

SENG 365 Week 12

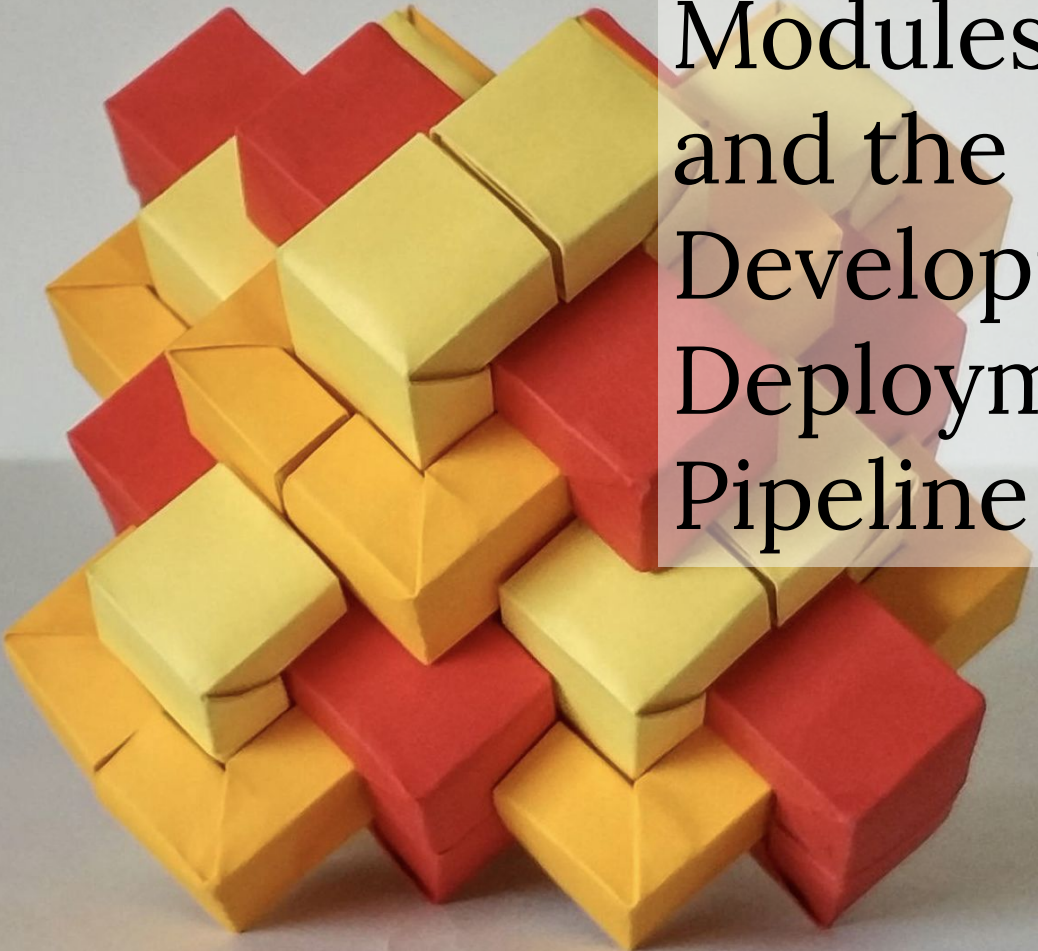
Front-end testing





Topics this week

- Modules and dev/deployment pipeline
- Debugging React
- Automated testing
- Webdriver
- Client-side JavaScript testing
- Final Exam

A cluster of red and yellow paper cubes, some of which are folded into a specific geometric shape, arranged in a 3D pattern. The cubes are made of paper and have sharp edges. The red cubes are interspersed among the yellow ones, creating a complex, multi-colored structure. The cubes are arranged in a way that they appear to be interconnected, with some cubes sitting on top of others, creating a sense of depth and volume. The background is a plain, light gray surface.

