End-to-End Testing with Cypress

- While unit tests can give us confidence that individual pieces of our code work as intended, and integration tests can give us confidence that small numbers of individual pieces of our code work well together, we currently lack a way to verify that *entire software systems* work as intended.

- This is where ***end-to-end tests*** (also called ***E2E tests***) come into play. E2E tests are a mechanism through which to test a software system or application while providing a high level of ***fidelity*** to the way that system or application is actually used in production.

- E2E tests are specifically tests that run an entire system or application (or a large, connected portion of it) similar to the way it would be run in production and automatically interact with the system or application to validate its behavior.

- For applications with graphical interfaces, E2E testing usually involves having a “robot” click, type, and scroll through the application’s running UI and verifying that each interaction produces the expected results.

- Importantly, because we want to maintain an E2E test environment with high fidelity to the application’s real, production runtime environment, E2E tests typically involve minimal (ideally zero) reliance on test doubles. In other words, during E2E tests, we want to test the *entire* application, so we typically rely on all of the application’s real dependencies, including ones that involve network communication or are otherwise too expensive to include in smaller tests like integration and unit tests.

- Because of this, E2E tests are often far more expensive to run and maintain than smaller tests, and for this reason, E2E tests typically make up the smallest proportion of an application’s entire test suite. In particular, E2E tests are often reserved for testing critical “flows” through an application.

- Here, we will study E2E testing in the context of a complete web application. We will use a tool called Cypress that will run our application in a real web browser (e.g. Firefox or Chrome). Cypress will automatically click and type in our running application in predetermined patterns and use assertions to verify that our application responds correctly to its interactions.

Running an application for testing with Cypress

- Before we start to explore how to use Cypress to perform E2E tests on an application, let's briefly discuss how to run an application so that it can be tested easily with Cypress.

- In particular, because Cypress is a tool designed for running E2E tests, it makes assumptions about the way an application is run for testing. Specifically, because web applications are run through a *server* in production, Cypress assumes the application being tested is running through a server.

- Thus, in what follows, we will assume that we are using Cypress to test a web application running through a server and that the application server can be launched with the following command:

  ```
  npm start
  ```

- For testing an application that is built entirely of HTML, CSS, and client-side JS, the `serve` tool can provide an easy way to run that application through a server. It only assumes that all the application's HTML, CSS, and JS files live somewhere within a common directory.

- To use a web application running through a server, we will need an `http://...` URL to use to communicate with that server from our web browser (or from the browser being used by Cypress). Many servers, including `serve`, will print the application URL when they launch the application. In what follows, we will assume that we know (or can find out) the `http://...` URL for the application we want to test.

Installing and setting up Cypress

- Our first step to be able to use Cypress for E2E testing is to install it using NPM:

  ```
  npm install --save-dev cypress
  ```

- The first time you run this command, it will go through an extra step to download and install the binary for the Cypress application itself.

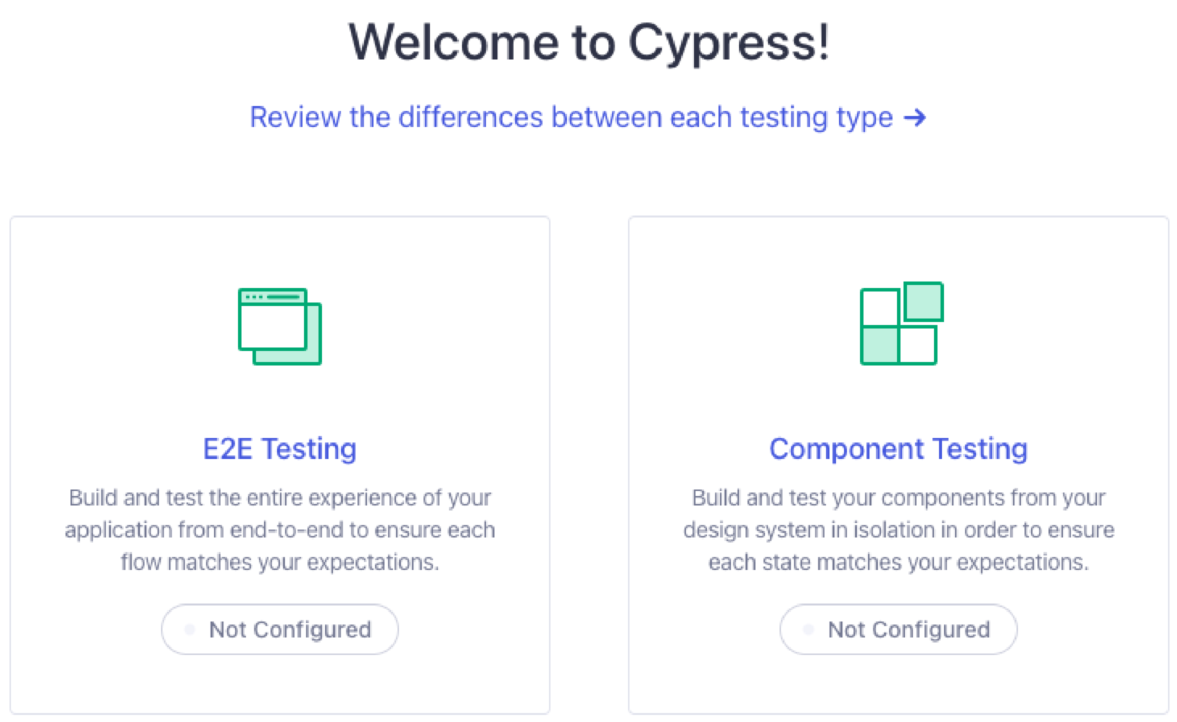
- Note that Cypress has specific system requirements for installation and usage. **Unfortunately, OSU's ENGR servers do not have all the required dependencies (even if you're using NVM to run a newer version of Node.js there).** Thus, you will need to use an alternative runtime environment if you're accustomed to working on the ENGR servers.
  - I'd strongly recommend installing Node.js locally on your laptop or desktop and running Cypress directly on that machine.

