Independent degree project - first cycle

Unit testing the User Interface
Strategy for automatic testing of web UI's in React

Paul Christersson Frend
Abstract

The objective of this study was to investigate tools and techniques for automated user interface tests of components written in the Javascript libraries React and Redux for the web application Console. Console is a social network platform which provides interconnection services that bypass the public internet. The study's aim was to suggest a recommended testing strategy to help prevent regressions and improve developer workflows. The study was performed by comparing two test types: unit and end-to-end, and how they can be incorporated into the front-end testing process. Each test type was evaluated based on its strengths and weaknesses by adding test coverage to a section of Console. Requirements were developed to ensure components could be tested in isolation, that test failures generated informative messages, and that component interactions could accurately be tested. The study found that using a test technique called shallow rendering, most component tests could be moved from the end-to-end level down to the unit level. This achieved a much faster test suite which allowed for continuous execution of the unit tests and provided a tighter feedback loop for developers. Shallow rendering failed to provide enough functionality when it came to testing component interactions and events, but limited interactions could still be tested with a headless browser. Integration tests and end-to-end tests were still seen as useful required test tools, but were not deemed optimal for validating data flow through UI components. These tests were seen as beneficial more for ensuring overall system stability rather than improving developer workflows.

Keywords: Javascript, unit testing, user interface testing, test driven development, React, Redux, web applications
Table of Contents

Abstract................................................................................................................. iii
Terminology........................................................................................................ vi

1 Introduction....................................................................................................... 1
  1.1 Background and problem motivation......................................................... 1
  1.2 Overall aim.................................................................................................. 2
  1.3 Scope.......................................................................................................... 2
  1.4 Detailed problem statement........................................................................ 3
  1.5 Outline........................................................................................................ 3
  1.6 Contributions.............................................................................................. 4

2 Theory............................................................................................................... 5
  2.1 Types of testing............................................................................................ 5
  2.2 Approaches to testing................................................................................ 6
  2.3 Testing the web front-end.......................................................................... 8
  2.4 Web technologies....................................................................................... 9
    2.4.1 React....................................................................................................... 9
    2.4.2 Redux..................................................................................................... 9
    2.4.3 Javascript – latest standard (ES2015).................................................... 10
    2.4.4 Webpack............................................................................................... 10
    2.4.5 Babel..................................................................................................... 10
    2.4.6 Testing libraries..................................................................................... 11

3 Methodology.................................................................................................... 13
  3.1 Initial research............................................................................................. 13
    3.1.1 Testing React components................................................................. 13
    3.1.2 Test frameworks................................................................................... 14
  3.2 Direction of study....................................................................................... 14
  3.3 Process......................................................................................................... 15
  3.4 Measurements............................................................................................ 16
    3.4.1 Test speed............................................................................................. 16
    3.4.2 Regression safety.................................................................................. 16
  3.5 Development Environment......................................................................... 16

4 Implementation............................................................................................... 17
  4.1 Overview..................................................................................................... 17
  4.2 Implementation requirements...................................................................... 18
  4.3 Test setup.................................................................................................... 18
  4.4 Tested components..................................................................................... 19
    4.4.1 Avatar................................................................................................... 19
    4.4.2 Dropdown............................................................................................ 20
    4.4.3 Heading................................................................................................. 21
    4.4.4 Message.................................................................................................. 22
    4.4.5 NetworkHeader..................................................................................... 23
    4.4.6 CircularIconButton............................................................................. 24
    4.4.7 Editable................................................................................................. 25
Terminology

Acronyms/Abbreviations

ACL Access Control Layer. Code which controls what users see and can't see, generally based on permissions or user types.

BDD Behaviour Driven Development

CI Continuous Integration. The practice of having developers merge code into a shared repository several times a day.

DOM Document Object Model. Convention for representing and interacting with objects mainly in HTML or XML. All web browsers have an implementation.

TDD Test Driven Development. Method where tests are written first, followed by code to make the tests pass.

E2E End-to-end. A type of test which tests how multiple systems function together.

MVC Model View Controller. Architecture for separating application concerns.

REST Representational State Transfer. An architectural coding style.

SUT System Under Test

UI User Interface
# 1 Introduction

## 1.1 Background and problem motivation

The complexity of front-end applications has had tremendous growth over the last decade. According to the latest annual Stack Overflow survey, Javascript is the most popular language on the web [1]. The average amount of Javascript on a web page also increased another 23% from 2014 to 2015 while average page weight increased by 16% [2]. Applications also don’t just have more code, but a lot more business logic is taken care of client side in the browser. This leads to an increasingly complex landscape to develop web applications in, worsened by variance in browser implementations and user systems [3].

A modern web application also needs constant attention, as users expect more and more due to large companies pushing boundaries and expectations. Technologies like client side routing, offline caching, and native app like experiences means developers need to keep refactoring and adding features [4]. There’s no such thing as building once, and revising occasionally, it's a continuous cycle.

Refactoring is often also needed after tight delivery schedules cause technical debt in the code base, a symptom often observed in agile environments [5].

Automated tests is a common solution to ensure no regressions or breaking changes happen when the application needs to change [6]. Testing business logic and server side logic is a well documented practice, but testing user interfaces is usually more complex as there are so many interacting components and systems. Due to this, testing user interfaces is often analogous to end-to-end testing [7].

Testing is also an integral part of continuous integration (CI), where tests are used as a quality and confidence metric to ensure changes can be pushed to user facing environments often and with confidence [8].

For the social network application Console, an increased need for user interface testing has arisen to help refactor code and add new features.

Console is in the process of being rewritten in the front-end web tools React and Redux, so Console's front-end testing strategy can be reevaluated based on the tools and architecture these frameworks provide.
1.2 Overall aim
This study's aim will be to find valuable techniques and tools for testing Con-sole's user interfaces built with the library React. This will hopefully uncover methods which can protect the application from regressions and improve the developer experience.

1.3 Scope
The study will focus on techniques and tooling for testing front-end web components used to assemble user interfaces.

Server side rendering is ignored, and focus is placed solely on what happens in the browser.

Tests will also only be applied to components within a React / Redux architecture, however any findings or conclusions should have relevance in other component architectures as well.

Integration tests will also be ignored since they require heavy involvement with Redux and this paper's main concern is with UI components.

Machine generated tests are also left out. Whilst interesting progress has been made in the field of automated testing [9], focus will only be placed on human written tests as the act of writing them is of interest to the study.
Detailed problem statement

The main objective of this study will be to investigate test tools and techniques to find a strategy which will protect a React+Redux application from regressions and enhance the developer experience.

Main problem statement:

- What is the best way to test a React/Redux based user interface to allow fast iterative development while minimizing regressions?

