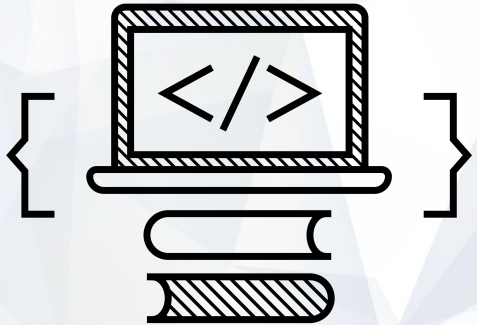
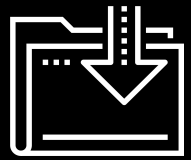


Behavior-Driven Development

Skills Bootcamp in Front-End Web Development

Lesson 14.2



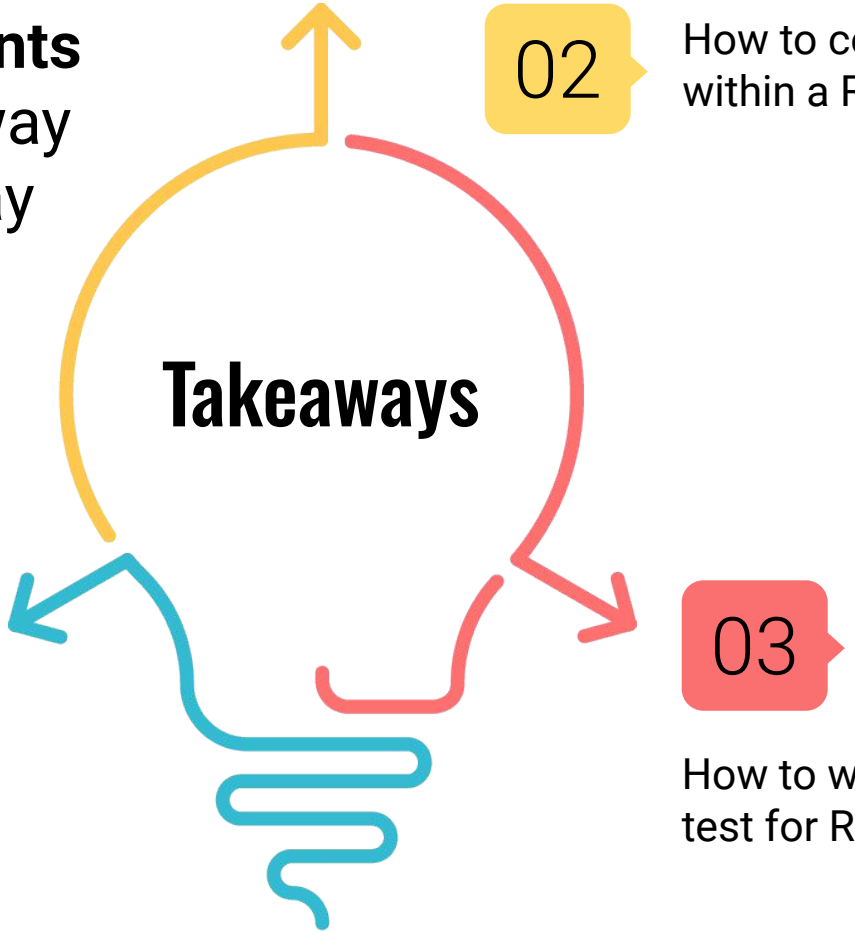
WELCOME

Learning Objectives

By the end of class, you will be able to:

- Compare and contrast BDD and TDD.
- Utilize fundamental concepts of BDD to approach testing React components with React Testing Library and Vitest.
- Explain the difference between Vitest and React Testing Library.

The three main points you should walk away understanding today are:



02

How to configure RTL within a React app.

01

BDD is the better option in comparison to TDD.

