React.6

tests

- What is unit testing?

  Unit testing is a type of testing to check if the small piece of code is doing what it is suppose to do.

- What is integration testing?

  Integration testing is a type of testing to check if different pieces of the modules are working together.

- What are regression tests?

  Regression tests are performed whenever anything has been changed in the system in order to check that no new bugs have been introduced.

  This means you re-run your unit and intergration tests after all patches, upgrades, and bug fixes

- What are acceptance tests?

  Acceptance tests make sure a feature or use case is correctly implemented. It is similar to an integration test, but with a focus on the use case rather than on the components involved

- What are functional tests?

  In functional testing, a tester isn't concerned with the actual code, rather he/she need to verify the output based on given the user requirements with the expected output.

- What is Jest?

Jest is the testing framework that Facebook puts out.

▼ Jest is built on top of what?

Jasmine

▼ Along with Jest you can use what instead of Enzyme?

@testing-library/react, formerly called react-testing-library

▼ What is @testing-library/react, formerly called react-testing-library?

is a tool that has a bunch of convenience features that make testing React significantly easier and is now the recommended way of testing React

▼ .babelrc for tests:

This is just saying when we run Jest (it runs with NODE_ENV in test mode by default, hence the env name) to transform the code to work for Node.js instead of the browser.

```
{
  "presets": [
    [
      "@babel/preset-react",
      {
        "runtime": "automatic"
      }
    ],
    "@babel/preset-env"
  ],
  "plugins": ["@babel/plugin-proposal-class-properties"],
  "env": {
    "test": {
      "presets": [
        [
          "@babel/preset-env",
          {
            "targets": {
              "node": "current"
            }
          }
        ]
      ]
    }
  }
}
```

▼ package.json for tests:

```
"test": "jest",
"test:watch": "jest --watch"
```

▼ A methodology for testing React

- Try to test functionality, not implementation. Make your tests interact with components as a user would, not as a developer would. This means you're trying to do more think of things like "what would a user see" or "if a user clicks a button a modal comes up."
- In general when I encounter a bug that is important for me to go back and fix, I'll write a test that would have caught that bug. Actually what I'll do is before I fix it, I'll write the test that fails.
- Ask yourself what's important about your app and spend your time testing that. Ask yourself "if a user couldn't do X then the app is worthless" sort of questions and test those more thoroughly.

▼ What should you test instead of implementation?

functionality

▼ test file example

Pet.test.js or Pet.spec.js in __ tests __ dir

```
test("displays a non-default thumbnail", async () => {
  const pet = render(
    <StaticRouter>
      <Pet images={["1.jpg", "2.jpg", "3.jpg"]} />
    </StaticRouter>
  );

  const petThumbnail = await pet.findByTestId("thumbnail");
  expect(petThumbnail.src).toContain("1.jpg");
});
```

▼ testing UI interactions example

user story: if a user clicks a thumbnail they expect to see the hero image change to that

In Carousel.js add the following `data-testid`s.

```
// to the hero image
data-testid="hero"

// to the thumbnail
data-testid={`thumbnail${index}`}
```

```
// Carousel.test.js
import { expect, test } from "@jest/globals";
import { render } from "@testing-library/react";
import Carousel from "../Carousel.js";

test("lets users click on thumbnails to make them the hero", async () => {
  const images = ["0.jpg", "1.jpg", "2.jpg", "3.jpg"];
  const carousel = render(<Carousel images={images} />);

  const hero = await carousel.findByTestId("hero");
  expect(hero.src).toContain(images[0]);

  for (let i = 0; i < images.length; i++) {
    const image = images[i];

    const thumb = await carousel.findByTestId(`thumbnail${i}`);
    thumb.click();

    expect(hero.src).toContain(image);
    expect(thumb.classList).toContain("active");
  }
});
```

▼ How to test custom hooks?