Supporting problem statements:

- What points of failure exist in the components?
- What test types are needed to confidently cover the possible points of failure?
- How can component interactions be tested?
- How can integration with other systems be tested?
- Is a browser needed for testing components effectively?

Outline

Chapter 1 – Introduction gives a brief overview to the project. The motivations behind the work undertaken are examined, as well as what the project aims to achieve.

Chapter 2 – Theory – gives a brief overview to the field of testing front end web applications. The greater field of software testing in general is also briefly examined.

Chapter 3 – Methodology – describes and motivates the approach undertaken to complete the project objective. Which application part got tested as well as which tools and environment was used is explained.

Chapter 4 – Implementation – demonstrates the various approaches performed to attempt to solve the project's problem set. Various testing approaches are examined, and requirements related to the problem definition are defined.

Chapter 5 – Results – here the outcomes of the objective tests are presented, mainly relating to speed tests and regression safety.

Chapter 6 – Discussion – analyses the techniques used and highlights their strengths and weaknesses. Future recommendations are outlined.
1.6 Contributions

David Tsuji, Senior Front End Developer at Console, supported in architectural discussions, test case designs and evaluation of testing methods.
2 Theory

Testing has long been a required part of established software processes. As applications grow and scale, maintaining a functional code base becomes increasingly hard without some form of automated testing to ensure that the application still works as expected [10].

Testing is generally divided up into three types.

2.1 Types of testing

Unit testing refers to the approach of dividing up an application into its smallest moving parts, then testing these parts in isolation. It can also be referred to as white box testing, as the concern is for how the internals of each unit operates [7], [10], [11].

Integration testing is a form of black box testing where attention is focused more on how integrated components produce expected output, without concern for how each component operates internally [12].

End-to-End testing tests multiple applications or system parts as they interact with each other. This often takes the form of mimicking actions as they would be performed by an end user [13], [14].

What is being tested is usually referred to as the System Under Test (SUT). A SUT can be tested in three different ways [15]:

- Return value verification simply calls a function and inspects its returned value.

- State verification uses various methods to inspect the internal state of an object after taking an action on it, generally through public accessor methods.

- Behavior verification ensures that when a method gets called, it calls the correct subsequent method.
2.2 Approaches to testing

A general industry guideline is to split the amount of tests up by type according to a pyramid with unit tests at the base, integration tests in the middle, and end-to-end tests at the top [7], [14]. The reason is because there should be many more unit tests than integration and e2e tests, as they're faster, more specific and targeted.

![Testing Pyramid](Image)

*Illustration 1: The testing pyramid shows the ratio of test types a code base should have [7].*

There are multiple viewpoints as to which testing strategy is the right one to follow. The main debate lies between advocates of end-to-end and integration testing verses advocates for unit testing.

Developers who prefer unit tests tend to follow Test Driven Development (TDD), which is a popular approach to incorporating tests into a codebase [6]. It works by executing a continuous cycle of writing a test case, improving the software until the test passes, and then making the test case more specific and detailed so that the software once again fails the test, forcing a refactor to make it pass again. This is then repeated until the test is detailed enough to cover required business logic.

TDD also suggests keeping units small, and that units should be tested in isolation. If units depend on other services, these should be replaced with fake services to ensure that only the unit is tested, while also providing ability to control the response from external services [16].
Unit testing the User Interface – Automatic testing of web UI's in React
Paul Christersson Frend
2016-05-14

Critics of TDD however believe it places too much focus on the test-first cycle which is often adopted as the “one true way”, where developers focus more on getting passing tests than writing quality software [17]. This can lead to “an overly complex web of intermediary objects and indirection in order to avoid doing anything that's slow” [18]. If functions and algorithms are being split up to support the testing process, then the system is being destroyed [19].

Achieving isolation in unit tests which use dependencies is done with fake objects, of which several types exist. From least complex to most complex, some of the most common ones are [20]:

- Dummies
  - A dummy simply returns a value.

- Spies
  - A spy wraps an object, giving useful ways to query it, i.e. if it was called, how many times it was called, if it threw an exception, that it was called with certain arguments etc.

- Stubs
  - A stub adds basic programmable behavior to a spy, giving the developer the option to force code down a specific path.

- Mocks
  - Mocks are essentially stubs with preprogrammed expectations. A mock can thus fail a test if it's not used as expected.

All these tools are used when a SUT takes an argument which you want to replace, inspect, or analyze how it's used with a pretend object.

Generally, mocking is used more in TDD when asserting behaviors, where as stubs are are used more for asserting state [20].

Mocks by their nature couple tests to implementation, as they are more concerned with how interactions happen between systems. If interfaces change often, mocked tests are thus more likely to break than those with state assertions [20], [21].

There are two general camps in TDD, mockists and classicists. Classicists will favor state based testing and mockists will favor behavior based testing with mocks.

Overuse of test doubles can however make tests harder to understand, maintain, and give you a false sense of security that things are working properly [21]. It's thus paramount to strike a good balance between real data and fake data.
There's also competing views as to how much to test. Some advocate for 100%, but this approach is becoming less popular in favor of testing critical paths and complex business logic. Kent Beck, who is often seen as one of the modern fathers of TDD states:

IT IS IMPOSSIBLE TO TEST ABSOLUTELY EVERYTHING, WITHOUT THE TESTS BEING AS COMPLICATED AND ERROR-PRONE AS THE CODE … YOU SHOULD TEST THINGS THAT MIGHT BREAK. IF CODE IS SO SIMPLE THAT IT CAN'T POSSIBLY BREAK, AND YOU MEASURE THAT THE CODE IN QUESTION DOESN'T ACTUALLY BREAK IN PRACTICE, THEN YOU SHOULDN'T WRITE A TEST FOR IT [10].

It's also worth highlighting that code coverage has been proven to not be an accurate representation of test suite effectiveness [22]. Coverage metrics are thus of little use, and can instead contribute to architecture decay [19].

A better approach to measuring the effectiveness of a test suite could be with mutation testing. Mutation testing involves seeding a code base with different types of bugs and seeing if the test suite is able to find them. This approach has been proven to have statistically significant correlation with finding real faults [23].

Regardless of which approach is favored, most of these general test principles can be applied no matter what language is being used.

2.3 Testing the web front-end

The need for testing front-end code has risen dramatically due to aforementioned increases in business logic executed client side [3].

To solve a lot of the complexities of this increased amount of business logic, most applications use some sort of framework which comes pre-equipped with built in aids for common problems, including testing.

Testing client side code is generally quite complex as it traditionally has to involve a browser [24]. A front-end testing stack can consist of a testing framework to execute tests, a test runner which can spin up a browser for the tests, one or several assertion libraries, and additional libraries to help test specific frameworks. This same stack can usually be used to write unit, integration and end-to-end tests.

Spinning up a real browser for testing is slow, so several workarounds exist, including headless browsers which execute without graphical interfaces [25].