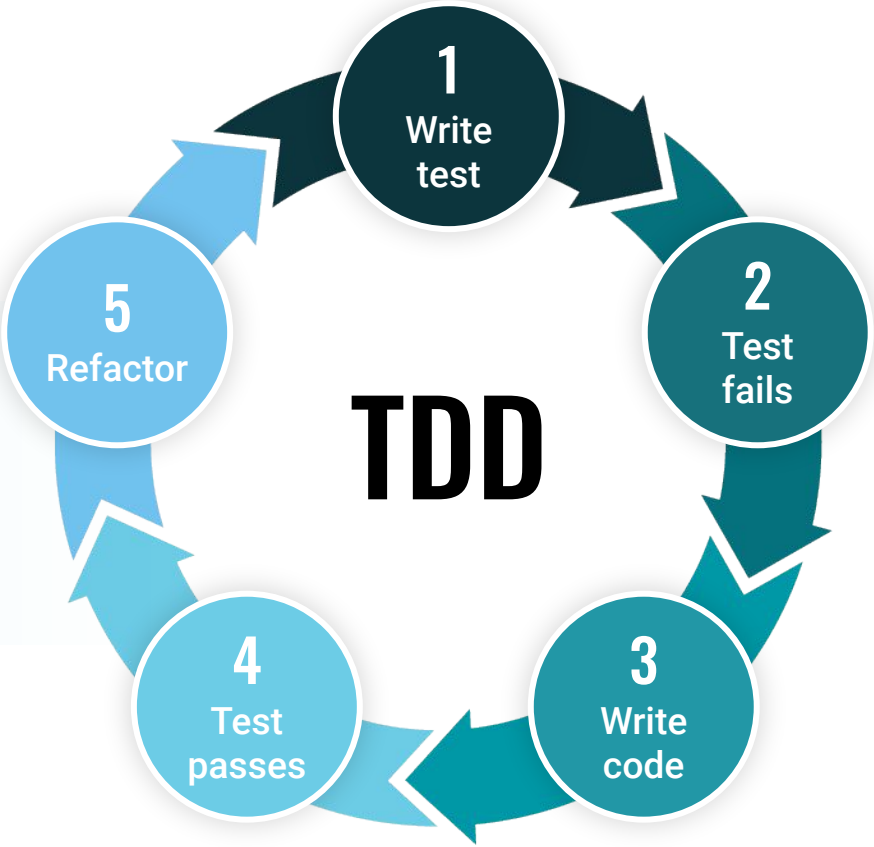
03

How to write a basic RTL test for React components.

TDD: Test-Driven Development

What Is Test-Driven Development (TDD)?

TDD is the practice of developing software by first writing a test and then writing code that passes said test.



Test-Driven Development (TDD) in Action

Steps of TDD:

01 Write the failing test.

02 Write the minimum amount of code to have for the test to pass.

03 Further enhance the tests for the code you just wrote.

04 Refactor the code to make the new tests pass.

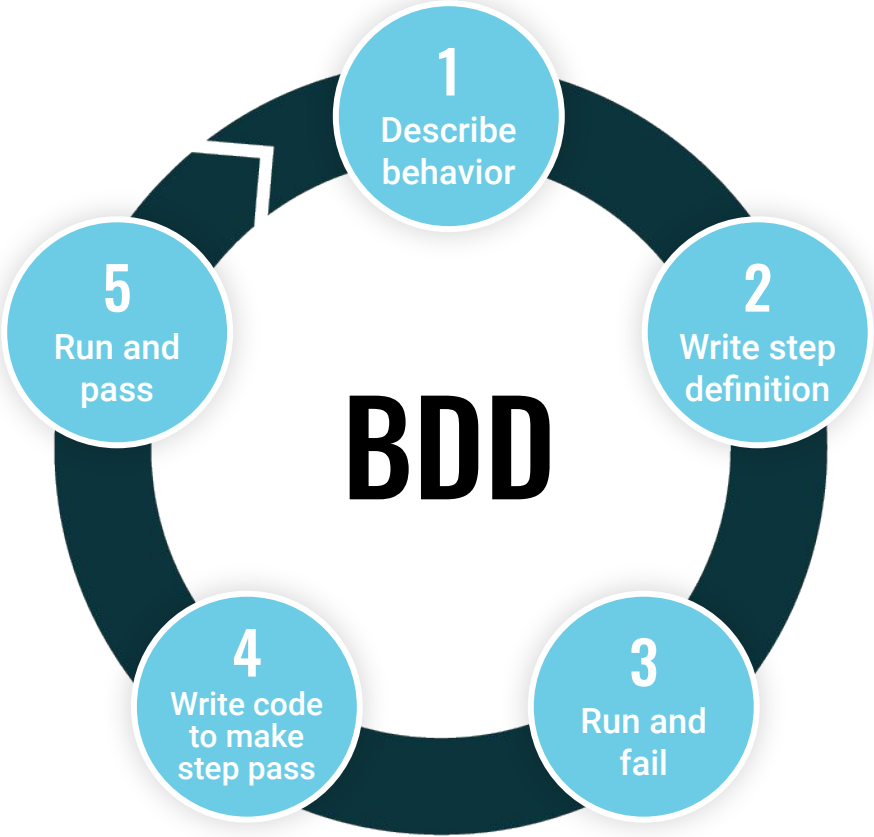
05 Iterate between adding tests and refactoring code until the desired functionally and code quality is achieved.

