07 - Cypress Testing Spike

COMP290 – Large Scale and Open Source Software Development
Dickinson College

Name:

Introduction:

In activities 02 (HTM) through 06 (FarmOS API) you have built a FarmData2 module that implements a simplified harvest report feature. In this activity you will learn about the Cypress testing framework and create some end-to-end (E2E) tests that exercise your harvest report.

Synchronizing with the Upstream:

1. Some time has passed since you created your fork and clone of the upstream FD2School-FarmData2 repository. It is possible that there have been updates to the upstream since you did so. So, as when working on any fork and clone it is important to synchronize your main branch with the upstream so that you have all of the recent changes.

a. Synchronize your local and origin FarmData2 repositories and your feature branch with the upstream. The steps you will need to do this include are:

1. Switch to the `main` branch.
2. Pull the `main` branch from the upstream.
3. Push the `main` branch to your origin.

If you don't remember the commands for this, you can refer back to the previous activity where you will have listed them.

b. Now because your `main` branch has been updated, you will need to merge those changes into the feature branch that you created in the prior activity. To merge the `main` branch into your feature branch:

1. Switch to your feature branch from the prior assignment (`<name>-06-API2`).
2. Merge the `main` branch into your feature branch.
3. Resolve any conflicts that arise (If you are just doing the FD2School activities there should not be any).

If you don't remember the commands for this, you can refer back to the previous activity where you will have listed them.

c. Your work on this assignment builds from what you did in the prior assignment. So you now need to:

1. Switch to your prior feature branch `<name>-06-API2` (if you are not there already).
2. Create a new feature branch named `<name>-07-E2E` from your prior feature branch.
3. Switch to your new feature branch.

You will do all of your work for this activity in this feature branch.

If you don't remember the commands for this, you can refer back to the previous activity where you will have listed them.

Getting Started with Cypress:

The Cypress project provides a good introductory tutorial. You can find this tutorial here:

- https://docs.cypress.io/guides/end-to-end-testing/writing-your-first-end-to-end-test

The activities in this section will help you to follow their tutorial to write your first E2E test using Cypress. This will give you a feel for how Cypress testing works and some practice with the Cypress application. Then in later sections you'll see how E2E tests are written in FarmData2.

2. The tutorial linked above begins with a link that describes how to install and open Cypress. You can ignore that link, because the FarmData2 development environment that you have been using already has Cypress installed. The following steps will get you to the place you need to be to start the tutorial:

a. Change into to the `FarmData2/farmdata2` directory. This directory should contain the file `test_runner.bash`. If it does not you are not in the correct location.

b. Use the following command to launch the Cypress test runner in the development environment.

```
./test_runner.bash
```

c. Click the E2E Testing box in the Cypress application.

d. Choose the "Electron" browser, if it is not already chosen.

e. Click the "Start E2E Testing in Electron" button.

3. The nest step in the tutorial shows a picture and says to use the "Create new empty spec button" button to create a new empty spec. However, FarmData2 is using a newer version of Cypress than was used when writing the tutorial. Instead, you will see a big blue "+ New Spec" button in the upper right.

a. Click the "+ New Spec" button.

b. Now you will see the "Create new empty spec button" shown in the tutorial. Click it.

c. Change the name of your spike to be:

`farmdata2/farmdata2_modules/fd2_school/first.spec.js`

d. Click the X to close the dialog rather than the blue "Okay, run the spec" button.

e. Now scroll down through all of the other tests (`.spec.js` files) that already exist in FarmData2. When you find the `fd2_school` folder you should also see your new `first.spec.js` file.

f. If you click on that file, it will run the "empty" spec that you created. It turns out that the spec created isn't exactly empty. It actually visits the page http://example.cypress.io. But that's all it does. Note: The tutorial says this test might fail, but it will pass and that is okay.

4. At this point you are up to the "*Write your first test*" heading in the Cypress tutorial. From here on you can follow the tutorial as it is written.

Complete the tutorial up to but not including the section with the header "*Record Tests with Cypress Studio*." As you do be sure to notice that:

- Every time you save your `first.spec.js` file, the tests are automatically rerun in the Cypress Application.
- You can rerun a test manually, and watch it run in the browser, by clicking the re-run button () in the testing window.