One of the recent trends in front-end development is the idea of user interfaces as components. Web components refer to a set of technologies to help produce new, isolated, self contained html elements [26].
2.4 Web technologies

2.4.1 React
React.js by Facebook is arguably the most popular component based front-end tool at the moment [1]. While not technically a framework, React is a UI library for building components and abstracts away when and how the UI should update in relation to data changes. It can be seen as the V in a traditional Model View Controller (MVC) structure.

React also comes with its own set of test utilities to help with component testing. React's test utilities provide two main ways to test components. The first is with a virtual DOM, where React will render the component into the document tree. For this to work a DOM needs to exist that React can hook into. Jsdom\(^1\) is a common tool for this, which is a Javascript implementation of a DOM.

The second is with a newer more experimental technique called shallow rendering\(^2\). Shallow rendering does not use a DOM at all, instead wrapping the component in an inspectable container which can be rendered one level deep.

React however couples a component's concerns (Javascript, HTML, and CSS) in a single file, so can be seen to go against the separation of concerns which has been so prolifically preached in software development, especially with the MVC pattern [27].

One common pattern to achieve separation of concerns and component reuse in React is to create container components and presentational components [28]. A container component does data fetching and then renders an underlying presentational component.

React does not offer much help on how to structure applications, leaving this to be solved by the developer. Facebook recommends a pattern called Flux\(^3\) for this, of which there are many implementations. One of the most popular Flux implementations is called Redux.

2.4.2 Redux\(^4\)
Redux is a state container for Javascript applications. It works by splitting an application up into actions, reducers and a global immutable store. Any changes to the store has to be expressed as a request in the form of an action. That action then passes through one or many reducer functions. Reducers are pure functions which based on an action and a state, creates a new state.

---

1 https://github.com/tmpvar/jsdom
2 https://facebook.github.io/react/docs/test-utils.html#shallow-rendering
3 https://facebook.github.io/flux/
4 http://redux.js.org/docs/introduction/
2.4.3 **Javascript – latest standard (ES2015)**

Javascript finalized the specification for its latest language version 6 in 2015, bringing lots of new language features including arrow functions, object destructuring, rest parameters and template strings. The official name of this specification is Ecmascript 2015 (ES2015).

2.4.4 **Webpack**

Webpack is an opinionated bundler and build tool with a large ecosystem of plugins. It enables automation of build tasks, and ultimately compiles modules down to static assets for serving up on the web.

2.4.5 **Babel**

Babel is a transpiler which can rewrite the latest Javascript standard down to an older standard so that you can still write with new features without breaking in older browsers. As an example the following ES2015 code uses object destructuring with new constant types.

```javascript
var object = {foo: 1, bar: 2, baz: 3}
const {foo, bar} = object;
```

*Figure 1: ES2015 object destructuring *

Babel would then transpile the second line of code to ES5 (the previous language version), like follows.

```javascript
var foo = object.foo, bar = object.bar;
```

*Figure 2: Transpiled ES2015 code to ES5 *

---

5 [http://www.ecma-international.org/ecma-262/6.0/](http://www.ecma-international.org/ecma-262/6.0/)
6 [https://webpack.github.io/docs/](https://webpack.github.io/docs/)
7 [https://babeljs.io/](https://babeljs.io/)
2.4.6 Testing libraries

Mocha.js

Mocha is a test framework running on Node.js. It doesn't provide its own assertion methods, so additional libraries need to be used for that. It can execute both on the browser and in server, runs tests serially, and has multiple built in reporting methods.

Chai

Chai is an assertion library that comes with three modules providing different testing styles:

- Should
  - BDD style method chaining.
  - Decorates the global Object prototype for declarative expressions.
- Expect
  - BDD style assertions.
- Assert
  - Classical TDD style assertions.

Enzyme

Enzyme is a helper library which wraps around React's own testing utilities. It provides a much easier API to work with by mimicking jQuery's selectors.

---

8 https://mochajs.org/
9 http://chaijs.com/
10 https://github.com/airbnb/enzyme
Nightwatch

Nightwatch is an end-to-end testing framework which runs on top of Selenium. Selenium is a browser automation tool written in Java, which many test libraries wrap around.

Nightwatch tests are written by constructing page objects to represent the pages to be tested. Page objects consist of sections which hook into specific parts of the HTML, and commands which take actions against the page like clicking buttons and submitting forms. This additional layer is there to create an API which is more human-like by abstracting away the underlying HTML actions required to manipulate the page. Test scenarios then consume the page objects to create more declarative tests.

Nightwatch also comes with multiple test runners to easily execute tests in different browsers.

11 http://nightwatchjs.org/
12 http://www.seleniumhq.org/
3 Methodology

3.1 Initial research

The study was commenced by researching best practice approaches and tools to testing React applications and user interfaces.

Tools were evaluated based on what execution environments they provided, such as real browsers, headless browsers, virtual DOM's and fake DOM's.

3.1.1 Testing React components

React comes with its own test utilities which provides options to wrap components in containers which can be inspected and rendered.

Out of the multitude of techniques and tools encountered, a subset were selected for the study:

- Component testing with React's test utilities
  - Shallow rendering
  - Headless browser testing with jsdom
  - Enzyme as a helper library for both shallow and headless testing

Shallow rendering was selected as it's the recommended approach by Facebook [29], and offers benefits to speed and isolation by not using a DOM.

Enzyme was selected as a helper library as it offers a much more declarative way of querying and traversing wrapped components compared to React's own test tools.

React's test utilities does not come with a test framework or an assertion library, so this also had to be selected.
3.1.2 **Test frameworks**

Mocha, Tape, Jasmine and Qunit were compared as potential test frameworks. Mocha was chosen because it was already in use by Console's API team, it can operate both in the browser and on the server, and it’s not opinionated in its use of assertion styles.

For the assertion library, Chai, Should and Assert were compared. Chai was chosen since it comes with multiple test styles. This enabled comparison of the different styles, as well as the ability to combine techniques without using several libraries.

For the end-to-end frameworks Nightwatch was compared against Protractor and Casper.

Nightwatch was chosen for its use of page objects and framework agnostic design. Protractor was rejected since it's primarily designed for Angular applications, and Casper didn't offer any way of executing the tests in a real browser. Nightwatch also integrated well with Mocha as the underlying test framework.

3.2 **Direction of study**

The application was analyzed to find a suitable section which would suit experimenting on with different test methods, based on the defined problem statements.

It was decided to write tests against real application code instead of a sample project. Whilst this provided less flexibility in creating different scenarios, it kept the tests applicable to a real world environment.