BDD: Behavior-Driven Development

What Is Behavior-Driven Development?

BDD is an evolution of TDD.

Whereas in TDD we write a test based on user requirements, in BDD we test to make sure that it is specific.



Behavior-Driven Development (BDD) in Action

Steps of BDD:

01 Describe behavior.

02 Write step definition.

03 Run and fail.

04 Write code to make step pass.

05 Run and pass.

TDD vs. BDD

TDD vs. BDD

While BDD and TDD sound very similar, there are a few defining factors that make them different:

01 TDD is a development practice, while BDD is a team methodology.

02 In TDD, developers write the tests.

03 In BDD, the automated specifications are created by users or testers.

04 For small, co-located, developer-centric teams, TDD and BDD are effectively the same. For a much more detailed discussion, InfoQ sponsored a virtual panel on the topic.

TDD vs. BDD

TDD	BDD
TDD is a development practice that focuses on **implementation of a feature**.	BDD is a team methodology that focuses on the **application behavior**.
In TDD, developers write the tests.	In BDD, the test are written by developers, customers, and/or QAs.
TDD's main focus is **unit testing**.	BDD's main focus is **testing application requirements**.

For small development teams, TDD and BDD are effectively the same thing.

Vitest

Vitest (vee-test) is a **JavaScript test runner and library** that can be utilized for creating, running, and structuring tests.

What Is Vitest?

We will be using it mostly for its test “runner” functionality.

Vitest has many benefits, including:

- It is compatible with Vite.
- It supports many JS testing conventions.
- It has great documentation.
- It is very easy to get up and running.

What Is Vitest?

Vitest's `test runner` allows you to have a dynamic environment for watching your test as you develop your code.

The Vitest methods we will be using in this lesson are as follows:

`it():`

A method that runs a test. It is an alias for the test() method.

`expect():`

Allows you to test a value based on providing arguments.