Testing custom hooks is a bit of a trick because they are inherently tied to the internal workings of React: they can't be called outside of a component. So how we do we get around that? We fake a component.

```
import { expect, test } from "@jest/globals";
import { render } from "@testing-library/react";
import { renderHook } from "@testing-library/react-hooks";
import useBreedList from "../useBreedList.js";

test("gives an empty list with no animal", async () => {
```

```
  const { result } = renderHook(() => useBreedList(""));

  const [breedList, status] = result.current;

  expect(breedList).toHaveLength(0);
  expect(status).toBe("unloaded");
});
```

▼ How to test API calls with mocks?

we don't actually want to fetch from our API. This can be slow and cause unnecessary load on a server or unnecessary complexity of spinning up a testing API. We can instead mock the call. A mock is a fake implementation. We could write our own fake fetch but a good one already exists for Jest called jest-fetch-mock so let's install that. Run npm install -D jest-fetch-mock@3.0.3.

We now need to make it so Jest implements this mock before we run our tests. We can make it run a set up script by putting this in our package.json:

```
{
  "jest": {
    "automock": false,
    "setupFiles": ["./src/setupJest.js"]
  }
}
```

Then let's make a file in src called setupJest.js.

```
import { enableFetchMocks } from "jest-fetch-mock";

enableFetchMocks();
```

Now it will fake all calls to fetch and we can provide fake API responses.

add to useBreedList.test.js:

```
test("gives back breeds with an animal", async () => {
  const breeds = [
    "Havanese",
    "Bichon Frise",
    "Poodle",
```

```
    "Maltese",
    "Golden Retriever",
    "Labrador",
    "Husky",
  ];
  fetch.mockResponseOnce(
    JSON.stringify({
      animal: "dog",
      breeds,
    })
  );
  const { result, waitForNextUpdate } = renderHook(() => useBreedList("dog"));

  await waitForNextUpdate();

  const [breedList, status] = result.current;
  expect(status).toBe("loaded");
  expect(breedList).toEqual(breeds);
});
```

The waitForNextUpdate allows us to sit back and wait for all of React's machinery to churn through the updates, effects, etc. until our data is ready for us to check on.

In general you should mock API calls. It will make tests run much faster and save unnecessary load on an API.

▼ What is snapshot testing?

With more-or-less a single line of code you can assert: this code doesn't break, and it isn't changing over time.

▼ How to snapshot test?

First we need to grab a test renderer so we can render out these snapshots. The React team makes an official one so grab it here: `npm install -D react-test-renderer@17.0.1`.

Let's test Results.js. It's a pretty stable component that doesn't do a lot. A low cost, low confidence test could fit here. Make a file called Results.test.js

```
import { expect, test } from "@jest/globals";
import { create } from "react-test-renderer";
import Results from "../Results";

test("renders correctly with no pets", () => {
  const tree = create(<Results pets={[]} />).toJSON();
```

```
  expect(tree).toMatchSnapshot();
});
```

Run this to see Jest say it created a snapshot. Go look now at Results.test.js.snap to see what it created. You can see it's just rendering out what it would look like.

Now if you modify Pet.js (that has its own tests already) your Results.js test is going to fail. Let's do a shallow render.

```
// top
import { createRenderer } from "react-test-renderer/shallow";

// replace second test
test("renders correctly with some pets", () => {
  const r = createRenderer();
  r.render(<Results pets={pets} />);
  expect(r.getRenderOutput()).toMatchSnapshot();
});
```

Now notice the snapshot renders `<Pet />` with their props rather than the actual rendered HTML. This is preferable because now Pet can shift (and you test Pet to have confidence in that component) with raising a false alarm in Results.

Update your snapshots by either running `npm run test -- -u` or you can use the watcher to do it with either `u` to update all at once or do `i` one-by-one.

You should commit snapshot files to git.

▼ What is Istanbul?

Jest has built into it: Istanbul.

a tool which tells you how much of your code that you're covering with tests

▼ How to run test coverage tests (with Jest)?

Add the following command to your npm scripts: "test:coverage": "jest --coverage" and go ahead run npm run test:coverage and open the following file in your browser: open coverage/lcov-report/index.html.

Add coverage/ to your .gitignore since this shouldn't be checked in.