Modules and the Development / Deployment Pipeline

Downloading Single Page Apps (SPA)

- A SPA usually consists mostly of JS and HTML templates. The CSS load is typically higher due to custom styling requirements
- Critically, the 'preferred form' of working with SPAs is not a good format for downloading:
 - Many small files,
 - Data that is not needed at runtime (ie, good function names),
 - Often the preferred coding language is not JavaScript (e.g. TypeScript)

Bundling

- Bundling is the process of taking the preferred form of Web code/content and putting it into the preferred form for a browser to consume
 - Analogous to compiling a C or Java program
- What does a browser/user prefer?
 - Fewer network requests (fewer files)
 - Smaller network requests
 - Parallel network requests
 - No network requests at all (caching)

Simple Bundling (fewer files)

- Concatenate all required *.js files to one big file
- Do the same for *.css files
- Tools like Gulp and Grunt follow this basic pattern:
 - Sweep up a bunch of files without knowing about file content
 - Perform some transform on them to generate (fewer) output files

Simple Bundling (smaller files)

- ⦿ Browsers don't care about JS function names (much)
- ⦿ We do care about data sent over the network though
- ⦿ UglifyJS shortens var and function names

```
1 function globalFunctionName(input) {  
2   function nestedFunction() {  
3     input = input + 1;  
4     console.log(input);  
5   }  
6   nestedFunction();  
7 }  
8 -----  
9 $ uglifyjs foo.js --mangle --compress --source-map=map.out  
10 function globalFunctionName(n){!function(){n+=1,console.log(n)}()}  
11
```

UglifyJS (terser, etc)

- Code is name mangled, some optimizations are applied
 - Mangling can only be applied when we are **sure** that all instances of the name are renamed
- Result is difficult to read, but very easy to reverse engineer – so this offers no IP protection
 - There is no real way to protect IP that runs in a browser
 - Anything you send is public knowledge
- To aid debugging, **sourcemaps** can be created:
 - Browser loads mangled code + sourcemaps and the dev tools presents the code as it originally was
 - Learn more:
<https://github.com/ryanseddon/source-map/wiki/Source-maps%3A-languages,-tools-and-other-info>

Webpack (rollup/parcel)

- Webpack and other bundlers are the current best practice
- Webpack understands the input language (JavaScript/TypeScript)
- Configure Webpack with
 - Where to find input files
 - Where to start from (app entry point)
- Webpack will recursively follow imports
 - `Import -> import foo from "module_name"`
 - `Export -> export foo = 42;`
 - Only the files that are used get pulled into the final output
 - Unused (dead) code can be removed
- Final output will be something like `main.9789bf1740765e50e459.js`
 - E.g. ~10Mb of JS

Lazy Loading

- Break your SPA into logical chunks, eg
 - Login (likely the main chunk)
 - Settings
- Only download the chunks you need
- Typically loading is triggered on router changes
 - /login
 - /settings
- The framework and the bundler need to agree on how to split the import graph, usually done with bundler plugins.

Lazy Loading

- Webpack transforms first line at compile time

```
// Instead of a usual import  
import MyComponent from "~/components/MyComponent.js";  
  
// do this  
const MyComponent = () => import("~/components/MyComponent.js");
```

See <https://webpack.js.org/guides/lazy-loading/>



Debugging React Apps



Tips on debugging

- When you run in dev, hot reloading is enabled
- Keep browser open with web page (e.g. `localhost:3000`) and console visible while developing
- Sometimes when it crashes (shows error message in browser) you need to reload the page