5. Make sure that you have your feature branch checked out. Commit your new `first.spec.js` file to your feature branch with a meaningful commit message that describes what you have done and push it to your origin.

6. On GitHub create a *Draft Pull Request* for your new feature branch to the upstream FD2School-FarmData2 repository. Be sure to link your PR to the issue for this activity by including a line like the following in the body of your PR:

Addresses #??

Replace the ?? by the issue number in the FD2School-FarmData2 issue tracker for this assignment.

Adding a New FarmData2 School Sub-tab:

With a little Cypress experience you are now ready to begin working on some E2E Cypress tests using your harvest report.

7. Make sure you have your feature branch checked out and then add another new sub-tab named **e2e** to the FD2 School tab. Have the contents of this new tab be contained in the directory cypress with the page content provided by the file `e2e.html`. Make a copy of your `api2.html` file into a file named `e2e/e2e.html`. Don't forget to clear the Drupal cache when you are done. The result should be that you now have four sub-tabs in the FD2 School tab: HTML, Vue1, Vue2, API, API2 and e2e. For now, your API2 and e2e tabs will be exactly the same. You'll be modifying and extending the e2e tab throughout this activity.

8. Commit your changes to your feature branch with a meaningful commit message that describes what you have done and push it to your origin to update your PR.

Testing the Harvest Report Header - Spike 1:

9. In the *Cypress Writing Your First E2E Test* tutorial they gave a list of four steps that map to the general structure of a Cypress test. What are those four steps?

Our Cypress tests in FarmData2 will generally follow those steps, but in order to access the FarmData2 pages it is necessary to log into the system. The general structure of a FarmData2 test will be:

1. Log in as the appropriate user (guest, worker, manager admin).
2. Visit the specific FarmData2 page that is to be tested.
3. Simulate user interaction with the page:
    - Get element(s) on the page.
    - Interact with the element(s).
4. Verify correct behavior:
    - Get elements(s) on the page.
    - Make *assertions* about the elments.

The first set of tests that we will create will verify that the Harvest Report page has the correct default content when it is loaded. Well just to keep things manageable we will check some of

it, not all of it. Tests like these can be useful for ensuring that the desired initial state of the application does not change without being noticed. For example, in FarmData2 the default start date for reports is supposed to be the first day of the current year. However, a developer might change that while working on a new feature just to save the effort of changing each time they try the new feature. That would be fine, but it would be a problem if they then forget to set it back to what it is supposed to be. A set of tests that check the initial state of the application would catch this type of change. These tests will typically include steps 1, 2, and 5 from the above process, as they are testing the initial state and no user interaction is required.

10. In FarmData2 each test file is named beginning with the name of the sub-tab containing the feature that is being testing (e.g. e2e) and ending with the suffix `.spec.js`. There will often be multiple test files for each sub-tab, and each file will contain a set of related tests. To distinguish these files, their names will contain additional words separated by dots (`.`) that give more information about what is being tested by the file.

Based on the naming convention just described, indicate which of the following would be acceptable names for E2E test files for the `e2e` sub-tab.

E2E Test Filename	Valid (Yes/No)?
`e2e.defaults.spec.js`	
`e2e.crops.filtering.js`	
`api.cropmap.spec.js`	
`e2e.report.spec.js`	

11. Since we'll be testing the default values on your `e2e` sub-tab, create a new file named `e2e.defaults.spec.js` in your `e2e` folder.

12. Add the following basic template to your `e2e.defaults.spec.js` file:

```
describe("Test the harvest report default values", () => {
    beforeEach(() => {
        cy.login("manager1", "farmdata2")
        cy.visit("/farm/fd2-school/e2e")
    })

    it("Check the page header", () => {

    })
})
```

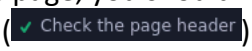
Most E2E tests in FarmData2 will have this general form with these parts:

`describe`: Encapsulates all of the tests in this file. The string gives a short but general description of what is being tested by all of the tests in this file.