- Once you have Cypress installed, you can start the Cypress application. To make this easier to do, we'll add the following entry to the `scripts` field of our project's `package.json` file:

  ```
  "scripts": {
      "cy:open": "cypress open"
  }
  ```

- **Note that the `cypress open` command will launch the GUI-based Cypress application.** If you're working somewhere that's not able to display graphical windows (e.g. in an SSH terminal session or in a GitHub Codespace), you won't be able to run `cypress open` and will have to configure Cypress manually. We'll see what this entails as we go here.

- When Cypress first launches, you'll see a screen that looks like this:


Welcome to Cypress!

Review the differences between each testing type →

E2E Testing

Build and test the entire experience of your application from end-to-end to ensure each flow matches your expectations.

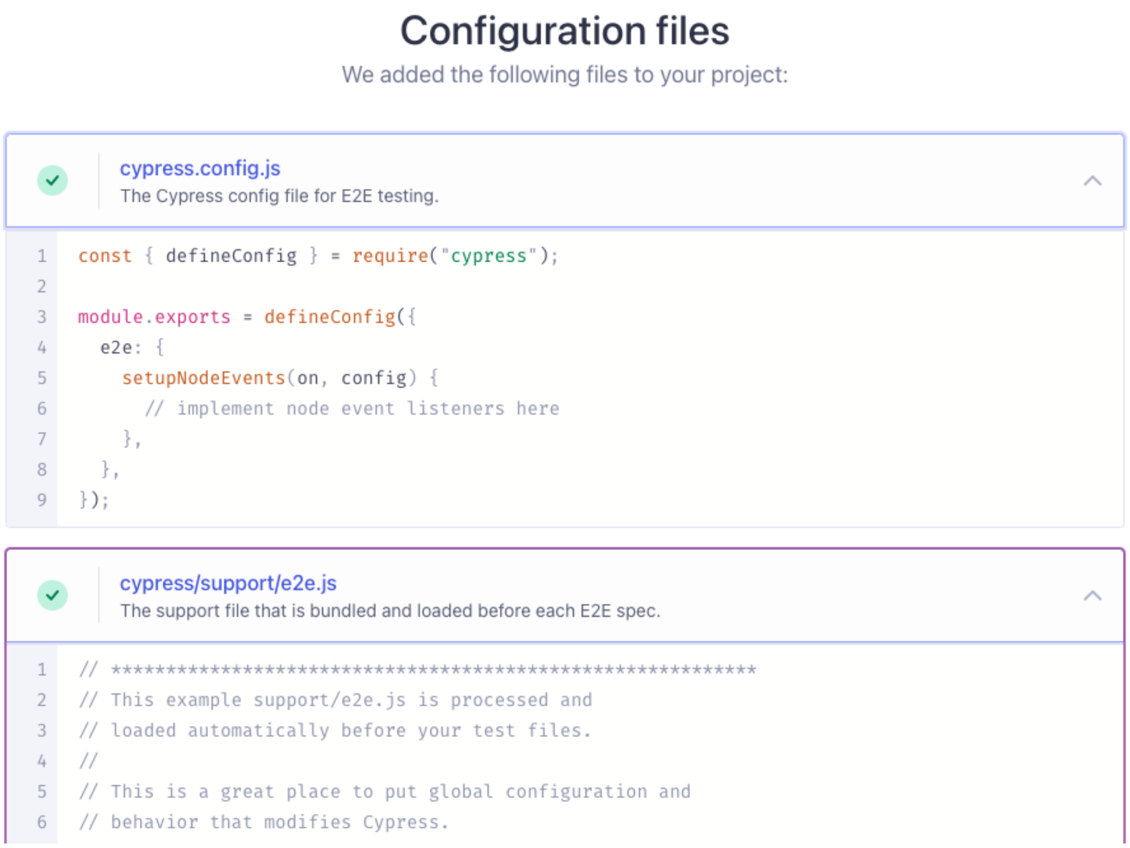
Not Configured

Component Testing

Build and test your components from your design system in isolation in order to ensure each state matches your expectations.

Not Configured

- From this screen, click on “E2E Testing”. This will tell Cypress to configure itself for E2E testing. On the next screen, Cypress will display the contents of the configuration files it will create:



Configuration files

We added the following files to your project:

cypress.config.js
The Cypress config file for E2E testing.

```
const { defineConfig } = require("cypress");

module.exports = defineConfig({
  e2e: {
    setupNodeEvents(on, config) {
      // implement node event listeners here
    },
  },
});
```

cypress/support/e2e.js
The support file that is bundled and loaded before each E2E spec.

```
// ***********************************************************
// This example support/e2e.js is processed and
// loaded automatically before your test files.
//
// This is a great place to put global configuration and
// behavior that modifies Cypress.
```