The selection requirements were:

- The page can only contain React components
- Some components must communicate
- Some operations in the section must be asynchronous
- The page must contain different levels of Access Control Layer (ACL)
- The page should have no current tests in place

With these requirements in mind the connection detail section was chosen, which lets a logged in user view and modify an existing network connection based on their granted level of access.
3.3 Process

Testing started with a bottom up approach where the smallest components were tested first. The reason for this is small components have the lowest cyclomatic complexity, and it was decided it would be better to have smaller components tested before they were being consumed by larger components to ensure the same logic wasn't tested multiple times.

Cyclomatic complexity refers to the amount of arguments and the amount of branching/ variations a component can produce [6].

Different test methods were then experimented with and analyzed from a speed perspective, with the aim of getting the tests to run as fast as possible.

Due to the nature of Redux, all tested components are stateless, so focus was placed on two types of tests:

- Return value verification
  - The component's rendered output was observed based on the variance in its inputs.

- Behaviour verification (testing interactions)
  - Given a supplied set of inputs, will the component interact with these inputs in an expected way when under the influence of external events.

The general approach was to analyse a component's concerns, and then write tests asserting expected output given its varying sets of inputs.

Here different assertion strategies were used to see which provided the best failure messages.

Once the testing methods were satisfactory for the presentational components, the same approach was followed for the container components. Here special attention was given to covering all code paths related to ACL.
3.4 Measurements

3.4.1 Test speed
Speed of execution was measured by the reported output of the testing tools.

For the unit tests this was reported by Mocha and for the end-to-end tests this was handled by Nightwatch, both from the command line reporter after execution.

3.4.2 Regression safety
Regression safety was measured by performing a simplified form of mutation testing. A common set of bugs were defined as measurement criteria. Once tests were written, these bugs were then seeded through the application manually to see if they would elicit failures in the test suite.

Based on subjective experience from bugs logged against the Console code-base, as well as reports on common application bugs [30]–[32], the following list of bug types were chosen:

- Syntax errors
- Logic errors
- Runtime errors
  - Property access on undefined objects
  - Unexpected input types as parameters
  - Event handling failure
  - External API failure
  - Styling / presentation errors

An explanation of each type can be viewed in appendix A.

3.5 Development Environment
All tests were written in Webstorm 2016 on OSX as the platform. Tests were executed on Node.JS version 4.2.1.
4 Implementation

4.1 Overview

The implementation consisted of tests on the connection detail page, which contains both presentational and container components.

*Illustration 2: Screenshot of the connection detail page*
4.2 Implementation requirements

Console's components are designed to be used across different sections and even other products. With this in mind, several requirements for the component tests were defined:

- **Components should be tested in isolation.** No changes to parts of the system outside of the component should be able to make the tests for a presentational component fail.

- **Tests should fail with informative messages.** This is so developers quickly can identify the root cause and spend their time fixing the issue rather than searching through the codebase.

- **Components should be tested based on structure and behaviour.** Both return value and behaviour should be testable.

4.3 Test setup

Before any tests could be written, the unit testing environment in Mocha had to be set up correctly. The following had to be setup for the testing tools to work:

- A virtual DOM needed to be added to Mocha's global object to provide a DOM for React's test utilities

- Mocha had to be setup to understand ES2015 code

- Mocha had to be setup to ignore style imports in components

This setup was placed in a file for Mocha to execute when it started. This boilerplate was put in an npm script so that tests could be started by simply running “npm test”.

```
"test": "mocha --require test/setup.js src/**/*.tests/**",
```

*Figure 3: alias for executing tests with npm*

Mocha comes built with functionality to watch directories for file changes, so the following alias was setup to be able to target all component test files.

```
"test:watch": "mocha --watch --require test/setup.js src/**/*.tests/**"
```

*Figure 4: alias for continuous test execution through npm*

Tests could then be executed continuously when code changes were made to provide instant failure information to the developer.
4.4 Tested components

4.4.1 Avatar

The avatar is responsible for displaying a user's profile picture, or a group of profile pictures based on usernames passed in as props. If more than 3 usernames are sent in, the component should show a link with a + and the number of remaining users in the group.

Figure 5: Avatar component with variations

The avatar was tested with shallow rendering only. The main technique used was to assert that the correct amount of image elements were rendered based on the amount of usernames in the input array.

```javascript
it( 'should render three avatars', () => {
  const wrapper = shallow(<Avatar apiBaseUrl="apiBaseurl" usernames={['user1', 'user2', 'user3']} />);
  expect( wrapper.find( '.ui.image' ).length ).to.equal( 2 );
});
```

Figure 6: Avatar test example with shallow rendering

When this test failed the output explains the difference between the expected lengths.

```
1) <Avatar /> should render three avatars:

   AssertionError: expected 3 to equal 2
   + expected - actual

     -3
     +2
```

Figure 7: Avatar test output
4.4.2 Dropdown

The dropdown is a wrapper around a dropdown component from the UI library Semantic UI\textsuperscript{13}. It provides the ability to have enhanced select lists, for example with built in search functionality, improved styling, multiple selects and other features.

The dropdown could be tested with mostly shallow rendering, but it was not possible to test interactions with it, for example ensuring menu items were visible once it was clicked. These interactions were not possible with Jsdom either, as Mocha needed both the Semantic UI library and jQuery on the window object to be able to instantiate the menu. Once this was added to the Mocha setup file no more errors were produced, but clicks still failed to simulate properly.

After investigating Semantic UI, it was discovered it has it's own event handlers which use event delegation from higher up in the tree, so using React's simulate would not trigger these events.

\textsuperscript{13} http://semantic-ui.com/
4.4.3 Heading

A heading tag which simply wraps html heading elements. It doesn't offer any additional functionality outside of the standard html version.

The tests use shallow rendering and simply assert that the component can be created. This has the effect of ensuring no trivial errors exist in the component.

```javascript
it( 'should render', () => {
    const wrapper = shallow( <Heading /> );
    expect( wrapper ).to.exist;
});
```

*Figure 8: Basic shallow render test to protect against syntax errors*
4.4.4 Message

Please see your company keyholder for the ability to view connections

Illustration 5: Screenshot of Message component

A generic message component to be able to display notifications to the user. Can be displayed in different modes like warning, info or error.

It wraps the children you pass in in a div with specific styling. The message component was tested with shallow rendering to verify the component exists, as well as to render the component to html, and assert the generated html matched expected output.

```javascript
it('should render children', () => {
  const wrapper = shallow(<Message>hello</Message>);
  expect(wrapper.html()).to.equal('div class="ui message">hello</div>');
});
```

Figure 9: Assert based on html output

Above test would fail like follows:

```
1) <Message /> should render children:

  AssertionError: expected '<div class="ui message">hello</div>' to equal '<div class="ui message">bye</div>'
+ expected - actual

-<div class="ui message">hello</div>
+<div class="ui message">bye</div>
```

Figure 10: Failed assertion from comparing html output strings

Since the rendered output of the component is relatively small, finding the diff is trivial.
4.4.5 NetworkHeader
A specific header relating to Console's network section to provide consistency between all headers in that area of the site.