console.log

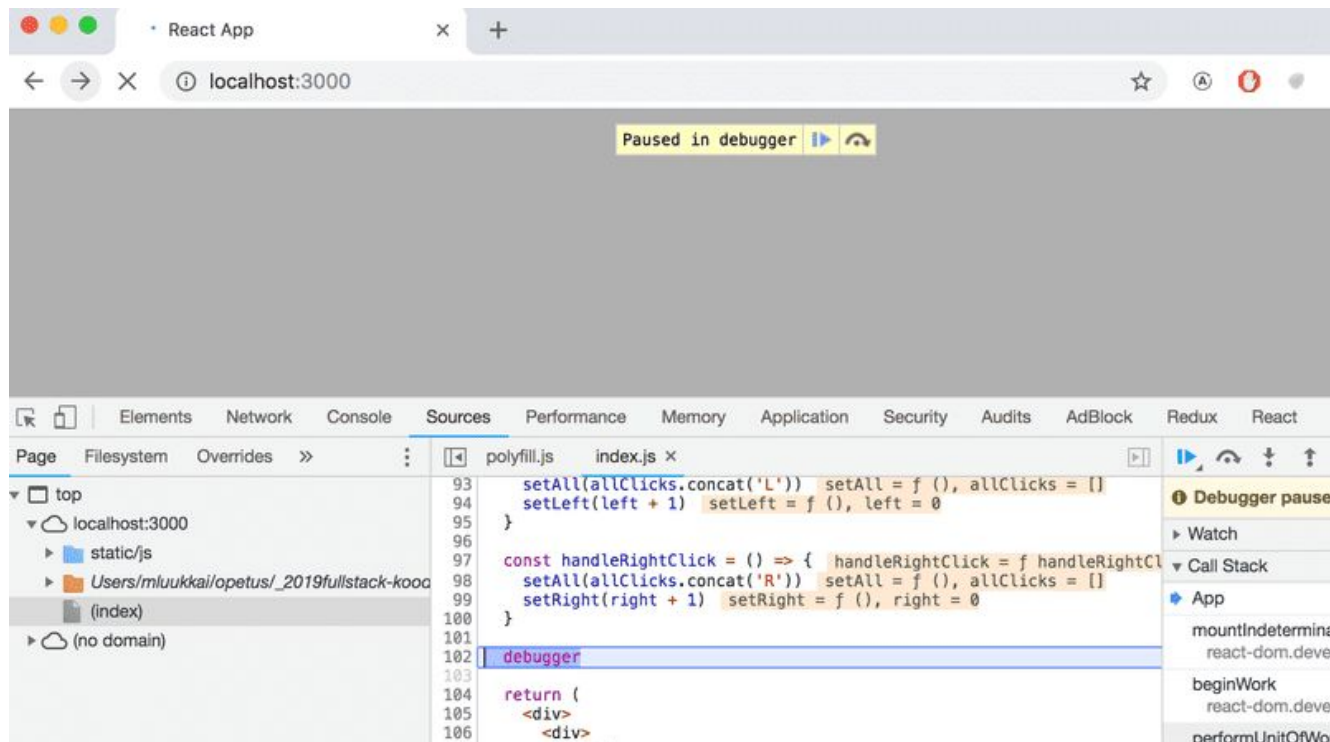
- `console.log` is still a great way to view the state of variables when components load
- Remember to remove these before production, though!

```
console.log('props value is', props)
```

Use comma, not + when concatenating complex objects to strings



debugger in Chrome developer



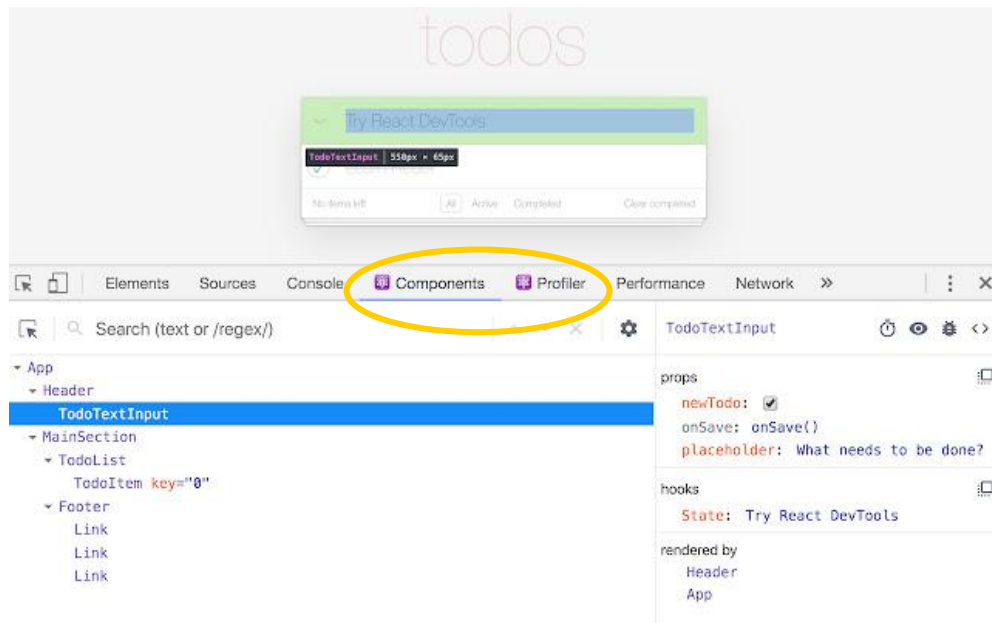
Can also set **breakpoints** manually.

Once it has stopped you can use the console to view the value of various variables.