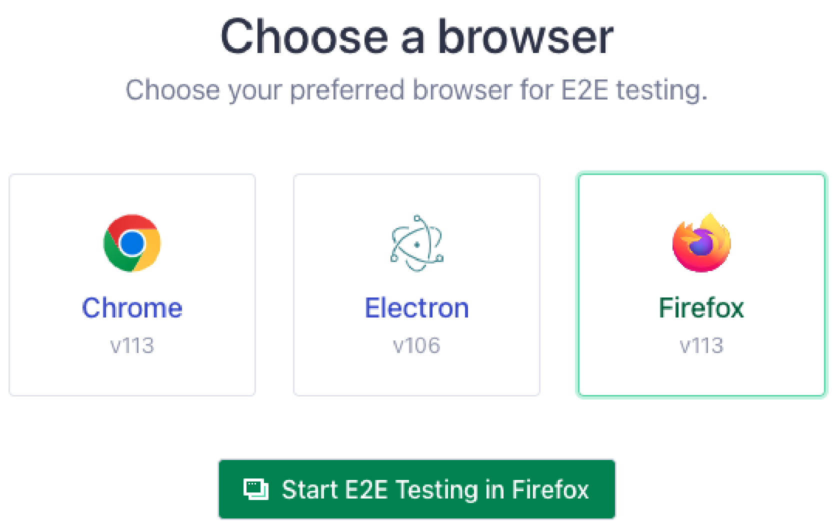
- In case you're not able to run `cypress open` and need to configure Cypress a different way, you can find the default E2E configuration files created by Cypress here.

- **Note that we'll need to make one very small change to the default configuration files for compatibility with our runtime environment.** Specifically, in the file `cypress/support/e2e.js`, we'll uncomment the `require()` line at the bottom of the file and comment out the `import` line just above it:

```
// Import commands.js using ES2015 syntax:
// import './commands'

// Alternatively you can use CommonJS syntax:
require('./commands')
```

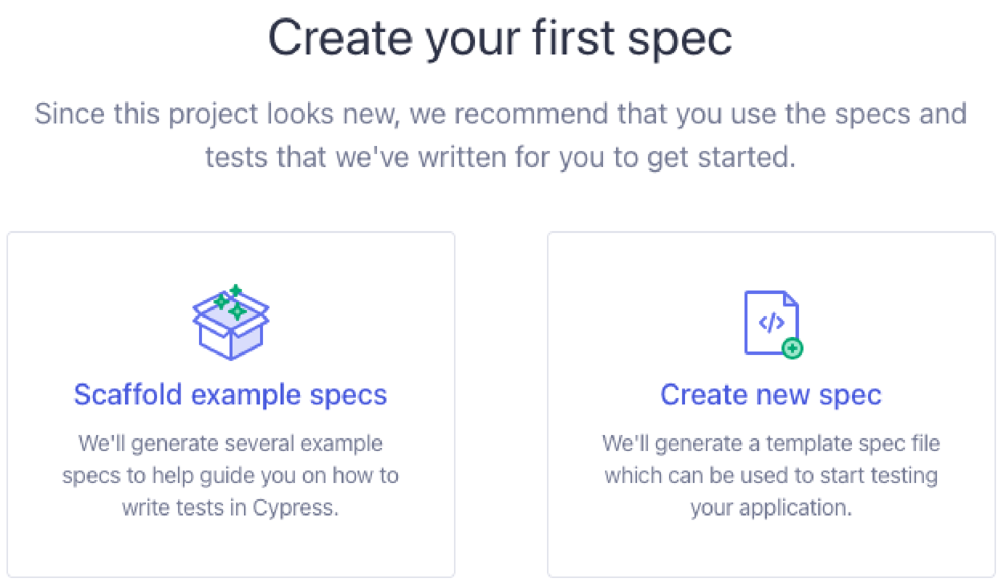
- Finally, Cypress will ask us to choose our preferred browser to use for testing from among the browsers Cypress detects on our system:


Choose a browser
Choose your preferred browser for E2E testing.
Chrome
v113
Electron
v106
Firefox
v113
Start E2E Testing in Firefox

- You can select whichever browser you'd like to use and click the "Start E2E Testing" button. This will launch a window of your selected browser. **This browser window will be controlled by Cypress.** We'll do some work there in the next section.

Writing our first Cypress test spec

- When Cypress first opens our chosen browser in a new project, Cypress will prompt you to set up a ***spec*** (i.e. a Cypress test specification). It will give us two different options:


Create your first spec

Since this project looks new, we recommend that you use the specs and tests that we've written for you to get started.

Scaffold example specs
We'll generate several example specs to help guide you on how to write tests in Cypress.
Create new spec
We'll generate a template spec file which can be used to start testing your application.

- We'll choose the "Create a new spec" option here. For now, we can accept the default filename Cypress uses: `cypress/e2e/spec.cy.js`. This will generate a simple test in the file `cypress/e2e/spec.cy.js`, which we should see listed under the "Specs" tab in the Cypress-controlled browser window.

- The test generated by Cypress simply visits the page https://example.cypress.io and does nothing else. If we click on our spec in the list of specs in the browser, we'll see Cypress go through the process of running the test by opening the specified page in the browser. We should see a test report indicating that the test passed.

- Let's modify the default test spec that was created by Cypress to explore some of Cypress's features. We can do this by opening the test spec file `cypress/e2e/spec.cy.js` in a code editor.

- In this file, we'll see one test defined, which will look something like this:

```
describe('template spec', () => {
    it('passes', () => {
        cy.visit('https://example.cypress.io')
    })
})
```

- Here, we can see a few elements that are somewhat similar to the mechanisms that are provided by Jest. In particular, we see:
    - A `describe()` block, which behaves basically just like a `describe()` block in Jest.
    - An `it()`, which is used to define an individual test, just like `test()` in Jest.

