Testing & Performance

SWE 432, Fall 2016

Design and Implementation of Software for the Web

GEORGE MASON UNIVERSITY

Today

- What's behavior driven development and why do we want it?
- Some tools for testing web apps - focus on Jasmine

For further reading:

Jasmine JavaScript Testing, Paulo Ragonha (Safari Books Online)

http://jasmine.github.io

http://reactkungfu.com/2015/07/approaches-to-testing-react-components-an-overview/

https://github.com/gmu-swe432/lecture12demos

https://gmu-swe432.github.io/lecture12demos/index.html

Unit Testing

- Unit testing is testing some program unit in isolation from the rest of the system (which may not exist yet)
- Usually the programmer is responsible for testing a unit during its implementation (even though this violates the rule about a programmer not testing own software)
- Easier to debug when a test finds a bug (compared to full-system testing)

Integration Testing

- Motivation: Units that worked in isolate may not work in combination
- Performed after all units to be integrated have passed all unit tests
- Reuse unit test cases that cross unit boundaries (that previously required stub(s) and/or driver standing in for another unit)

Unit vs Integration Tests

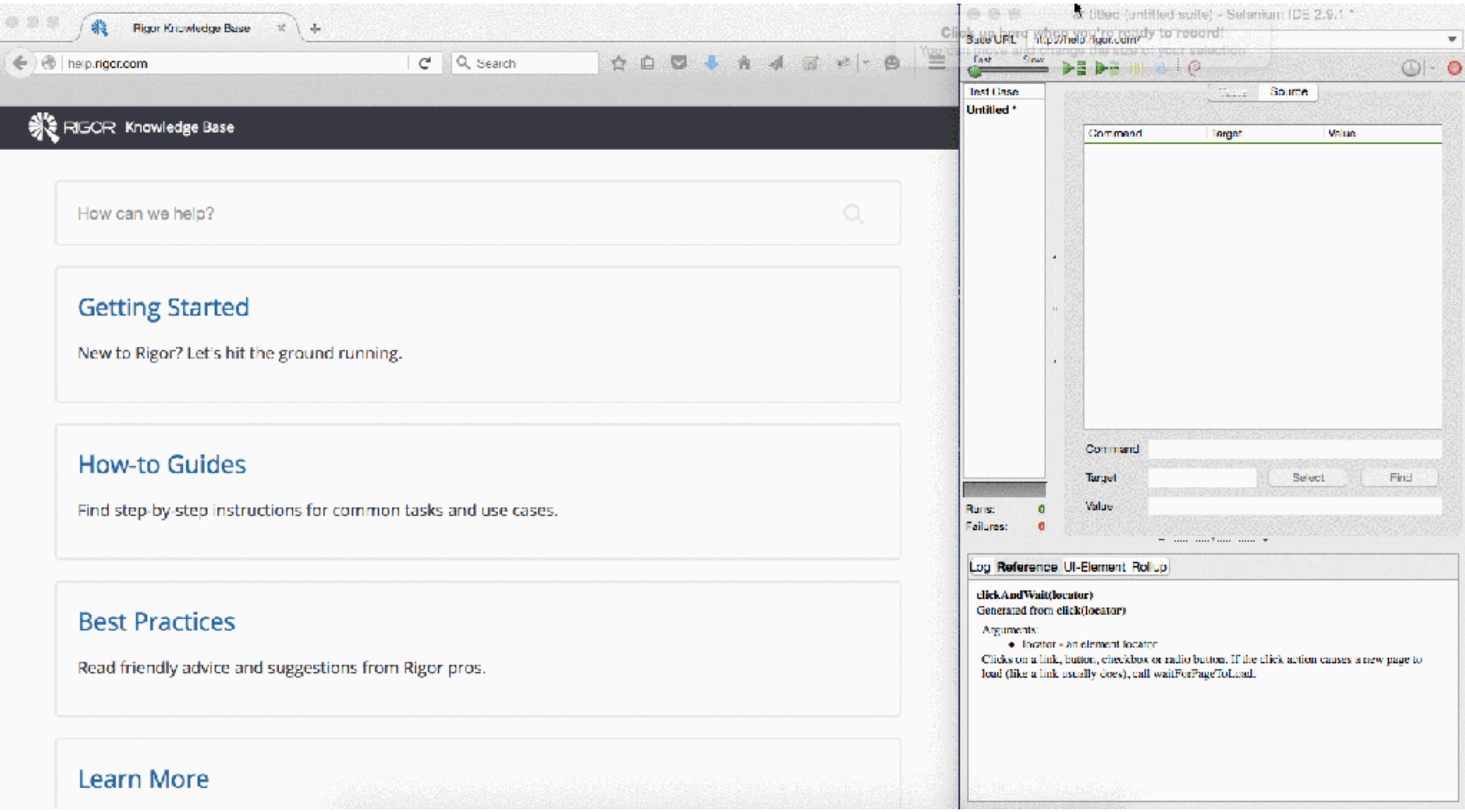
Automated Web App Testing

- Express to some script:
  - What inputs to feed into your app
  - How to feed those inputs in
  - What the result should be
  - How to identify the result
- For JS functions:
  - Easy: write some code
- For interaction with DOM/browser...
  - Trickier

