example of a basic test and follow a TDD approach to writing code. At first, following TDD can slow you down. It takes time and practice to perfect the art of writing tests. Give yourself time to adjust!

Resources

- **The Art of Agile Development: Test-Driven Development (http://www.jamesshore.com/v2/books/aoad1/test_driven_development)**