- By default Cypress outputs functions as [JS arrow function expressions](#) (i.e. `() => {}`). These are essentially the same as anonymous functions, with very minor differences. For example, we could rewrite the current test spec as follows to achieve the same thing:

```
describe('template spec', function () {
    it('passes', function () {
        cy.visit('https://example.cypress.io')
    })
})
```

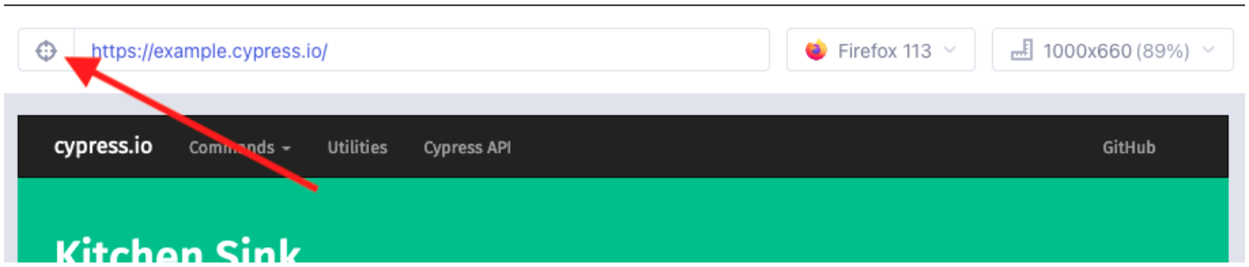
- In addition to the familiar test setup, we see that the test contains a single command, a call to [`cy.visit()`](#). This is a command that's built into Cypress that instructs Cypress to visit the page at a specified URL. In this test, Cypress is specifically visiting [`https://example.cypress.io`](https://example.cypress.io).

- As we saw, the test report indicated that this test passed, even though it doesn't include any assertions. This is because Cypress was able to successfully visit the page at the specified URL. If there had been some kind of error loading the page at the specified URL (e.g. a 404 response from the server or an error in the application's JS code), then the test would have failed.

- Though it's somewhat nonstandard to write tests against an application we don't control, let's continue testing against `https://example.cypress.io` for now, since it's designed as an example testbed, anyway. We'll keep adding on to the first test to see a few more of the kinds of things we can do with Cypress.

- Let's specifically add a call to the test to grab an element on the page. For example, let's try to grab the "dblclick" link, so we can click on it. One way to do this would be to use Cypress's ["Selector Playground" feature](#), which is an interactive tool that can help us determine how to select an element.

- We can launch the Selector Playground by clicking the “crosshairs” icon next to the URL at the top of the Cypress test runner:



- Once the Selector Playground is launched, we can click on a specific element in the page to find a specific selector we can use to select that element. For example, if we click on the “dblclick” link, the Selector Playground will give us this selector, wrapped in a call to `cy.get()`, which can be used to select a specific element using a selector:

```
cy.get(':nth-child(3) > ul > :nth-child(7) > a')
```

- This selector is quite… specific. It is, in fact, a CSS selector that matches the “dblclick” link. This hardly represents how a real user would locate a given element in our page. In general, this is a drawback of the Selector Playground.

- Instead of using a call to `cy.get()` to select an element based on a very complex CSS selector, let’s use a different mechanism that selects an element in a way that more closely resembles the way a user might select an element.

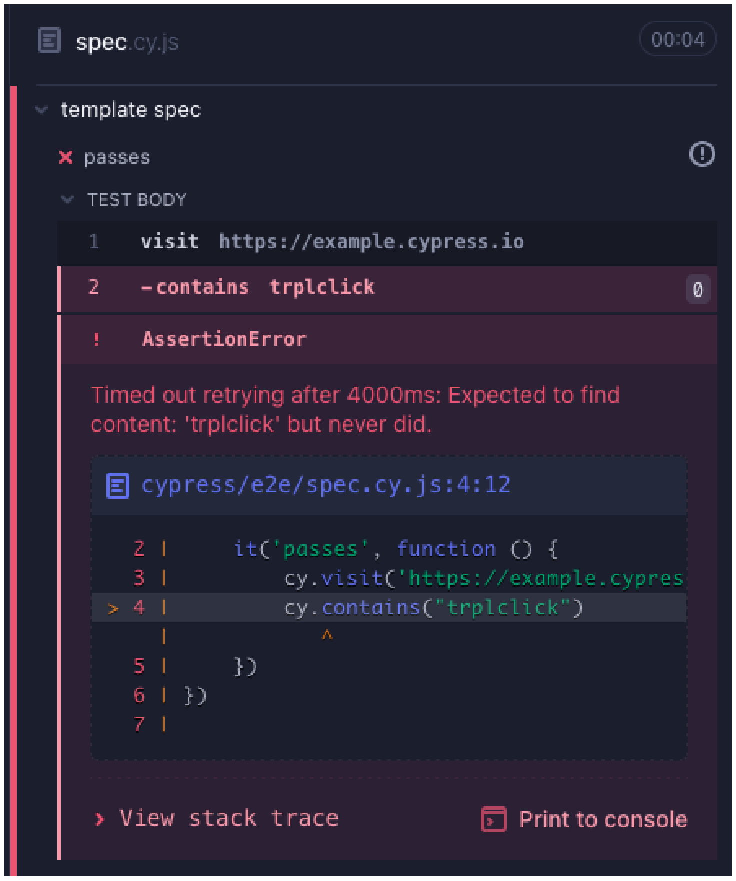
- In this case, we can use `cy.contains()`, which allows us to select an element based on the text it contains, e.g.:

```
cy.contains("dblclick")
```

- If we run the test after adding that command, we should see it pass. As an experiment, let’s temporarily modify the `cy.contains()` command to look for something that’s *not* on the page, e.g.:

```
cy.contains("trplclick")
```

- If we run the test with this modification, we’ll see the test fail:



```
visit  https://example.cypress.io
-contains  trplclick                                    0
!  AssertionError
```

Timed out retrying after 4000ms: Expected to find content: 'trplclick' but never did.

```
cypress/e2e/spec.cy.js:4:12

      it('passes', function () {
          cy.visit('https://example.cypres
>         cy.contains("trplclick")
             ^
      })
  })

```

- Importantly, though, notice that the test doesn't fail until 4 seconds have passed. This is because `cy.contains()` behaves much like the DOM Testing Library's `findBy...` queries, waiting for an element to potentially appear in the page that matches the specified text. In this case, no such element appears after a 4 second timeout, so the test fails.