`it`: each `it` defines one test. A describe can contain many separate `it`s. The string gives a short but more precise description of what is being tested by this specific test. This test, once we complete it, is going to check that the header of the page says "Harvest Report".

`beforeEach`: defines a function that is run *before each* `it`. Here the `beforeEach` contains code that logs into FarmData2 as the user `manager1` and then visits the `e2e` page that we are testing.

13. Open the Cypress application (if it is not already open) and run the `spec` file that you just created. When it runs, you will see the following in the test runner:

- In the browser window, you will see the test log in to FarmData and visit your `e2e` sub-tab.
- In the middle pane, showing the results of the test:
  - You will see the string's from your describe and it statements.
  - Below the string for the it, you will see the results of the BEFORE EACH and the TEST BODY.
    - There you will see the login, the visit and xhr (Axios) requests for the farm_crops and possibly the farm_areas (if you populated the areas dropdown from the database in activity 05).
  - Because the test doesn't yet make any assertions about the page, you should also see a green check mark indicating that the test passed (

).

14. Now to complete this test we will need to get the page header and check that it contains the correct text (i.e. "Harvest Report"). We'll break this into multiple steps. In this question we'll learn how to get a specific HTML element that we are interested in a Cypress test. Then in the next question we'll see how we can test the contents of (i.e. make *assertions* about) an element.

a. The easiest and the best way to get a specific element in a Cypress test is to add a special attribute to the HTML tag for the element you want. This attribute has the name `data-cy` and its value will be a name that you use to refer to that element in your test. Here we want the element that contains the page heading:

```
<h1>Harvest Report</h1>
```

Find this heading in your `e2e.html` file and a `data-cy` attribute to it so that it looks as follows:

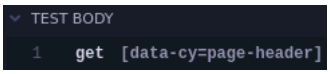
```
<h1 data-cy="page-header">Harvest Report</h1>
```

This essentially gives this HTML element a name that we'll be able to use to get it in our test.

b. Cypress' `cy.get` function gives us a way to get an element using its `data-cy` attribute. Add the following line to the body of the `it` in your `spec` file:

```
cy.get("[data-cy=page-header]")
```

c. Save your `spec` file and run it in Cypress. Now under TEST BODY you should see:



Because the test passed, this means that Cypress was successful in finding the HTML element with the `data-cy` attribute with the value `page-header`.

d. Change the `cy.get` function in your test so that it looks for an HML element with a `data-cy` attribute that has the value `pg-header`. What happens when you run the test now? Why?

e. Fix the error that you introduced in part d.

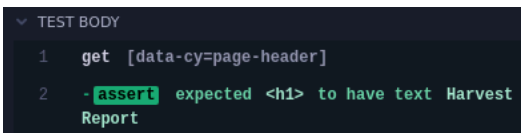
15. Now that we have the HTML element that we want to test, we can use Cypress' `should` function to make an assertion about it.

a. Change the line containing your `cy.get` statement to look as follows:

```
cy.get("[data-cy=page-header]")
    .should("have.text","Harvest Report")
```

Here we use the assertion `"have.text"` in the `should` function to test that the element has the specified text ("Harvest Report"). The `"have.text"` assertion is one of the most commonly used assertions in FarmData2 testing. But you will see as you continue through this activity that there are quite a few more.

b. Save your spec file and look at the results in Cypress. Remember that Cypress will run a `spec` automatically when it is saved. Now under TEST BODY you should see:



Because the assert is green and because the test passed, we know that the `<h1>` element with the `data-cy` attribute with the value `page-header` actually had the value "Harvest Report".

c. Change the value in your should statement to "Harvest Rpt". How does Cypress indicate that the test has failed now? What specific information does it give you that would be helpful in fixing the page or the test?

d. Fix the error that you introduced in part c.

16. Congratulations! You have now written your first real FarmData2 E2E test using Cypress. Commit your changes to your feature branch with a meaningful commit message that describes what you have done and push it to your origin to update your PR.

17. As a little review, let's collect together some of the information that you learned in this section. This will also help you by placing all of that information in one place as you go forward and continue to add more Cypress knowledge. Fill in the right-hand column with the piece of information described in the left-hand column. The first one is done as an example.