React developer tools

<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi>



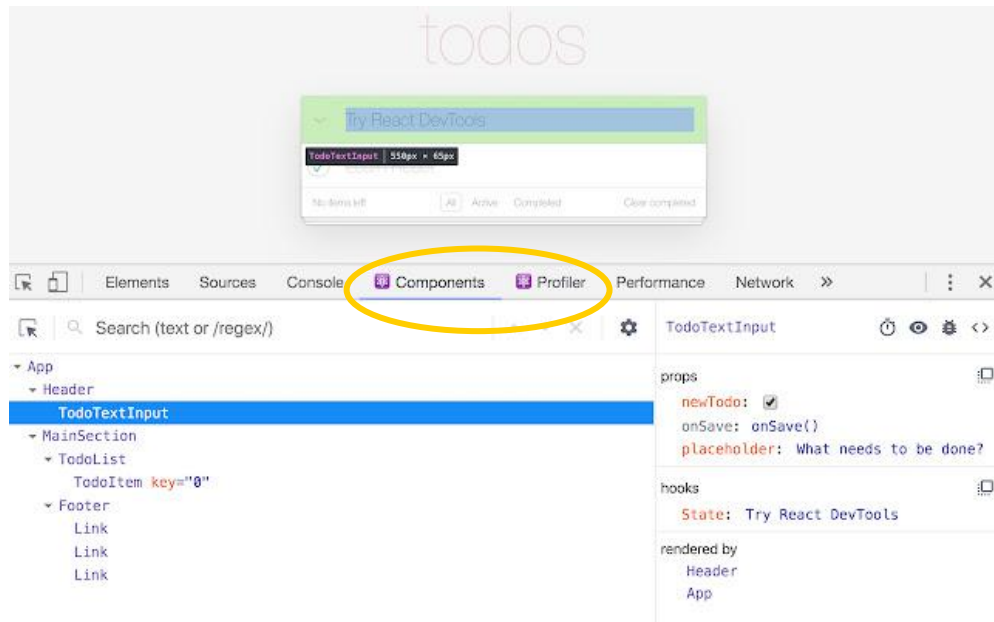
Adds a **Components tab** to the developer console

Can view component **state** and **props**, and **hooks** in order of definition



React developer tools

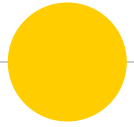
<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi>



Adds a **Components** and **Profiler tabs** to the developer console

Can view component **state** and **props**, and **hooks** in order of definition

Profiler allows you to test performance of components



Automated testing

Web Applications

Brief overview to testing

- Many different types of tests
 - Unit tests, behavioural tests, acceptance tests, regression tests
- Many different kinds of system to test
 - Safety-critical, embedded systems, real-time systems etc
- Particular challenges for testing web applications:
 - Different browsers (FireFox, Chrome)
 - Different versions of browser (IE6 vs IE8)
 - Differences in versions of HTML, ECMAScript (JavaScript)
 - Differences in APIs e.g. how XHRs are handled by different browsers
 - Differences in the handling of the DOM, JavaScript etc.
 - Differences in libraries and versions

Advantages of automating testing

- Automation means you can offload testing to machines (rather than rely on humans)
- (More) frequent testing e.g. regression testing
- Quick and regular feedback to developers
- Virtually unlimited iterations of test case execution
- Support for Agile and extreme development methodologies
- Disciplined documentation of test cases
- Customized defect reporting
- Finding defects missed by manual testing
- Supports continuous integration and continuous deployment

When not to automate (and manually test instead)

- On the client side:
 - If the user interface is rapidly changing
 - e.g. HTML elements are changing
 - Will need to keep changing the tests to match the interface
- On the server side
 - If the API is rapidly changing...
 - Tight timescales
 - Don't have time to develop the tests



Automated testing of clients: WebDriver API

<https://www.w3.org/TR/webdriver/>



WebDriver API

- Intended to enable web authors to write tests that automate a user agent using a separate controlling process
 - May also be used to allow in-browser scripts to control a (possibly separate) browser
- Provides a set of interfaces to:
 - discover and manipulate DOM elements in web documents; and
 - to control the behavior of a user agent (a browser).
- Provides a platform-neutral and language-neutral wire protocol as a way for out-of-process programs to remotely instruct the behavior of web browsers.
- Forms part of the W3C Web Testing Activity

WebDriver API



- Intended to enable web authors to write tests that automate a user agent using a separate controlling process...
- Provides a set of interfaces to discover and manipulate DOM elements in web documents; and
- Provides a platform-neutral and language-neutral wire protocol as a way for out-of-process programs to remotely instruct the behavior of web browsers.

Questions

- Any implications relating to the DOM here?
- Any concerns relating to out-of-process programme controlling web browsers?