NetworkHeader simply wraps the passed in children with expected styling. It's generally used with a link to the left and an action button to the right.

Shallow rendered verification was the only test method used.

```javascript
it('should render', () => {
  const wrapper = shallow(<NetworkHeader />);
  expect(wrapper).to.exist;
});
```

Figure 11: Basic existence test for the NetworkHeader

With the requirement of testing components in isolation it made little sense to complicate these tests further with variations in passed in children.
4.4.6 CircularIconButton

This is a round action button with an icon in the middle.

While this component looks visually simple, it's got a few intricacies, mainly being able to switch between being a button element or an anchor element.

Testing involved length based assertions mixed with regular expression based assertions to test that expected classes, properties, and elements were being generated by the component.

```javascript
it( 'should set a class on the wrapper for the prop `type`', () => {
  const wrapper = shallow( <CircularIconButton type="basic" /> );
  expect( wrapper.find( 'button.basic' ).length ).to.equal( 1 );
});

it( 'should assign `link` to the `href` attribute and generate an `a` tag', () => {
  const wrapper = shallow( <CircularIconButton link="//wert" /> );
  expect( wrapper.find( 'a' ).html() ).to.match( /<a.+href="//wert/ );
});
```

Figure 12: Regular expression based assertions
4.4.7 **Editable**

Editable is a text field, which when clicked changes to an input field made for updating a value. This is a complex presentational component which consists of several smaller components: The div containing the value to update, an action button to enter edit mode, a form inside edit mode, an input field for updating the value, and accept, and cancel buttons.

**Illustration 8: Screenshots of Editable in both states**

Testing here mainly involved using spy functions to ensure correct callback functions passed in as props were being executed, as well as that the logic for switching between modes was working correctly.
4.4.8 Toggle

Similar to a checkbox, but with a sliding toggle as can commonly be seen in iOS apps. Only shallow rendering was needed for the tests, and adequate coverage was achieved by ensuring that the toggle was either checked or not checked depending on its properties.

```javascript
it( 'should be checked', () => {
  const wrapper = shallow( <Toggle checked={ true } /> );
  expect( wrapper.find( 'input' ) ).to.be.checked;
});
```

Figure 13: Asserting input field states with jQuery like assertions

“Checked” is a special assertion helper from the chai-jquery library which adds jQuery inspired selectors, especially useful for form and input assertions. However omitting this library would not make the tests fail, i.e. “be.checked” still passed even when the element wasn't checked, leaving room for error.

A separate test asserted that the callback function passed in to the toggle executed on click. Shallow rendering successfully simulated this interaction.

```javascript
it( 'should click once', () => {
  const onChange = sinon.spy();
  const wrapper = shallow( <Toggle selected="true" onClick={onChange} /> );
  wrapper.simulate( 'click' );
  expect( onChange.calledOnce ).to.equal( true );
});
```

Figure 14: Asserting behavior with simulated clicks
4.4.9 **HorizontalTabs**

![Connections, Console Routers, Utilization]

*Illustration 10: Screenshot of HorizontalTabs with three child elements*

This component lets you group content under tabs, and takes props for which tab should show, as well as callbacks for when tabs are clicked.

It depends on sub components called HorizontalLinkTab to provide the menu items and the content to go inside. They simply convert to styled anchor tags. It also accepts HorizontalTab child components, which show or hide their children based on visibility properties.

While it may seem that this group of components have a few moving parts, they don't have to maintain any internal state themselves, since it's being stored in Redux. So all that needs to be tested is that it can take children, that only one of the children show based on a property, and that it executes callbacks on click.

These tests are split up between the wrapping component and the sub components, all shallow rendered.

It was not deemed necessary to test how these components work together, as having them tested in isolation was sufficient. There would be little benefit in asserting that a callback gets executed on a tab item that's been inserted as a child of the tab menu, as that's already been tested in the tab item's own set of tests.

4.4.10 **Icon**

*Illustration 11: Two icon components*

Simple component for displaying icons. Wraps an i tag with predefined styling, and takes type as a property, which is injected as a class.

Only a few shallow rendered tests were written due the low complexity of the component.
4.4.11 Loader

Illustration 12: loader component in its inline mode

Loader is a generic component used when something is loading. It can operate in two modes, as an inline loader which simply shows a spinner, or as a dimmer, which also dims out all the content in the wrapping container element where it is used.

Here the assertion “contains” was used for the validation.

```javascript
it('should render a loader', () => {
  const wrapper = shallow(<Loader />);
  expect(wrapper.contains(
    <div className="ui basic center aligned segment">
      <div className="ui active inline loader"></div>
    </div>
  )).to.equal(true);
});
```

Figure 15: content validation with contains assertion

However, when this test failed, it would produce the following output.

```
1) <Loader /> should render a loader:

   AssertionError: expected false to equal true
   + expected - actual
     -false
     +true
```

Figure 16: Contains test failure output

Knowing what doesn't match here is not obvious, so would require further inspection and work from the developer before the real issue can be found.
### 4.4.12 Connection Detail

The connection detail is the wrapping component for nearly everything on the connection detail page.

It's the only container component on the page, so its props will be bound to Redux actions and state. It then passes these props down to the presentational components nested within it, which have no awareness of Redux.

The first approach used when testing was to treat it more like an integration test. For this to work a fake store was set up, and the component was wrapped in a Redux Provider component, which is necessary for when it binds to Redux.

```javascript
let wrapper = mount(
  <Provider store={ store }>
    <ConnectionDetailSection>
    </ConnectionDetailSection>
  </Provider>
);
```

*Figure 17: Wrapping a redux aware component in Provider will give it access to certain Redux properties*

*Illustration 13: The ConnectionDetailSection, the only component of those tested with awareness of Redux*
To set up the store properly the logged in user also had to be mocked and inserted into the store.

The component could then be tested by dispatching relevant actions and observing how it responded to changes in the store from these actions.

Some of the more critical tests for this component were those concerned with the access control layer (ACL). The related user can have three permissions which affect the component's output:

- Can the user view connections
- Can the user create connections
- Can the user delete connections

To test these scenarios, different user objects with faked permissions had to be set up, and the component could then be mounted again with different user permissions to see how it would respond. This method was quite cumbersome as it involved a lot of boilerplate.

Another example would be testing the loading state of the component. For this to work the store had to be set up with an empty connection. Then the action for requesting a connection had to be dispatched before an assertion that a loader existed could be run. Testing this way is fine if a blackbox approach is desirable, but for the connection detail the main concern is how data flows through the component, and how it reacts to varying inputs. Not exposing these inputs thus hides aspects of the component the developer is concerned about.