The statement in a spec file that contains all of the tests.	**`describe`**
The statement in a spec file that defines code that runs before every test.	
The statement in a spec file that defines an individual test.	
HTML attribute used to name HTML elements for Cypress	
Cypress function that finds an HTML element in a page.	
Cypress function used to make assertions about HTML elements.	
Cypress assertion to check the text in an HTML element.	

Testing the Harvest Report Default Dates - Spike 2:

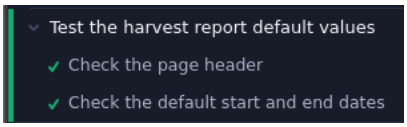
In this spike we'll create a test that checks that the default start and end dates for the harvest report are correct when the page is loaded.

18. Add `data-cy` attributes to the input elements for the start and dates so that you will be able to easily get them in a test. Be sure to use meaningful values for your `data-cy` attributes

(e.g. `start-date` is much better than `sd` or `start`). You may have noticed by now that the values for `data-cy` attributes always use *kabob-case*, where the words are separated using dashes (-). This is a common convention that is used and is followed by FarmData2.

19. Add a new empty `it` to your `spec` for this test. The new `it` must be contained within the `describe`, and it should have a string that is descriptive of what is being tested. You'll also need to be careful with the `()` and `{}` in the `it` to be sure it is correct. Run the `spec` in Cypress to check that your empty `it` is correct - if it is, the test will pass.

20. Notice that when you have multiple tests and they pass, Cypress will compress its output. So, when your new test passes you should see something like:



What happens if you click on the descriptive string for one of the tests?

21. Add `cy.get` statements to your it that get the HTML input elements for the start and end dates. Run the `spec` in Cypress to check that the test passes.

22. When testing the header, you used the `"have.text"` assertion to check that the <h1> element contained the text "Harvest Report". In this test you are working with HTML input elements. When you want to test the value of an HTML element you will use the `"have.value"` assertion.

Append `should` calls to your `cy.get` statements that will test that the start and end dates have the correct default values. Note that when you created the page you were asked to make the default start date 05/05/2020 and the default end date 05/15/2020.

Run the `spec` in Cypress. It will most likely fail. This is because the way that the HTML input element reports its value is different than how it displays it. Use the information provided about the failing assertions in Cypress to figure out the required format for the dates in your `"have.value"` assertions and fix your tests.

23. Once your test for the default dates passes, commit your changes to your feature branch with a meaningful commit message that describes what you have done and push it to your origin to update your PR.

Testing the Harvest Report Crop List - Spike 3:

In this test you will check that the area dropdown is likely correct. Recall that this dropdown is now populated using data from the database. It contains quite a long list of crops. So rather than checking that every crop is correct, you'll check a few of them and also that there are the right number of crops. This type of sampling rather than exhaustive testing is common when checking long lists of things. It gives high confidence that things are correct, without being overly burdensome to write the tests.

When complete the test you create will get the HTML `<select>` element that is the dropdown. From that element it will get an array containing all of the `<option>` elements in the dropdown, which hold the crop names. It will then check the length of the array of `<option>` elements (i.e. the number of crops) and then also the text (i.e. crop names) in a few of those `<option>` elements. The following steps take you through the creation of this test.

24. Add a `data-cy` attribute with a meaningful value to the `<select>` element for the crop dropdown so that you will be able to easily get it in your test.

25. Add a new `it` to the `describe` in your `e2e.defaults.spec.js` file for this test. Be sure to use a string in the it that is descriptive of what you are testing. Also add a `cy.get` statement that gets the crop dropdown HTML element. Run your test to ensure that it correctly finds the `<select>` element for the crop drop down.

26. Now, to test the contents of the dropdown we need to sample the `<option>` elements that are contained in its `<select>` element. These are the `<option>` elements containing the crop names that are created using a `v-for` in your page. This question takes you through that process.

a. The test you create you will sample the dropdown by checking the names of the first, fifth and last crops in the list, and the length of the list. Use the DevTools to fill in the following table with the values you'll need for your tests.