Selenium

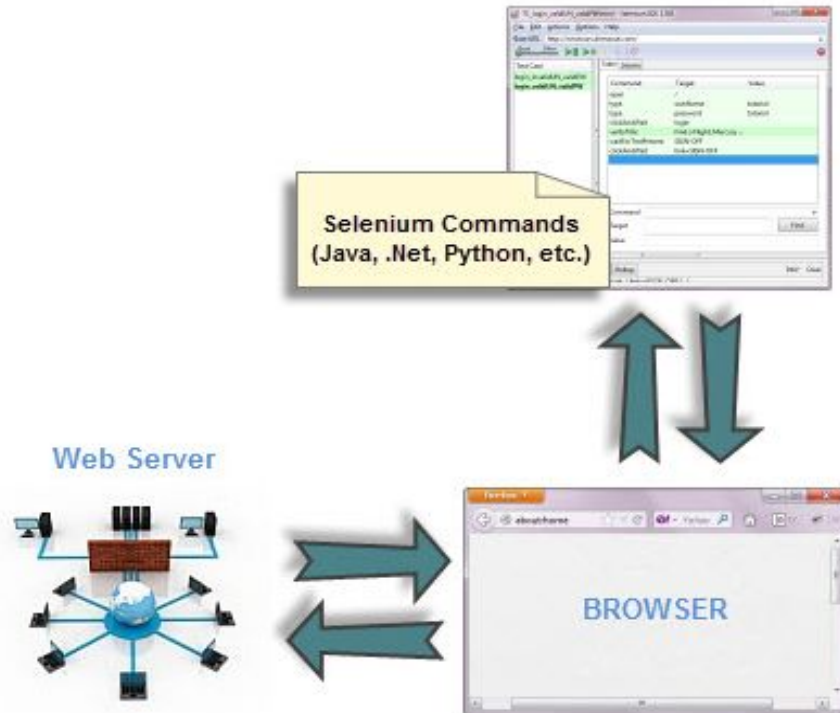
Selenium-WebDriver | Selenium 2.0

- Automates a browser
 - Use to automate testing
 - Use to automate routine web tasks e.g. some admin task
- Selenium WebDriver: drive the browser (automatically) the way a user would
 - Automate what you want the user to do
 - Automate what you want the user not to do
 - Automate unintentional behaviour, accidental behaviour, stupid behaviour, risky behaviour (e.g. security attacks?)

Selenium-WebDriver

- Developed to better support automation of **dynamic web pages**,
 - Dynamic pages: elements of a page may change without the page itself being reloaded. (What is a webpage?)
 - Single Page Applications generate dynamic web pages
- WebDriver relies on the browser's built-in (native) support for automation
 - You'll need an update-to-date browser
 - Harder to automatically test older browsers

Selenium Webdriver



Cross-browser testing

“When we say “JavaScript” we actually mean “JavaScript and the DOM”. Although the DOM is defined by the W3C each browser has its own quirks and differences in their implementation of the DOM and in how JavaScript interacts with it. HtmlUnit has an impressively complete implementation of the DOM and has good support for using JavaScript, but it is no different from any other browser: it has its own quirks and differences from both the W3C standard and the DOM implementations of the major browsers, despite its ability to mimic other browsers.”

http://www.seleniumhq.org/docs/03_webdriver.jsp#htmlunit-driver

Implication: automate your testing on **different** browsers



Example automated testing

```
const webdriver = require('selenium-webdriver'),
    By = webdriver.By,
    Key = webdriver.Key,
    until = webdriver.until;

const driver = new webdriver.Builder().forBrowser('firefox').build();

driver.get('http://google.co.nz')
    .then(_ =>
        driver.findElement(By.name('q')).sendKeys('university of canterbury',
            Key.RETURN))
    .then(_ => driver.wait(until.titleIs('university of canterbury - Google Search'),
        1000))
    .then(_ => driver.quit());
```



Example automated testing

```
const webdriver = require('selenium-webdriver'),
```

```
By = webdriver.By,  
Key = webdriver.Key,  
until = webdriver.until;
```

```
const driver = new webdriver.Builder().forBrowser('firefox').build();
```

```
driver.get('http://google.co.nz')
```

```
.then(_ =>  
  driver.findElement(By.name('q')).sendKeys('university of canterbury',  
    Key.RETURN))
```

```
.then(_ => driver.wait(until.titleIs('university of canterbury - Google Search'),  
  1000))
```

```
.then(_ => driver.quit());
```