There's also little benefit to dispatching actions and mocking stores when the tests aren't concerned with these operations. The connection detail component also has a lot of complexities involved where users can accept / reject / withdraw connections to each other, so testing all these scenarios with the blackbox approach would not scale.

The second approach was to split the container component into two versions, the container component and the underlying presentational component.

```javascript
export { ConnectionDetailSection };
export default connect(
  mapStateToProps,
  mapDispatchToProps
)(ConnectionDetailSection);
```

Figure 18: Splitting up a container component in two made testing a lot easier
This approach enabled testing the whole component in a more isolated manner, where focus could be on the variations of inputs rather than how it reacted to changes in the store.

This created a nice API, as the test could simply import the presentational version, whilst the real application imported the default container version. The only different between the two is the container has the Redux bindings.

Using the previous example of testing loading state, the presentational component could now simply be set up with its isFetching prop to true, ignoring any redux bound operations.

```javascript
it('should show a loader if loading', () => {
    const wrapper = shallow(<ConnectionDetailSection canReadConnection isFetching />);
    expect(wrapper.type()).to.equal(Loader);
});
```

Figure 19: Testing display logic of sub components with shallow rendering

The redux bindings were then tested separately if complex, or not at all as they were often trivial functions.

To test the ACL, all that was needed was to modify the canCreateConnection, canDeleteConnection and canEditConnection boolean props the component accepts.

```javascript
const wrapper = shallow(<ConnectionDetailSection
    canCreateConnection=false
    canReadConnection=true
    connection={receiverConnection} />);
```

Figure 20: Testing ACL with shallow rendering

The shallow renderer was now also more suitable, as tests had already been written on the unit level for the nested components.

The majority of tests written were mainly concerned with the state of subcomponents based on the variance in the connection-detail-section's props, similar to the loader in figure 19.
4.5 Testing in isolation

To test components in isolation they can not have any hard coded dependencies. In Javascript this can easily happen as functions can access objects outside of their scope. A common pattern is to require in a dependency before a function declaration, and then access the external dependency from within it.

Any components that had these hard coded dependencies were refactored to take them as props instead.

This made it trivial to replace them when testing and also helped improve the purity of the components. Mocking or stubbing could now be ignored as the components' input were under full control.

Some exceptions to this rule were made. Two of the used libraries, lodash\(^{14}\) and moment\(^{15}\), were used so heavily that passing them in as props everywhere wasn't worth the effort. Constants were also consumed from outside of the components scope in most cases.

The testing type that suited isolated tests the best was shallow rendering. Since shallow rendering only renders one level deep, any nested components are left untouched. This suited unit tests perfectly, since a nested component can be seen as a hard coded dependency. Shallow rendering essentially mocks these lower components out.

4.6 Testing component interactions

For presentational components spy functions were used as props. Interactions like clicks could then be spied on and asserted that callbacks were called as expected. Enzyme's simulate method was used to simulate events as it offered the same API regardless of using the shallow renderer or the virtual DOM.

Shallow rendering worked well as long as the callback got executed in the same component. If lower than one level deep a virtual DOM had to be used instead. This also provided the opportunity to hook into React's component lifecycle methods if required.

Neither shallow rendering or virtual DOM methods worked well when testing components which wrapped external libraries like Semantic UI\(^{16}\). Simulating a click would not trigger the event, so the only way to get it to work was with a slower end-to-end test.

---

\(^{14}\) [lodash.com/](https://lodash.com/)
\(^{15}\) [http://momentjs.com/](http://momentjs.com/)
\(^{16}\) [http://semantic-ui.com/](http://semantic-ui.com/)
4.7 End-to-end tests

The majority of data interactions for the Console front-end happens against a REST inspired Node.JS API, which has its own test suite.

Nightwatch test scenarios were built to ensure the UI components worked together as a whole and that they demonstrated expected behaviour against a real API.

Smoke tests were written for log in, log out and visiting the connection page, while specific tests for the connection detail page handled enabling and disabling a connection.

To target the correct elements, a custom HTML attribute called "iht" was used to not couple test targeting to class names and ID's. This approach did not work well with React as it strips custom attributes from html, so the approach was changed to target name-spaced class names.

4.7.1 Nightwatch setup

To test the connection detail page three pages had to be visited. First the home page to be able to login, which in turn redirects to the activity page on success, then once there it's possible to navigate to the connection detail.

For this to work a database state which had two users and an active connection between them had to be setup. This was performed by building state from using the system and then exporting the resulting database.

Page objects were then set up for the home page, the activity page, and the connection detail page.

Nightwatch was configured with the Chrome web driver, which will cause it to execute the tests in a real Chrome browser.
4.7.2 Smoke tests
Two smoke tests were written, one for logging in and logging out, and one for visiting the connection page.

```javascript
module.exports = {
    before: partials('exec', 'IHT_PRETEST'),
    after: partials('exec', 'IHT_POSTTEST'),

    'User-A logs in': partials('console-login', {
        username: process.env['IHT_CONSOLE_USER_A_EMAIL'],
        password: process.env['IHT_CONSOLE_USER_A_PASSWORD']
    }),

    'User-A visits network page': partials('console-connections'),

    'User-A logs out': partials('console-logout', true)
};
```

Figure 21: Login test scenario example

Two commands were created for the homepage page object, signIn and signOut. These were then used from the smoke test scenario.
Unit testing the User Interface – Automatic testing of web UI's in React
Paul Christersson Frend 2016-05-14

```javascript
signIn: function (_data ) {
    var signInForm = this.section.signInForm.elements,
        page = this.api.globals.page;

    return this.navigate()
        .waitForElementVisible(signInForm.emailInput.selector)
        .setValue(signInForm.emailInput.selector, _data.username)
        .setValue(signInForm.passwordInput.selector, _data.password)
        .click(signInForm.submitButton.selector)
        .waitForElementVisible(page.userNav.selector);
}
```

Figure 22: The signIn command on the homepage object which fills out the login form.

4.7.3 Enable / disable connection

To test enabling and disabling the connection the login scenario was reused from the smoke test. Additional commands for enable and disable connection were created for the connection page object.

The commands just click on buttons and wait for expected results to happen on the page in sequence. Some special selectors had to be used to be able to wait for content to change before the command navigated away to a different page.

Figure 23 Shows how the test waits for the status element to contain the text 'Active'. It will wait 1000 ms for this to happen before it will fail the test.

```javascript
expect.element(connection['status'].selector).text.to.contain('Active').before(1000)
```

Figure 23: Waiting for content to appear during test execution
5 Results

For our component tests, almost all of the tested components were able to be tested with shallow rendering. The connection detail and the dropdown were the only two components that needed more complex tests.