What to Test	Expected Value
First Crop Name	
Fifth Crop Name	
Last Crop Name	
Number of Crops	

b. Now, in the test you will need to get the specific `<option>` elements that you want to test. Because the `<option>` elements are within the `<select>` element they are called *child elements*. Cypress provides a function named `children` that returns all of the child elements of another element. Cypress then also provides an `eq` function that will return a

single child element by index. For example, if the `data-cy` of your dropdown has the value `crop-dropdown` then the following statement will give the `<option>` element for the first crop:

```
cy.get("[data-cy=crop-dropdown]").children().eq(0)
```

You can then use the Cypress `should` function to make an assertion about the text contained in that element.

Add statements to your test that makes assertions about the text (i.e. crop name) in the first (index 0), fifth and last elements of the crop dropdown.

c. To finish the test, you will need to check the total number of crops. Cypress provides a `"have.length"` assertion that checks the number of elements in a list or collection. Add a statement to your test that makes an assertion which checks that total number of children of the `<select>` element (i.e. the number of crops) is correct.

27. Once your test for the crops dropdown passes, commit your changes to your feature branch with a meaningful commit message that describes what you have done and push it to your origin to update your PR.

28. You've now learned a few new things about Cypress E2E testing. So it might be worth collecting that all together like we did earlier. Fill in the right-hand column with the piece of information described in the left-hand column.

Cypress assertion to check the value of an HTML input element.	
Cypress function to get all of the children of an HTML element.	
Cypress function to get a child element by its index.	
Cypress assertion to check the length of a list or collection.	

Testing the Harvest Report Generation - Spike 4:

While we haven't fully tested all of the defaults for the Harvest Report at this point, we have done enough to give you some practice and to introduce some of the basics of E2E testing with Cypress. There are some optional extras at the end of the assignment that will fill out the rest of these tests if you are interested.

In this section we move on to some tests that simulate user interaction. So we will be adding in steps 3 and 4 of the general testing process defined earlier. These steps were where we will get

HTML elements and perform actions on them (e.g. clicking on a button, choosing an element, or typing text). This will enable us to test that the dynamic features of our pages work correctly.

29. The tests in your `e2e.default.spec.js` file were focused on testing the default values of the page. The tests you will create from here on will focus on checking that the harvest report is generated correctly when the "Generate Report" button is clicked. Because these new tests have a different focus, they should be contained in a separate `spec` file.

Create a new `spec` file in your `e2e` directory for these tests. Be sure to give your file a descriptive name using dots (`.`) as described in question #10.

30. Add a `describe` with a descriptive string, a `beforeEach` and an empty `it` to your new spec file and run it in Cypress to be sure you have the basic structure correct.

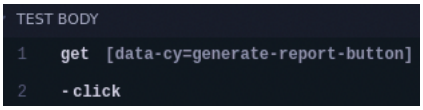
31. To generate the Harvest Report your test will need to get the "Generate Report:" button and then click on it. Add a `data-cy` attribute to your html and a `cy.get` in your `it` to get the "Generate Report" button in your test.

32. Now you need your test to click on the "Generate Report" button. Cypress provides the click function to click on HTML elements. For example, if your button has the `data-cy` value `generate-report-button` then the following statement will click on that button:

```
cy.get("[data-cy=generate-report-button]").click()
```

Add a statement to your test that clicks your "Generate Report" button. Run your test in Cypress and visually confirm that the report is in fact generated.

33. This is a good point to pause and look at a few more of Cypress' features. Look at the TEST BODY in section in the middle panel in Cypress. You should see (among other things) lines for the get and the click operations in your test:



If you point at these lines in Cypress, the browser window on the right will show you what was happening at each point.

a. When you point at the "get" line what happens in the browser window to indicate that your `cy.get` statement has found the button?

b. When you point at the “click” line what happens in the browser to indicate that your test has clicked on the button?

c. If you leave your mouse hovering over the “click” line, the browser will alternate between two views. What are these two views showing you?