Vitest Test Runner

The file(s) from which the tests are being run

Number of passing and failing tests and test suites

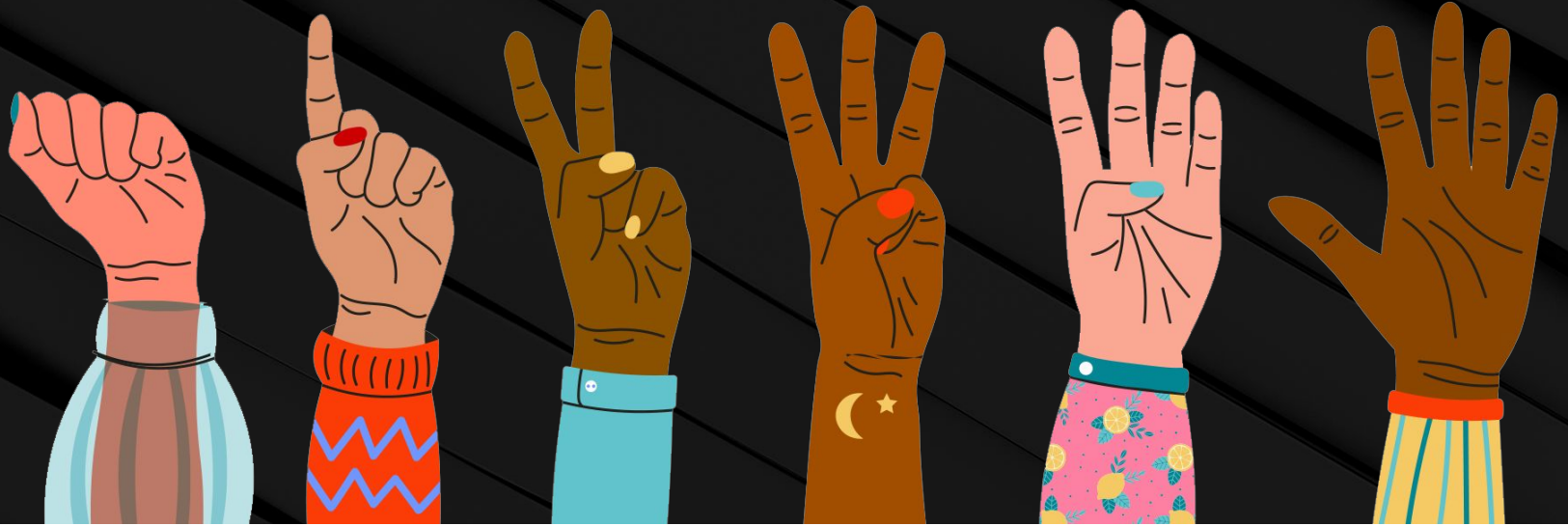
```
PASS  src/tests/App.test.jsx
  ✓ renders learn react link (31 ms)

Test Suites:  1 passed, 1 total
Tests:        1 passed, 1 total
Snapshots:    0 total
Time:         2.508 s
Ran all test suites.

Watch Usage: Press w to show more.
```

FIST TO FIVE:

How comfortable do you feel with the concepts we covered for Vitest?