Summary of what you can do

- Fetch a page:
- Locate a UI (DOM) element
- Get text values
- User input
- Move between windows and frames
- Popup dialogs
- Navigation and history (may be harder in SPA)
- Cookies
- Drag and drop
- Check out the details:

<https://www.selenium.dev/documentation/en/>

Example of what you can do

Fetch a page:

```
driver.get('http://www.google.com');
```

Locate a UI (DOM) element By ID:

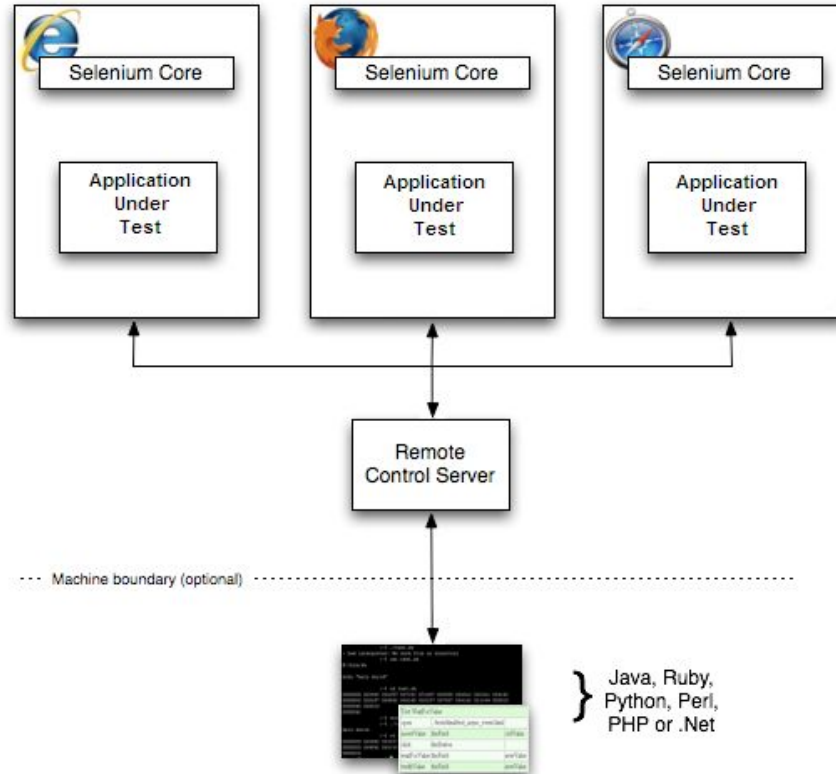
- **let** element =
 driver.findElement(By.id('someID'));
- Compare with JavaScript getElementById();

WebDriver vs the Selenium-Server

- You can use WebDriver without Selenium Server
 - Browser and tests will all run on the same machine
- However, there are reasons to use the Selenium-Server, e.g.
- You are using Selenium-Grid to distribute your tests over multiple machines or virtual machines (VMs).
- You want to connect to a remote machine that has a particular browser version that is not on your current machine.
- You are not using the Java bindings (i.e. Python, C#, or Ruby) and would like to use HtmlUnit Driver
 - HtmlUnit is a "GUI-Less browser for Java programs"

Selenium-Server

Windows, Linux, or Mac (as appropriate)...



Test Utilities for JavaScript frameworks

- Vue, React, Angular, etc. all have **test utilities**
- **JavaScript testing**, *not browser testing* (Selenium-Webdriver)
- Allows you to run **unit and integration tests**
- JavaScript **test runner**:
 - Agnostic about the framework
 - Jest, <https://jestjs.io>
 - JavaScript test runner that lets you run tests of DOM interaction
- **React Testing Library**
 - <https://testing-library.com/docs/react-testing-library/intro>

React Testing Library

```
test('loads and displays greeting', async () => {  
  render(<Fetch url="/greeting" />)  
  
  fireEvent.click(screen.getByText('Load Greeting'))  
  
  await waitFor(() => screen.getByRole('heading'))  
  
  expect(screen.getByRole('heading')).toHaveTextContent('hello there')  
  expect(screen.getByRole('button')).toBeDisabled()  
})
```



Final Exam

When: June 25, 9:30 am

Time: 2 hours

Format: Mix of short answer (incl. code), multiple choice

Where: Final Exam tab on Learn page.

Exam password will be shared on Learn and via email 5 minutes before exam starts.

No collaboration/helping others allowed, including shared documents, during exam time.