34. Now that your test has clicked the “Generate Report” button, it needs to make some assertions that confirm that the Harvest Report is now displayed in the page. We’ll start in this question by just confirming that the report header is created and displayed when the report is generated. Some of the optional extensions suggested at the end would make this test more comprehensive.

a. The report header, the <h1> element that displays the text from the “Title” input field appears only when the report is generated. Before the “Generate Report” button is clicked, this element does not exist in the page. Thus, checking for the absence or visibility of this element is an initial way to check that the “Generate Report” button is generating the report.

Cypress provides the `"not.exist"` assertion to check that an element does not exist.

Add a statement to your test, before the report is generated, that checks that the report header does not exist. Hint: you’ll also want to add a `data-cy` attribute to that element in your html. Run your test in Cypress to check that it passes so far.

b. After the “Generate Report” button is clicked the `<h1>` element for the report header should be visible on the page. The Cypress assertion `"be.visible"` checks that an element is visible on the page.

Add a statement to your test, after the report is generated, that checks that the report header is visible. Run your test in Cypress to check that it passes.

35. Once your test for the “Generate Report” button passes, commit your changes to your feature branch with a meaningful commit message that describes what you have done and push it to your origin to update your PR.

Testing the Harvest Report Details - Spike 5:

When the harvest report is generated, there is additional information about the farm that is also displayed. This includes the name of the farm, the username of the logged in user, and the name of the language being used. In this section you'll create a test that checks that these values are correct.

36. Each of the values to be tested appears within an `<li>` element such as:

```
<li><strong>Farm:</strong> {{ farmName }}</li>
```

Add a `data-cy` attribute to your `html` file and a new `it` to your `spec` file so that you can tests that the name of the farm is correctly set when the report is generated. This test will also need to click the "Generate Report Button".

37. In the prior question when you used a `"have.text"` assertion you needed to check for the full string "Farm: Sample Farm". This is because all of that text is within the `<li>` element. In practice, we might only care about the "Sample Farm" part of this string. Cypress has a `"contain.text"` assertion that checks that the text in the element contains a string. Add a statement to your it from the previous question that uses the `"contain.text"` assertion to check the username that is displayed.

38. The approaches used in the prior two questions worked, but sometimes you will want to be more precise in your testing. In the first approach, we had to test against more of the string than we may have wanted to. In the second approach, we tested only that our target string was sub-string within the element's text. Here' we'll see a third approach that will let us use `"have.text"` while still checking only for the string that we care about.

The HTML `<span>…</span>` tag can be used to markup any content in an HTML file without changing its rendering in the page. It may seem strange to have such a tag, but one of its uses to create elements that are used just for testing purposes. For example, you can surround Vue content with a `<span>`, give it a `data-cy` attribute. For example:

```
<span data-cy="language">{{ language }}</span>
```

Add a `<span>` with a `data-cy` attribute to your html file and a statement to your it that uses `"have.text"` to test the language that is displayed.

Note: In this way a `<span>` element is much like a `<div>` element. The one difference is that, at least by default, a `<span>` does not affect page rendering at all. While content within a `<div>` element will appear with blank space before and after it.

39. Once your test for the report details passes, commit your changes to your feature branch with a meaningful commit message that describes what you have done and push it to your origin to update your PR.

40. You've now learned a few new things about Cypress E2E testing. So, it might be worth collecting this all together like we did twice earlier. Fill in the right-hand column with the piece of information described in the left-hand column.

Cypress assertion to check if an element exists.	
Cypress assertion to check if an element is visible.	
Cypress assertion to check if an element has a string as a sub-string of its text.	
HTML element that can be used to markup content within other tags for more precise testing.	

Other Cypress Assertions and Actions:

At this point you've had some experience writing some E2E tests with Cypress. You have worked with a number of different assertions (e.g. `"have.text"`) and one action (`click`). There are other things to Cypress tests beyond assertions and actions, but as outlined at the start, most tests follow the pattern where you get an element, perform an action on it, and make an assertion about the result. So much of what you will need to know are what actions can you perform and what assertions can you make.