- If we change the call to `cy.contains()` to find the "dblclick" link again, we can chain a call to Cypress's `.click()` command onto that call:

```
cy.contains("dblclick").click()
```

- This will instruct Cypress to perform a click interaction with the element grabbed by `cy.contains()`. Importantly, the click interaction performed by Cypress will be similar to the interactions we previously initiated with the Testing Library collection's User Event library. In particular, it will perform the interaction in a way that attempts to mimic the way a real user would interact with the application, e.g. scrolling the element into view, determining whether the element is “actionable” (e.g. not hidden or disabled), and then initiating the click itself.

- If we go back to the Cypress test runner and rerun the test, we will see Cypress (very quickly) step through the different commands, loading the page, finding the “dblclick” link, and clicking it, which results in the app navigating to a different page with the URL `https://example.cypress.io/commands/actions`.

- Let’s wrap this test up by adding an assertion to make sure clicking the “dblclick” link correctly navigates to the “Actions” page. Assertions in Cypress use a slightly different syntax from what we are used to from Jest. In particular, in Cypress, assertions are made with [the `.should()` command](), which comes from the popular [Chai assertion library]().

- For example, we could assert that the URL is correct after clicking the “dblclick” link by chaining a call to `.should()` onto a call to [the `cy.url()` command](), e.g.:

```
cy.url().should("include", "/commands/actions")
```

- Now, when we re-run the test, we should see the assertion pass.

Setting Cypress up to test our own application

- Let’s start to write some real tests for our own application using Cypress. The application we’ll test will be a simple to-do tracker. You can find the code for this application linked on our course website. [Here is a video demonstrating]() the essential functionality of this app.

- Just to get an idea about the elements we’ll be working with in our tests, here’s the body of the HTML for the app’s home page, where to-dos are entered:

```
<div id="app">
    <form id="todo-form">
        <input placeholder="Enter a To-Do" id="todo" />
        <button>Add To-Do</button>
        <a href="/archive" class="archive-link">Archive</a>
    </form>
    <div>
```

```
        <div class="todo-list-container">
            <h2>Incomplete</h2>
            <ul id="incomplete-todos"></ul>
        </div>
        <div class="todo-list-container">
            <h2>Completed</h2>
            <ul id="completed-todos"></ul>
        </div>
    </div>
</div>
```

- There's more to the app, including CSS and client-side JS, and it's also set up to be served using `serve`.

- Let's first make sure the to-do application server is running by running the following command in the server's project directory:

  ```
  npm start
  ```

- Then, let's create a new Cypress test spec in the file `cypress/e2e/todos.cy.js`. If we navigate to the "Specs" tab in the Cypress-controlled browser, we should see the `todos.cy.js` spec appear as soon as the file is created.

- Let's start our first test in `todos.cy.js`, in which we'll simply verify that we can successfully create a to-do within the app. The first thing we'll do within this test is visit the app's home page. To do this, we'll pass the application server's URL to `cy.visit()`. In this example, we'll assume the application server's URL is `http://localhost:3000`:

  ```
  it("creates a to-do", function () {
      cy.visit("http://localhost:3000")
  })
  ```

- If we run the new test spec in the Cypress test runner, we'll see Cypress open our to-do application.

- Before we go any further with this test, let's think ahead to imagine some of the tests we might want to write in the future. Whatever these tests might be, all of them will involve visiting some page within our application. Instead of typing the URL `http://localhost:3000` into every test, let's take advantage of a configuration option Cypress provides for specifying a "base URL" for all of our tests.

- Specifically, let's open the Cypress configuration file, `cypress.config.js` and modify the configuration to include the `baseUrl` option, whose value should contain the URL of our application server:

  ```
  const { defineConfig } = require("cypress")

  module.exports = defineConfig({
      e2e: {
          baseUrl: "http://localhost:3000"
      }
  })
  ```

- Note that when you save that modification to the Cypress config file, Cypress will automatically restart itself to incorporate the updated configuration.

- Now, when we call `cy.visit()`, we can specify a URL relative to the value of `baseUrl`. For example, in our first test, we can update the call to `cy.visit()` to look like this:

  ```
  cy.visit("/")
  ```

- This is useful, since it keeps our tests a little more concise and makes it easier to update the tests if the URL for our application server changes.

- Before we proceed any further with our first test, let's take a brief detour to a tool that will make our tests a bit more elegant.

Finding elements using the Cypress Testing Library

- As we've discussed, using `cy.get()` can force us to rely on brittle implementation details (i.e. specific CSS selectors) in our tests. Above, we used `cy.contains()` to get around this issue, but `cy.contains()` is somewhat limiting, since it only allows us to find elements based on their text content.

- When we previously implemented integration tests for DOM based applications, we employed the DOM Testing Library to provide us with a flexible set of queries that allowed us to grab elements in the application based on visual characteristics similar to the ones real users would use to find application elements.

- Luckily, the Testing Library collection also contains a library called the [Cypress Testing Library]() that allows us to use the same queries we used previously (e.g. `findByLabelText()`) to find elements in the application. Let's use this library here to give ourselves a little more flexibility in how we find application elements in our Cypress tests.