Table 1: Generic Test Summary

<table>
<thead>
<tr>
<th>Component</th>
<th>Component type</th>
<th>Test amount</th>
<th>Test types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection Detail</td>
<td>Container / Presentational</td>
<td>25</td>
<td>Shallow, mount, end-to-end</td>
</tr>
<tr>
<td>Dropdown</td>
<td>Presentational</td>
<td>7</td>
<td>Shallow, mount</td>
</tr>
<tr>
<td>Heading</td>
<td>Presentational</td>
<td>4</td>
<td>shallow</td>
</tr>
<tr>
<td>Message</td>
<td>Presentational</td>
<td>4</td>
<td>shallow</td>
</tr>
<tr>
<td>NetworkHeader</td>
<td>Presentational</td>
<td>1</td>
<td>shallow</td>
</tr>
<tr>
<td>CircularIconButton</td>
<td>Presentational</td>
<td>8</td>
<td>shallow</td>
</tr>
<tr>
<td>Editable</td>
<td>Presentational</td>
<td>1</td>
<td>shallow</td>
</tr>
<tr>
<td>Toggle</td>
<td>Presentational</td>
<td>3</td>
<td>shallow</td>
</tr>
<tr>
<td>HorizontalTabs</td>
<td>Presentational</td>
<td>4</td>
<td>shallow</td>
</tr>
<tr>
<td>Avatar</td>
<td>Presentational</td>
<td>6</td>
<td>shallow</td>
</tr>
<tr>
<td>Icon</td>
<td>Presentational</td>
<td>3</td>
<td>shallow</td>
</tr>
<tr>
<td>Loader</td>
<td>Presentational</td>
<td>2</td>
<td>shallow</td>
</tr>
</tbody>
</table>

Total: 63
5.1 Test type breakdown
The total breakdown of unit tests with React's test utilities vs end-to-end tests with Nightwatch can be seen in Figure 24: Breakdown by test type.

![Figure 24: Breakdown by test type](image)

5.2 Test speed
The execution time of the tests were compared by type. As observed in Table 2, the shallow rendered tests executed in 1 second, whilst the end-to-end tests took 32 seconds combined. Out of the end-to-end tests, the login smoke test was the slowest at 10.5 seconds.

<table>
<thead>
<tr>
<th>Test method</th>
<th>Total test amount</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shallow</td>
<td>63</td>
<td>1s</td>
</tr>
<tr>
<td>End-to-end</td>
<td>5</td>
<td>32.6s</td>
</tr>
</tbody>
</table>
5.3 Regression safety

Regression safety was tested by seeding the tested components with various types of errors and observing if the tests would fail. Both the unit tests and the end-to-end tests were tested.

The unit tests would in general fail much closer to the actual issue. Syntax errors would not cause failing tests if shallow rendering was used and the error was in a sub component as it only renders one level deep.

Logic errors failed if the test was testing for expected output.

Interaction errors failed as long as they weren't reliant on external API's with custom event handling.

Table 3: Regression safety – unit tests

<table>
<thead>
<tr>
<th>Error type</th>
<th>Fails a test</th>
<th>Error proximity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax errors</td>
<td>Yes</td>
<td>Close</td>
</tr>
<tr>
<td>Logic</td>
<td>Yes</td>
<td>Close</td>
</tr>
<tr>
<td>Property access</td>
<td>Yes</td>
<td>Close</td>
</tr>
<tr>
<td>Unexpected inputs</td>
<td>Yes</td>
<td>Close</td>
</tr>
<tr>
<td>Event / interaction</td>
<td>Sometimes</td>
<td>Close</td>
</tr>
<tr>
<td>External API failure</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>Styling / presentation</td>
<td>No</td>
<td>N/A</td>
</tr>
</tbody>
</table>
For the end-to-end tests, more failures could be covered with less tests but the information concerning the error was often located far from the root cause.

HTML syntax errors were not always detected as these can fail silently based on the strictness of the browser. React's test utilities for the unit tests had stricter evaluation policies.

Unexpected input errors failed if they caused runtime errors, but could also slip through undetected. Presentational errors would only fail if they were explicitly tested for, i.e. by measuring the distance between elements or testing for the presence of a color.

<table>
<thead>
<tr>
<th>Error type</th>
<th>Fails</th>
<th>Error proximity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax errors</td>
<td>Sometimes</td>
<td>Far</td>
</tr>
<tr>
<td>Logic</td>
<td>Sometimes</td>
<td>Far</td>
</tr>
<tr>
<td>Property access</td>
<td>Yes</td>
<td>Far</td>
</tr>
<tr>
<td>Unexpected inputs</td>
<td>Sometimes</td>
<td>Far</td>
</tr>
<tr>
<td>Event / interaction</td>
<td>Yes</td>
<td>Far</td>
</tr>
<tr>
<td>External API failure</td>
<td>Yes</td>
<td>Far</td>
</tr>
<tr>
<td>Styling / presentation</td>
<td>Sometimes</td>
<td>Far</td>
</tr>
</tbody>
</table>
6 Discussion

6.1 Regression safety

As can be seen in Table 4 and Table 3, neither unit tests nor end-to-end tests were able to confidently protect against all the types of bugs introduced to the components by themselves. However their combined coverage did manage to report all seeded bugs. There are naturally a lot more bug types than those used for this study, but this result at least indicates the usefulness of a combined approach.

The fact that the unit tests failed much closer to the location of the real issue validates why a code base should have more unit tests than integration tests. Finding the root cause for the failed unit test was often trivial, whereas the failed end-to-end test required a lot of investigation.

One concern often raised especially when testing dynamic languages is that too many tests are written that don't really test anything of value, for example tests which ensure arguments are of a certain type [19]. An unexpected benefit of testing React components was that none of these test types were needed as React offers the ability to define property types and defaults in the component definition. The actual tests could then focus on asserting business logic instead. This made the process of writing tests easier and more enjoyable.

Even though tests were easy to write, it was decided that testing every path through a component would not be a feasible strategy to follow. There are simply too many variations to achieve this, so emphasis should instead be placed on testing critical paths that have external business value, or paths which are deemed likely to break. Critical errors not tested for on the unit level can still be caught in an e2e test, it will just be harder to find out what failed.

The method used for mutation testing could be improved by adopting a more rigorous, scientific approach. Mutations could also be machine generated rather than manually inserted, providing a larger data set to measure.
6.2 Execution speed
Since the unit test suite completed in one second, it can easily be executed continuously during development thanks to Mocha's built in file watching functionality.

This should lead to improvements in developer confidence when it comes to refactoring, since as long as the tests stay green the health of the component code base should be ok. Having the code base protected by tests should hopefully also encourage experimentation with novel, interesting problem solutions.

6.2.1 Protecting continuous test execution
Based on the results, two recommendations were developed to ensure that the execution time of the unit test suite doesn't start increasing.

Firstly, no asynchronous operations should be hard-coded into presentational components or be consumed from a scope external to the component.