There are some excellent and comprehensive references to assertions and actions that can be used with Cypress:

- Assertions: https://docs.cypress.io/guides/references/assertions#Chai
- Actions: https://docs.cypress.io/guides/core-concepts/interacting-with-elements

41. This question will give you a little practice finding new assertions and actions for things that you might want to do. Use the references linked above to fill in the right-hand column with the assertion or action that would be needed to perform the operation in the left-hand column.

Assert that a value is greater than another value.	
Assert that a value is less than or equal to another value.	
Assert that a value is true.	
Assert that a value is one of a set of possible values.	
Assert that a value is not one of a set of possible values.	
Double click on an element.	
Enter text into a text field or text area.	
Erase the text that is currently in a text field or text area.	
Choose an element in a dropdown menu.	

42. In practice it is also often quite effective to just use your favorite search engine to figure out how to perform a particular test in Cypress. For example, imagine that in a test you want to enter text into a text field and simulate pressing the enter key.

Try the search "Cypress enter text and press enter".

Give a statement that will type your name into a text field followed by the enter key.

Optional Extras: Adding to the Test Suite

All of the following exercises are optional.

In the above exercises you gained a good bit of experience creating E2E tests with cypress. While we tested a number of things, it is probably pretty clear that we did not produce a full test suite for your page. That would take a good bit more work. That said, below are some additional exercises you can try to fill out the test suite and to practice your Cypress E2E testing skills a little more. Some of them you will be able to complete with what you practiced already. Some of them will require you to use things you've found in questions 41 and 42. And some may require you to learn new things about Cypress along the way.

If you attempt any of these features, be sure to make a commit to your feature branch (with a meaningful commit message) for each exercise that you complete and then push your feature branch to GitHub to update your PR.

More Default Tests:

Add `its` for the following tests to your `e2e.default.spec.js` file.

43. Add a test that checks that KALE is the crop that is selected by default in the crop dropdown when the page is loaded.

44. Add a test to check that the area drop down is correctly populated when the page is loaded. This test should use sampling like the one you wrote for the crop dropdown.

More Harvest Report Tests:

Add `its` for the following tests to the `spec.js` file that you created for to test the generation of the harvest report.

45. Extend your test that checks that the report header appears when the “Generate Report” button is clicked to also check that the `<ul>` element containing all of the “Details.” Your test should check that this element does not exist before the “Generate Report” button is clicked and then that it is visible after the “Generate Report” button is clicked.

46. Repeat #45, but perform the checks for the harvest report table.

47. The “Details” of the harvest report display the start and end date and the crop that has been selected. When displayed these values are linked (via Vue data binding) to the input elements at the top of the page. Add a test for each of the following:

a. Test that when the start date input element is changed that the value displayed in the harvest report changes.

b. Test that when the end date input element is changed that the value displayed in the harvest report changes.

c. Test that when the crop selected in the crop drop down is changed that the value displayed in the harvest report changes.

48. The title that appears on the harvest report at the bottom of the page is bound to the “Title” input field at the top of the page. So, when the text in the input field is changed, the title of the harvest report should also change. Add a test that check that the title of the harvest report is bound to the “Title” input field.

49. If the sample database does not contain any harvest logs for the combination of the date range and crop selected then your harvest report should display a message similar to “No Harvest Logs to Display!” Add a test that checks that this message is properly displayed when the date range and crop combination do not contain any harvest logs.

50. Extra Challenging: You’ll notice that we still have not tested anything to do with the contents of the harvest report table. We will do some testing with that in the next assignment. But if you want to try some now here are a few things you could test. Hint: You can use the `children` and `eq` functions to get a row (`<tr>`)from the table and then use the `children` and `eq` functions on the row to get a column (`<td>`).

a. Write a test that checks that there are two harvest logs for KALE in the default date range.

b. Write a test that checks that the dates in the first and last row of the table are between the start and end date for the search. Hint: You can use the dayjs library to compare dates.

c. Write a test that checks every row of the table contains the crop that is selected in the crop drop down. Hint: Use a `for` loop.

Optional: To help us improve and scope these activities for future semesters please consider providing the following feedback.

a. Approximately how much time did you spend on this activity outside of class time?

b. Please comment on any particular challenges you faced in completing this activity.