- We'll begin by installing the Cypress Testing Library as a development dependency for our project:

```
npm install --save-dev @testing-library/cypress
```

- The Cypress Testing Library extends the commands available on Cypress's global `cy` object, allowing us to use the same `findBy...` and `findAllBy...` queries we previously used from the DOM Testing Library. To add these extended commands, we must add the following line to the file `cypress/support/commands.js`:

```
require("@testing-library/cypress/add-commands")
```

- Then, within our Cypress tests, we'll be able to use commands like `cy.findByRole()`, etc. Let's go back and finish our first test with the help of these commands.

- Within the first test we started above, we want Cypress to simulate two interactions with the app: typing a to-do into the text input field and then clicking the "Add To-Do" button to create the to-do.

- Let's start by finding the text input field. With the Cypress Testing Library commands available to us, we can do this in a familiar way, based on its placeholder text:

```
cy.findByPlaceholderText("Enter a To-Do")
```

- To have Cypress perform a typing interaction on the input field, we can chain a call to the `.type()` action onto the command above:

```
cy.findByPlaceholderText("Enter a To-Do")
    .type("Add our first to-do")
```

- Now, if we go back to the Cypress test runner and re-run our test, we'll see Cypress actually "type" the specified text, one character at a time, into the input field.

- We can use a similarly structured command/action chain to find the button based on its role and text and then click on it:

```
cy.findByRole("button", { name: "Add To-Do" })
    .click()
```

- Again, if we watch Cypress re-execute this test in the test runner, we should see it type the to-do text and click the button, and a new to-do element will appear as a result.

- Finally, let's add a basic assertion to make sure that the to-do was correctly added:

```
cy.findByText("Add our first to-do").should("exist")
```

- Now, we should see the assertion executed in the Cypress test runner, and the test should pass.

Test isolation in Cypress

- Let's start working on a new test to make sure a to-do can be correctly marked as "completed" by checking its checkbox. Let's start with just an empty test:

```
it("completes a todo", function () {})
```

- Before adding any code to this test, let's go back to the Cypress test runner and re-run the entire test suite. If we do, we'll see (if we have good, quick eyes) the first test run, with Cypress typing a to-do and clicking the submit button. Then, when Cypress gets to our new second test, we'll see nothing at all happen. In fact, Cypress won't even be displaying our to-do app anymore, just what it calls the "default blank page".

- The important thing to understand here is that our second test didn't pick up right where our first test left off. Instead Cypress isolates each test from the results of other tests.

- In particular, each test starts in a fresh browser context and any data stored in cookies, local storage, or session storage is cleared. In other words, each test must re-visit the application and take whatever actions are necessary to set up the DOM and browser state for each test.

- We can see the effects of test isolation if we add a call to `cy.visit()` to our new test to re-visit the home page of the app:

```
it("completes a todo", function () {
    cy.visit("/")
})
```

- Now, if we re-run the tests, we'll see that the second test starts without any to-dos, even though we had created one in the first test. This is because the to-dos the app creates are stored in local storage, which is cleared before each test. Thus, the to-dos we create on one test will not carry over to other tests, and we'll have to freshly create the to-dos we need for each test (in the next section, we'll see a mechanism for accomplishing this without writing too much code).
    - If the to-do app was implemented using a different architecture, with to-dos stored in a remote database instead of in the browser's local storage, it would be our responsibility to reset and seed the database with data before each test.

- Isolating tests is a purposeful design decision in Cypress. The goal behind this design decision is to make tests more reliable. Specifically, when each test is isolated from other tests, the likelihood of nondeterministic test failures resulting from application state from an earlier test is greatly reduced. In other words, isolated tests are less likely to be flaky.

Using custom Cypress commands for complex, repeated actions

- Because tests are isolated from each other, we will sometimes find ourselves needing to repeat complex actions in multiple tests to set up the application state for a test. In order to help reduce the amount of complex, repetitive code in our tests, Cypress allows us to define custom commands, which are essentially just functions that combine multiple individual commands into a single command.

- When writing tests for our to-do application, for example, we will frequently find ourselves needing to insert a new to-do into the app by typing out a to-do in the text entry field and clicking the button to submit the to-do, like we did in our first test. Instead of repeating the individual commands to do this every time we need to create a to-do, let's create a custom command to do this.

- Custom commands are defined in the file `cypress/support/commands.js`. At the bottom of this file, we'll add a new command by calling `Cypress.Commands.add()`. Each command consists of a name and a function, the latter of which we should set up to accept whatever arguments we want to pass when we call the command.

- In this case, we'll create a new command named `addTodo` that accepts the text of the new to-do as an argument. The command will simply perform the same actions as we performed in our first test to insert a new to-do:

```
Cypress.Commands.add("addTodo", function (text) {
    cy.findByPlaceholderText("Enter a To-Do").type(text)
    cy.findByRole("button", { name: "Add To-Do" }).click()
})
```

- Now, in our tests, whenever we want to insert a new to-do, we can use this command. For example, in our first test, we can replace the original actions with a call to `cy.addTodo()`, which is our new command:

```
it("creates a to-do", function () {
    cy.visit("/")
    cy.addTodo("Add our first to-do")
    cy.findByText("Add our first to-do").should("exist")
})
```

- We could use the command to add a new to-do in the second test, too:

  ```
  it("completes a todo", function () {
      cy.visit("/")
      cy.addTodo("Complete this to-do")
  })
  ```

- Now we need to work on marking the to-do as complete by clicking its checkbox. To be able to do this, we'll need to be able to *find* the checkbox for the to-do. This will take a bit of work.