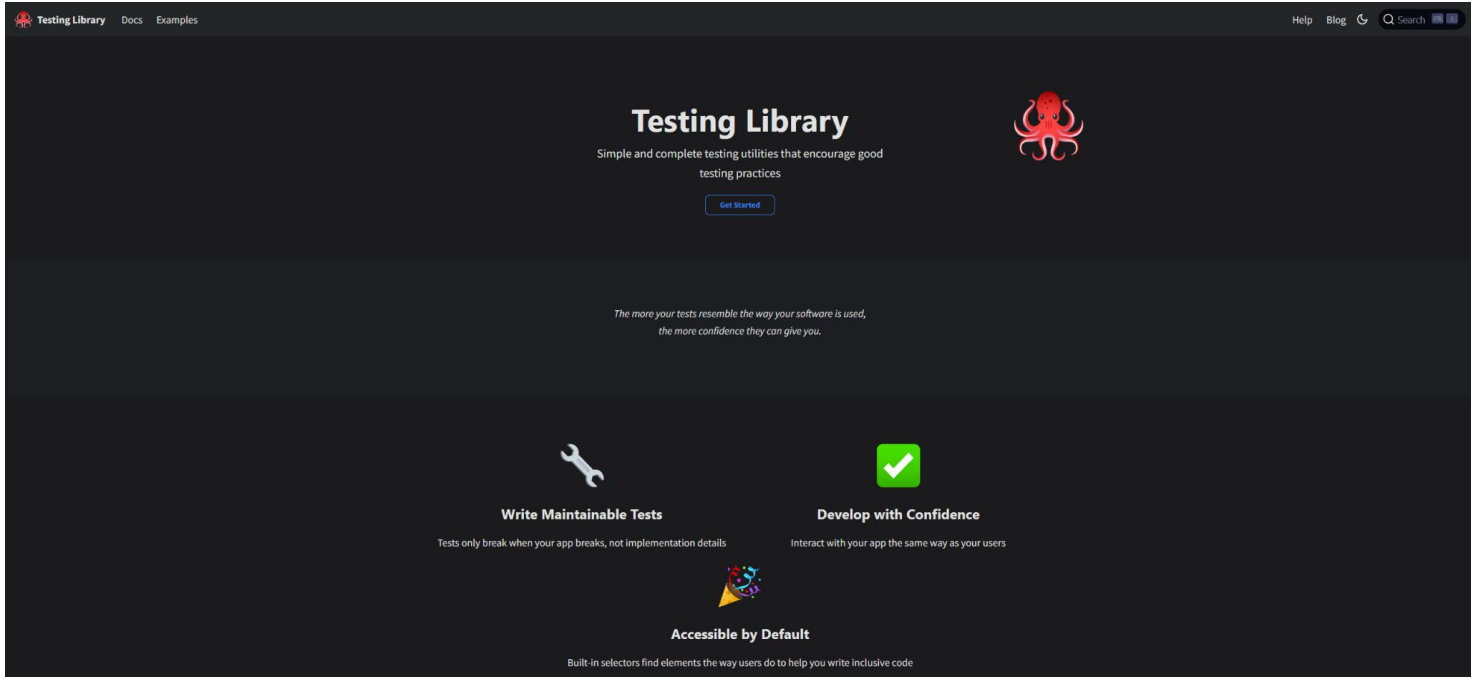


React Testing Library

RTL is a **testing utility for React** created by Kent C. Dodds and other contributors.

What Is React Testing Library?

In short, it is a utility that we can use to write a test for a React component.
We use `RTL` to write our tests and `Vitest` to run them and display their results.



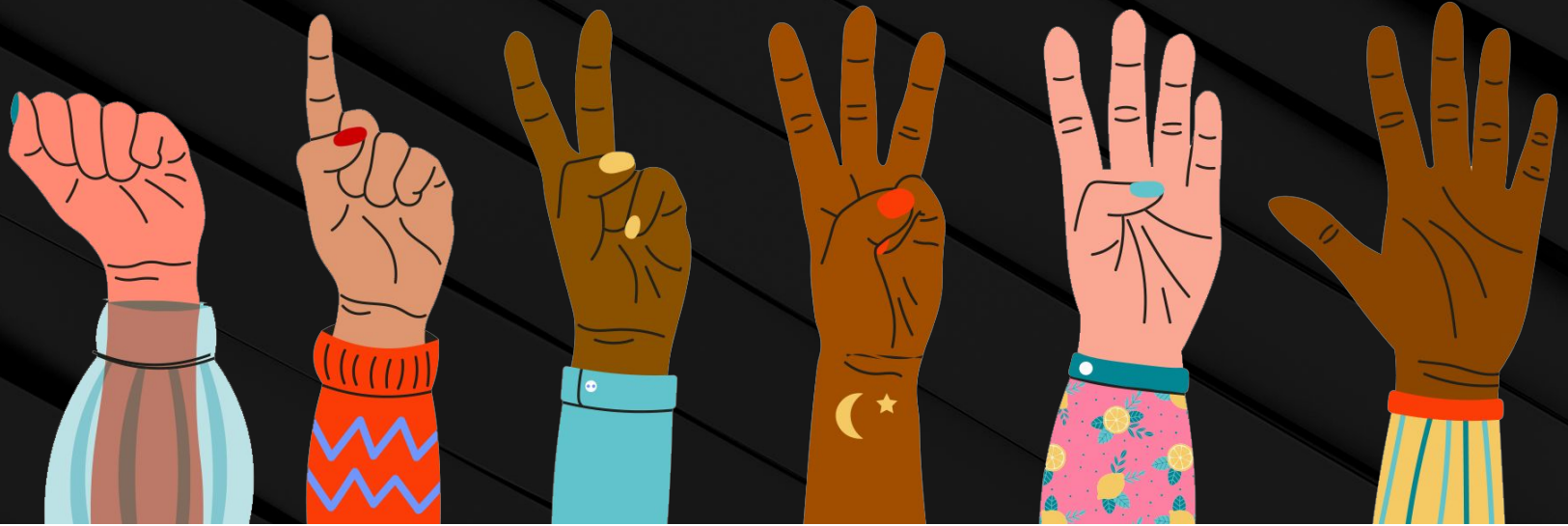
RTL Methods

The RTL methods we will be using in this lesson are as follows:

`getByRole()`	Finds all nodes in the rendered DOM tree that match the accessibility role passed as the argument.
`getByText()`	Finds all nodes in the rendered DOM tree that contain the text content passed as the argument.
`render()`	Renders the passed DOM element or React component into `document.body`.
`userEvent()`	Simulates an event on the DOM element upon which it has been invoked.

