Express Testing Practices

Download Demo Code

Goals

- Revisit some essential concepts with testing
- Understand what mocking is
- Examine end to end tests with Cypress

Good Testing Practices

- Make sure you write tests!
- Don't get too attached to coverage percentages
- Make sure in your readme you specify how to run the tests!

Seeing tests in action

```
/** POST /cats - create cat from data; return `{cat: cat}` */

describe("POST /cats", function () {
  test("Creates a new cat", async function () {
    const response = await request(app)
      .post(`/cats`)
      .send({
        name: "Ezra"
      });
    expect(response.statusCode).toBe(201);
    expect(response.body).toEqual({
      cat: { name: "Ezra" }
    });
  });
});
```

- We're not testing if we actually created anything!
- How should we test this? What do we test?

One option

```
/** POST /cats - create cat from data; return `{cat: cat}` */

describe("POST /cats", function () {
  test("Creates a new cat", async function () {
    const response = await request(app)
      .post(`/cats`)
      .send({
        name: "Ezra"
      });
    expect(response.statusCode).toBe(201);
    expect(response.body).toEqual({
      cat: { name: "Ezra" }
    });

    const catsQuery = await db.query("SELECT name FROM cats;")
    expect(catsQuery.rows[0]).toEqual({ name:"Ezra" });
    expect(catsQuery.rows).toHaveLength(1);
  });
});
```

A better way to test

```
/** POST /cats - create cat from data; return `{cat: cat}` */

describe("POST /cats", async function () {
  test("Creates a new cat", async function () {
    const response = await request(app)
      .post(`/cats`)
      .send({
        name: "Ezra"
      });
    expect(response.statusCode).toBe(201);
    expect(response.body).toEqual({
      cat: { name: "Ezra" }
    });

    const getCatsResponse = await request(app).get(`/cats`)
    expect(response.body[0]).toEqual({ name:"Ezra" });
    expect(response.body).toHaveLength(1);
  });
});
```

- Instead of testing the database, test the API
- Stay consistent with what you are testing

Test Driven Development

- Write tests **first** - they will fail!
- Only write the code necessary to get the tests to pass
- Focus on completing the task/user story at hand
- As you write more code, keep running tests and make sure they are passing

Red, Green, Refactor

- Your tests fail (**red**)
- You write the code to get the tests to pass (green)
- You refactor!

Mocking

When testing, you will commonly hear the term "mocking."

- Mocking is primarily used in unit testing
- An object under test may have dependencies on other (complex) objects
- To isolate the behavior, you replace other objects by mocks that simulate their behavior
- This is useful if the real objects are impractical to incorporate into the unit test.

Advantages of mocking

- It can be faster.
  - You don't have to wait for an API response
  - You don't have to deal with rate limits.
- It makes your tests 'pure'. Whether they fail or pass depends only on your code, not on anything externally built.

Challenges with mocking

- It sometimes requires a convoluted setup
- It is not always necessary and can be an over-optimization

Mocking with Jest

- There are quite a few libraries used for mocking, including **sinon**
- Jest comes in the with ability to mock functions
- https://jestjs.io/docs/en/mock-functions.html

An example

demo/mocking-demo/dice.js

```
function rollDice(numSides) {
  return Math.floor(Math.random() * numSides);
}

module.exports = rollDice;
```

Our tests

demo/mocking-demo/dice.test.js

```
const rollDice = require("./dice");

describe("rollDice", function() {
  Math.random = jest.fn(() => 0.5);

  test("it rolls the correct amount of dice", function() {
    expect(rollDice(6)).toEqual(3);
    expect(Math.random).toHaveBeenCalled();

    expect(rollDice(2)).toEqual(1);
    expect(Math.random).toHaveBeenCalled();
  });
});
```

What kinds of things can you mock?

- AJAX requests
- Reading/Writing to files
- impure functions like Math.random

Continuous Integration (CI)

Continuous Integration is the practice of merging in small code changes frequently, rather than merging in a large change at the end of a development cycle.

- The goal is to build better software by developing and testing in smaller increments.

What can CI do for you?

- Automate running your tests when pushing your code
- Reject deployments if your tests do not pass
- Easily notify you when changes to your test suite occur

How does it work?

- It integrates with tools like GitHub and carries out a series of tasks to build and test your code
- If one or more of those tasks fails, the build is considered broken
- If none of the tasks fail, the build is considered passed, and Travis CI can deploy your code

Common CI Tools

- Travis CI
- Jenkins
- Circle CI
- Buddy

Using Travis CI

Imagine we have the following code:

demo/travis-ci-demo/operations.js

```
function add(a = 0, b = 0) {
  return a + b;
}

function average(...numbers) {
  let total = 0;
  if (numbers.length === 0) return 0;
  for (let num of numbers) {
    total += num;
  }
  return total / numbers.length;
}

module.exports = { add, average };
```

And the following tests

demo/travis-ci-demo/operations.test.js

```
const { add, average } = require("./operations");

describe("#add", function() {
  it("adds numbers", function() {
    expect(add(2, 2)).toEqual(4);
  });
  it("handles empty inputs", function() {
    expect(add()).toEqual(0);
  });
});

describe("#average", function() {
  it("calculates the average", function() {
    expect(average(2, 2)).toEqual(2);
    expect(average(2, -2)).toEqual(0);
  });
  it("handles empty inputs", function() {
    expect(average()).toEqual(0);
  });
});
```

Here's what a simple Travis config looks like

demo/travis-ci-demo/.travis.yml

```
language: node_js
node_js:
  - "lts"
script:
  - jest operations.test.js
```

Seeing it in action

https://app.travis-ci.com/github/rithmschool/travis-ci-demo/builds

End to End Tests

- End-to-end testing tests an application's flow from start to end.
- The purpose of E2E testing is to simulate an entire real user scenario.

Pros of E2E tests

- You are also going to find a lot more user-impacting bugs up front, because you are working directly with the application at the user's perspective.
- You don't have to be as familiar with the specific implementation, or even how coding works to write automated UI tests. Many tools allow you to just click record, perform some actions, and save a script.

Cons of E2E tests

- E2E tests are not nearly as maintainable as unit tests. They break easily when one feature changes.
- They are much more time consuming to write and can be handled by QA teams.

Common E2E Testing tools

- Selenium
- Cypress

An example with Cypress - Meme Generator!

demo/cypress-demo/cypress/integration/meme.spec.js

```
describe("Meme Generator", function() {
  beforeEach(function() {
    cy.visit("/index.html", { timeout: 5000 });
  });

  it("loads correctly", function() {
    cy.get("#meme-form").should("exist");
  });

  it("adds a meme when the form is submitted", function() {
    cy.get(".meme").should("not.exist");
    addMeme();
    cy.get(".meme").should("exist");
  });

  it("removes a meme when the meme is clicked", function() {
    addMeme();
    cy.get(".meme").click();
    cy.get(".meme").should("not.exist");
  });
});
```

Basic Cypress Setup

```
$ npm i --save-dev cypress
```

In ***package.json***:

```
"scripts": {
  "cypressopen": "cypress open"
},
```

For more, check out the docs!