Secondly, slow tests which include slow operations should be placed in a separate folder or named differently so that they can be excluded from the watch patterns of the unit test suite. This can be done once the speed of the test suite starts becoming an issue, as Mocha will alert which tests are running slow.

6.3 Shallow rendering
One of the biggest wins of the implementation was discovering the usefulness of shallow rendering. It made testing UI components trivial, and allowed focus on data flow through the component without having to deal with the browser environment.

Most errors tended to come from components reacting unexpectedly to changes in their inputs. Shallow rendering solved this by ensuring that each component can be unit tested in isolation, providing a fast, intuitive way to test invariants.

One of the main issues with shallow rendering was that the simulate method was not that effective at simulating complex events. It would be great to see improvements to this method in the future so that more events can be tested on the shallow level.

Shallow rendering can also give you a false sense of security since the component is not being tested in a real browser environment.

This 'real' testing is however better done with end-to-end tests which can ensure expected behavior is present across multiple browsers and environments. Its also worth noting that the output a component can render does not vary between browsers, only how browsers interpret that output.
6.4 End-to-end tests

Writing the end-to-end tests involved a lot of boilerplate to set up the page objects correctly for the pages under test. However, a lot of this boilerplate could be reused as page actions were often required multiple times. This decoupling pattern offered by Nightwatch worked really well.

Coming up with a way to avoid brittle test design was challenging, as React would not allow the use of custom html attributes for selectors. This left no other option but to use classes and ID's for targeting elements.

It would be great if Nightwatch could evolve to allow specifying React components as targets. This would remove the need for targeting the underlying structure of the component, which causes brittle selectors. This would have the effect of information hiding, and the component could be interfaced with through its type rather than its underlying structure.

Even though only three test scenarios were written which covered the connection-detail page, they were enough to ensure the components were working as expected.

One of the more interesting comparisons between end-to-end testing and shallow rendering can be made in relation to ACL testing. One could argue that this is one type of test where end-to-end testing is much better suited, however this study found that ACL testing with shallow rendering was sufficient.

Testing all ACL variations with end-to-end testing could quickly get out of hand if there are multiple paths to test per page.

Due to the slow nature of the end-to-end tests and the ability to cover complex UI logic with shallow rendering, not a lot of end-to-end tests were needed. While they are vital for ensuring product stability, these tests are probably of higher value to the business than the individual developer.

It also doesn't make sense to have developers execute test suites which take several minutes, as this would produce a lot of wasted time. The e2e tests are thus better suited for execution by a Continuous Integration tool like Travis or Jenkins [33].
6.5 Tests affecting component design

Another interesting outcome was how component design was affected by the testing process. This is not necessarily a bad thing, just something that needs to be taken into consideration.

During the implementation components were occasionally rewritten to reduce their amount of inputs, reducing cyclomatic complexity. Testing also promoted early refactoring and introduction of more subcomponents as this made it easier to write simple assertions.

Care however needs to be taken to ensure that the process of testing the component doesn't change a simple design to complex just for increased testability.

6.6 Redux impact on testability

One benefit of using Redux for component state was the fact that all state usually internal to the component could be moved out to the Redux store and mapped as input data.

This eliminated the need for testing internal state of functional components, as they never contained any. This required a pretty large mind shift in how components were designed however. It took heavy adjusting to move away from storing internal component state (e.g. wasClicked, activeColumn etc), to having it passed in as properties.

6.7 Recommended assertion style

Enzyme's jQuery like API was highly effective at declaratively targeting elements to be asserted.

In terms of assertion styles, The final approach was to emphasize those which generated useful failure messages.

The slowest assertions were those using regular expression matching or where entire subtrees were rendered out for the assertion diff.

Testing based on boolean expressions was declarative and easy to write, however it produced confusing failure messages of the form “expected true to be false”. This was not very informative since it lacked context of why the test failed.

Testing by asserting rendered html output worked well for components which don't include other components, but when the whole component tree got rendered out the test became too brittle as the parent's test would fail whenever a child changed its structure.
6.8 Conclusion

As described by Wacker [14], a testing strategy can not just be evaluated by how it finds bugs, it must also be evaluated on how it enables developers to fix and prevent them.

By measuring speed of execution and regression safety, this study has demonstrated how React applications can effectively be tested to help prevent applications from regressions and increase developer confidence.

Shallow rendering on the unit test level was the most useful tool for this.

Whilst previously some sort of browser environment was always required to run tests against user interfaces, removing this overhead has made it much easier to reason about components in isolation. It also promotes early refactoring, encouraging organic improvement of the code base.

If referring back to the testing pyramid described by Fowler [7], React's test utilities allow the UI tests to move from their traditional spot at the top all the way down to the bottom.

These tools and techniques offer exciting advancements in testing approaches which can hopefully pave the way for more novel methods to testing the user interface in the future.
References


Unit testing the User Interface – Automatic testing of web UI's in React
Paul Christersson Frend 2016-05-14


Unit testing the User Interface – Automatic testing of web UI's in React

Paul Christersson Frend

2016-05-14


Appendix A: Explanation of bug types

Syntax errors
Syntax errors are errors in the sequence of characters which are written in the program. Compiled languages won't compile with syntax errors, but for dynamic languages like Javascript, syntax errors aren't usually detected until execution of the code block. If using a transpiler like Babel however, most syntax errors will be caught in the transpilation step.

Logic errors
Logic errors are seen as the hardest to find, as they usually won't cause the program to break. They'll instead cause incorrect behavior, but this is not something the program is aware of itself. An example could be a shopping cart which charges customers twice.

Runtime errors
Runtime errors occur when the program tries to execute something it is either incapable or unwilling to do. The classical example is to try and ask a program to divide a number by zero, which will often cause a crash.

For the purpose of this study, runtime errors were further classified into additional bug types based on those most commonly observed at Console.

Property access on undefined objects
One of the most common Javascript bugs comes from taking action on objects which are not what they're expected to be. An example would be trying to access a property on an object which is undefined. Another common cause of this bug is where objects are accessed multiple levels deep. As soon as one level doesn't exist, the next call will fail.

Unexpected input types as arguments
This bug is one of the most common ones in Javascript since there's no way to ensure that arguments are of a certain type. This can lead to issues where strings get concatenated instead of numbers added together, or method types get called which don't exist on the argument because it's not of the expected type. An example of the latter would be calling ".length" on a number.
**Event handling failure**

When an event that's meant to trigger fails to do so, or triggers at the wrong time. For instance clicking a button and nothing happens.

**External API failure**

An external API going down or changing contracts can cause errors from incorrect handling of these responses. Some common examples include applications showing loader graphics which never go away, or the assumption that response data has a particular structure.

**Styling / presentation errors**

These errors refer more to the visual nature of a user interface. An example of this type of failure would be a button which changes color because the CSS it was previously using got overridden by another rule.