FIST TO FIVE:

How comfortable do you feel with the concepts we covered for RTL?



Partner Activity: TDD vs. BDD

With a partner, you will discuss the differences between BDD and TDD.

Suggested Time:

15 minutes

Questions?

Time to Code

Demo Install Vitest

Suggested Time:

10 Minutes

Instructions: Demo Install Vitest with RTL

Follow the instructions in the file provided to:

- Create a React project with `Vite`.
- Install `Vitest`.
- Configure `Vitest` with a setup file.

Questions?

Time to Code

Configure RTL Within a Vite React Project

Suggested Time:

15 Minutes

Instructions: Creating a React Project with Vite

Follow the instructions in `03-We-ConfigureRTL\README.md` to configure `RTL` and its sub-dependencies within your React project.

Questions?

Instructor Demonstration

Initial Testing

Activity: Initial Testing

In this activity, you will complete each prompt listed in `05-Stu-InitialTesting\README.md`.

Suggested Time:

15 minutes

Before You Get Started...

`Name`

Is the method that should be the name that is used as an argument for the test.

`Test cases`

Are the requirements for that specific test.

Time’s Up! Let’s Review.


Break

Activity: Component Buildout Part 1

In this activity, you will will develop code in your `App.jsx` file so that the RTL tests written in `App.test.jsx` all pass.

Suggested Time:

10 minutes

Time’s Up! Let’s Review.

Questions?

Activity: Component Buildout Part 2

In this activity, you will continue to develop code in your `App.jsx` file so that the RTL tests written in `App.test.jsx` all pass.

Suggested Time:

15 minutes

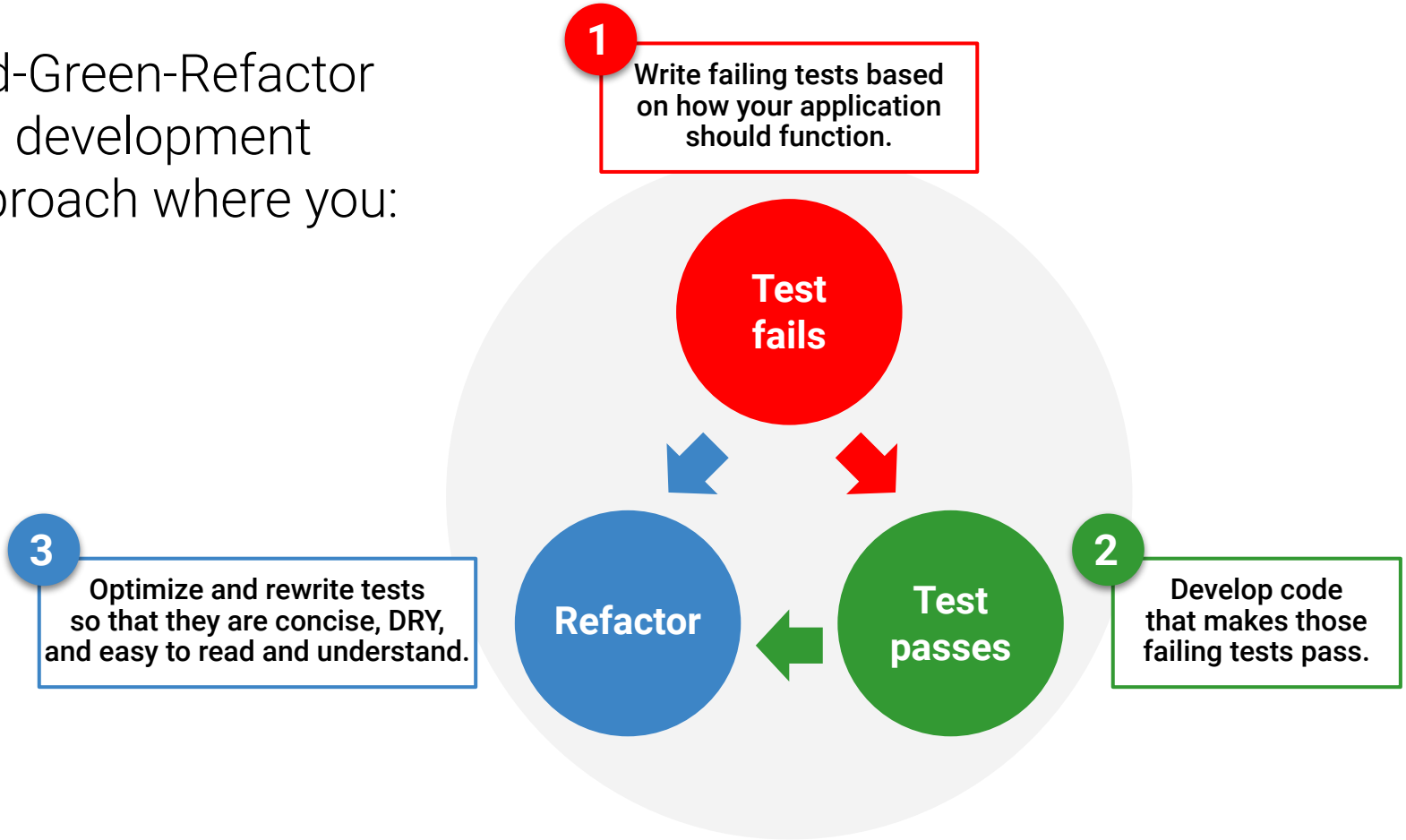
Time’s Up! Let’s Review.