Using test IDs to help find specific elements

- We want to be able to easily locate the checkbox for a specific to-do so we can click it to toggle the to-do's completed state. Looking ahead, we may frequently need to toggle a to-do's completed state, so we may want to create a custom command to do this for us. This will mean we'll want to be able to locate a to-do's checkbox in a general way, not assuming, for example, that there is only one to-do in the application.

- We'll create a new custom command to work in here. We'll set the command up so that it toggles a to-do's completed state based on the text of the to-do (we're safe assuming each to-do we create in our tests will have unique text):

  ```
  Cypress.Commands.add("toggleTodoCompleted", function (text)
  {
     ...
  })
  ```

- The question remains: how do we locate the checkbox for a specific to-do, given that to-do's text? We can start out by locating the element containing the specified text, which we know how to do:

  ```
  cy.findByText(text)
  ```

- What next? An individual to-do is a somewhat complex element. Each to-do looks something like this (this is a simplified version of an actual to-do element):

```
<li class="todo">
   <input type="checkbox" />
   <p class="todo-text">This is the text of the to-do</p>
</li>
```

- When we call `cy.findByText()` with the text of the to-do we want to grab, that command will actually provide a reference to the `<p>` element where the text lives.

- To be able to access the checkbox from the reference to the `<p>` element we grab with `cy.findByText()`, we could take advantage of the fact that these two elements are siblings in the DOM tree (Cypress provides a .siblings() query that we can use to find an element's siblings). However, this would be a brittle way to implement a test, since we might some day want to change the structure of a to-do element, e.g. nesting the `<p>` element within another element. This could break the test.

- We can be more confident that the element containing the to-do text will always be a *descendent* of the top-level to-do element itself. This opens the possibility of using Cypress's `.parents()` query, which can be used to find an *ancestor* of a given element.

- However, to be able to use the `.parents()` query effectively, we need to have a way to identify the top-level to-do element in a way that's resilient to changes to the code. For example, we don't want to rely on the top-level to-do element's tag type (`<li>`) or CSS class (`todo`), since these are parts of the code we might want to change down the road, e.g. if we want to update the structure or styling of the app.

- This is where test IDs can be useful. A test ID is a special identifier we attach to an element to make it easy to find during testing. A test ID is typically added to an element as an attribute in the HTML that begins with the prefix "`data-`", e.g. `data-testid`.

- For example we could add a test ID to each to-do element when it's created. This might look something like this (the suffix "`-1`" here is a unique identifier tied to this specific to-do):

```
<li class="todo" data-testid="todo-1">...</li>
```

- Note that test IDs like this, specifically using the attribute name `data-testid`, are directly supported by the libraries in the Testing Library collection.

- With a test ID like that assigned to the top-level to-do element, we can easily find the to-do element itself after locating its text using Cypress's `.parents()` query:

```
cy.findByText(text)
    .parents("[data-testid*='todo']")
```

- This query specifically finds the ancestor of the to-do text element that has a `data-testid` attribute whose value contains "`todo`", i.e. the top-level to-do element itself.

- Once we identify the top-level to-do element corresponding to the specified text, we can easily locate the checkbox inside it and click that checkbox to toggle the to-do's completed state:

```
cy.findByText(text)
    .parents("[data-testid*='todo']")
    .findByRole("checkbox")
    .click()
```

- That completes our `toggleTodoCompleted` action. We can now use it in our test to mark the to-do we created as completed:

```
it("completes a todo", function () {
    cy.visit("/")
    cy.addTodo("Complete this to-do")
    cy.toggleTodoCompleted("Complete this to-do")
})
```

- If we run the test now, we'll see Cypress click the to-do's checkbox to mark it as completed, and it'll move over to the "completed" column. We'll want to finish the test with an assertion that somehow confirms that the to-do is marked as completed.

- The most straightforward way to accomplish this would be to assert that the to-do lives within the list of completed to-dos (and perhaps not in the list of incomplete to-dos). However, these lists are another example of the kind of element that is hard to grab onto. They each are another good candidate for a test ID:

  ```
  <ul id="incomplete-todos"
  data-testid="incomplete-todos"></ul>
  ...
  <ul id="completed-todos"
  data-testid="completed-todos"></ul>
  ```

- With test IDs assigned to those two lists, we can finish our test with these two assertions:

  ```
  it("completes a todo", function () {
      cy.visit("/")
      cy.addTodo("Complete this to-do")
      cy.toggleTodoCompleted("Complete this to-do")
      cy.findByTestId("completed-todos")
          .should("contain", "Complete this to-do")
      cy.findByTestId("incomplete-todos")
          .should("not.contain", "Complete this to-do")
  })
  ```

- When our test runs, these two assertions should both pass.

- Before we move on, there are a couple important things to say about test IDs. In particular, finding elements using test IDs may seem like incorporating implementation details into the test. In some ways it is. However, it is important to note the following things:
  - The rationale for avoiding implementation details within tests is because we want our tests to be resilient to implementation changes. Test IDs themselves are more resilient to implementation changes than other implementation-based mechanisms for finding elements since, unlike `class`, `id`, or other similar attribute values, there is no implementation-driven reason to *change* a test ID once it's established.

    - Just based on its presence, a test ID can be a signal to developers that an element is used in testing.
    - Test IDs should be used as a last resort, an “escape hatch” of sorts, for when there’s no other easy way to find an element.
    - For example, in the cases where we applied test IDs above, we were using them to identify elements that were easily distinguishable in the application based on their visual characteristics (i.e. they would be recognizable visual landmarks to a user) but that we simply lacked an easy way to identify programmatically. These are the ideal situations to use test IDs.