Questions?

Red-Green-Refactor

What is Red-Green-Refactor?

Red-Green-Refactor is a development approach where you:



Refactor App.test.jsx:
Create Custom Helper Functions

Refactor App.test.jsx: Create Custom Helper Functions

Inside of `src/tests/App.test.jsx`, create the following functions that meet the criteria provided:

01

`renderApp():`

This function should accept no arguments, render the app, and return an object that contains helper methods for each element we wish to test.

02

`get[Element]():`

Within the return object, for each element used in your tests, create a function that will use RTL's `screen.getBy()` methods to return the element.

ex: `getIncrementBtn()`

Refactor App.test.jsx: Create Custom Helper Functions

Prompt 1:

`renderApp()`

This function should accept no arguments and return an object that will contain helper functions for each element we wish to test.

```
const renderApp = () => {
    render(<App />);

    return {
        // TODO: write a helper function for each element we wish to test
    };
}
```

Refactor App.test.jsx: Create Custom Helper Functions

Prompt 1: major takeaways

Our custom render function renders our app component and returns an object with test utility functions to help us DRY up our test code.

It allows us to optimize our code so that we do not have to instantiate an instance of the app component inside of every test, which in turn speeds up how much time it takes to run our test.

We are concerned about the time it takes to run our test because, in a large application, you can have hundreds, if not thousands, of tests. The more time we can save in running our test, the faster we can deploy our code.

Refactor App.test.jsx: Create Custom Helper Functions

Prompt 2:

`get[Element]()`

The object returned by the `renderApp()` function should return a method for each repeated element we'll want to include in our tests.

```
const renderApp = () => {
...
    return {
        getIncrementBtn: () => screen.getByRole('button', { name: 'increment counter' }),
        getDecrementBtn: () => ...,
        getErrorHeading: () => ...,
        getCountAtZero: () => ...,
        getCountAtOne: () => ...,
    };
}
```

Prompt 2: Major Takeaways

These functions are meant to speed up our coding process by giving us reusable functions to quickly get the elements on which we're making assertions in our tests.

We could even rewrite these functions to accept search criteria to make them less specific. In this case, we only have a handful of elements to test.

Activity: Refactor App.test.jsx: Create Custom Helper Functions

In this activity, you will reinforce and build upon the testing skills that you have learned so far by refactoring the code inside of `App.test.jsx` to contain the `renderApp` and find[Element] functions.

Suggested Time:

15 Minutes

Time’s Up! Let’s Review.

Questions?

The End