Testing a more complex, multi-page flow

- As a final test, let’s validate a more complex, multi-page flow within the to-dos app. Specifically, let’s validate that a to-do can be successfully deleted. Deleting a to-do involves the following steps:
    - Creating the to-do on the app’s home page.
    - Clicking the to-do’s checkbox to mark it as completed.
    - Clicking the to-do’s “archive” button to send it to the archive.
    - Navigating to the archive page (e.g. by clicking the “Archive” link).
    - Clicking the to-do’s “delete” button.

- This test will give a good example of what a real, full E2E test might look like.

- To make the test itself more concise, let’s implement two new custom Cypress commands to archive a to-do that’s already been marked as completed and to delete a to-do that’s already been archived. These commands will be similar in structure to the `toggleTodoCompleted` command we implemented above:

```
Cypress.Commands.add("archiveCompletedTodo", function
(text) {
    cy.findByText(text)
        .parents("[data-testid*='todo']")
        .findByRole("button", { name: "Archive to-do" })
        .click()
})

Cypress.Commands.add("deleteArchivedTodo", function (text)
{
    cy.findByText(text)
        .parents("[data-testid*='todo']")
        .findByRole("button", { name: "Delete to-do" })
        .click()
})
```

- The only difference between these commands and the `toggleTodoCompleted` command is that we are finding the appropriate buttons here instead of the checkbox.

- Our test will start out similar to the way our previous test started, adding the additional action of archiving the to-do:

```
it("deletes a todo", function () {
    cy.visit("/")
    cy.addTodo("Delete this to-do")
    cy.toggleTodoCompleted("Delete this to-do")
    cy.archiveCompletedTodo("Delete this to-do")
})
```

- Here, though, we'll want to navigate to a different page in the app, the archive page. We could do this by calling `cy.visit("/archive")`, but a user would be more likely to navigate to the archive page by clicking the "Archive" link on the app's home page, so let's do that:

```
cy.findByRole("link", { name: "Archive" }).click()
```

- If we run the test now, we should see that it ends up on the archive page, where our to-do now lives. Let's delete the to-do using our `deleteArchivedTodo` action:

```
cy.deleteArchivedTodo("Delete this to-do")
```

- Now we want to make some assertions to verify that the to-do was deleted. There are different ways we could accomplish this, but we'll do it by verifying that the to-do's text doesn't appear anywhere in the app.

- We can start on the archive page, since we're already there. We'll verify that the entire body of the document does not contain the to-do's text:

  ```
  cy.document()
      .its("body")
      .should("not.contain", "Delete this to-do")
  ```

- We can then navigate back to the app's home page (the only other page in the app) and perform the same assertion there:

  ```
  cy.findByRole("link", { name: "Home" }).click()
  cy.document()
      .its("body")
      .should("not.contain", "Delete this to-do")
  ```

- That completes our test. If we run it in the Cypress test runner, we should see the assertions pass.

Running a Cypress test spec headlessly

- In some situations, we'll want to run a Cypress test spec "***headlessly***", that is, without showing the browser window. This is useful for situations where a graphical display is not available, like when our tests are running as part of a continuous integration pipeline.

- For such situations, Cypress includes an alternative command line command, `cypress run`. Let's add a new entry to the `scripts` field in our project's `package.json` file to run the `cypress run` command:

  ```
  "scripts": {
    "cy:run": "cypress run"
  }
  ```

- For now, let's ensure that our application server is still running and then run:

  ```
  npm run cy:run
  ```

- We should see Cypress print out messaging guiding us through its progress running *both* test specs (our original spec, `spec.cy.js`, and the spec for the to-dos app, `todos.cy.js`). At the end, Cypress will report the results of all the tests.

- Cypress will also even create videos of the tests being executed during a headless run, which it will store in the directory `cypress/videos/`. These videos can be useful for helping to visualize the test results and/or debug failing tests.
    - Test videos *should not* be included in version control. For example, we'd want to add a line to our `.gitignore` file to ignore the directory `cypress/videos/`.

- Here, we were able to manually ensure that our application server was running before the tests ran against it. In some situations (like when our tests run in a continuous integration pipeline), we won't be able to manually start the server. To make it easy to make sure our server is running before our tests run, we can use the `start-server-and-test` package from NPM.

- This package does just what its name implies: it starts our application server, ensures that the server is running (by polling the server URL), and once the server is running, runs our tests against it.

- We can install this package using NPM:

  ```
  npm install --save-dev start-server-and-test
  ```

- Then, we can add an entry to the `scripts` field of our project's `package.json` to run the tests using `start-server-and-test`. The `start-server-and-test` command takes three arguments:
    - The name of the command to use to start our application server. In our case this is `npm start` (though we just need to specify `start`).
    - Our application server's URL (used to poll the server to determine when it's ready). In our case, this is `http://localhost:3001`.
    - The name of the command to use to run our tests. In our case this is `npm cy:run` (though, again, we just need to specify `cy:run`).

- The entire `scripts` entry will look like this:

  ```
  "scripts": {
      "test":
          "start-server-and-test start http://localhost:3000
  cy:run"
  }
  ```

- Now, we can run `npm test`, and we'll see our server launch and the tests run against it.

Closing thoughts

- End-to-end testing is a complex process, and Cypress is a complex tool. We're only really scratching the surface of both of them here.

- For more detailed coverage of end-to-end testing (other kinds of larger tests), I'd strongly recommend reading Chapter 14 ("Larger Testing") of Software Engineering at Google.

- For more detailed coverage of Cypress in particular, the Cypress documentation is excellent and extensive. I'd specifically recommend their guide on best practices.