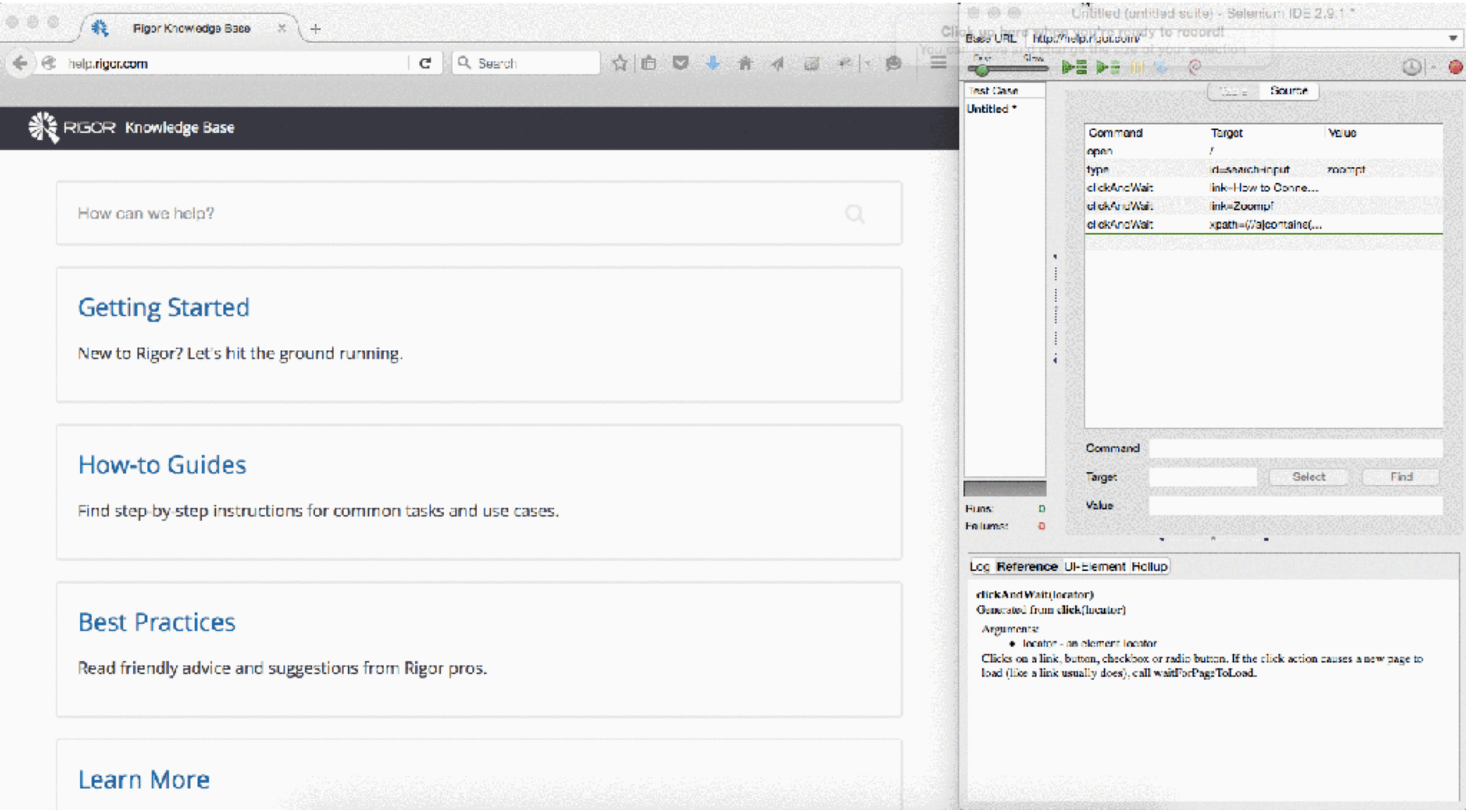
Automating Browser Interactions

- Record & Playback (e.g. Selenium)
  - Record your manual testing
  - …and it plays it back automatically, checking that the visual result is the same
- Good news:
  - Really fast to get started
  - Requires no prior experience with testing

Recording web interactions



Playing back web interactions



Record & Playback: The Dirty Side

- Very brittle:
  - Tools usually record absolute path to an element:
    - "Click the first button in the second div in the 3rd row of the first table in the body"
  - To write new tests, need to record a whole new interaction
  - Maintaining these things is tough
- End up with a lot of duplication
  - Unable to re-use setup between different tests

Writing good tests

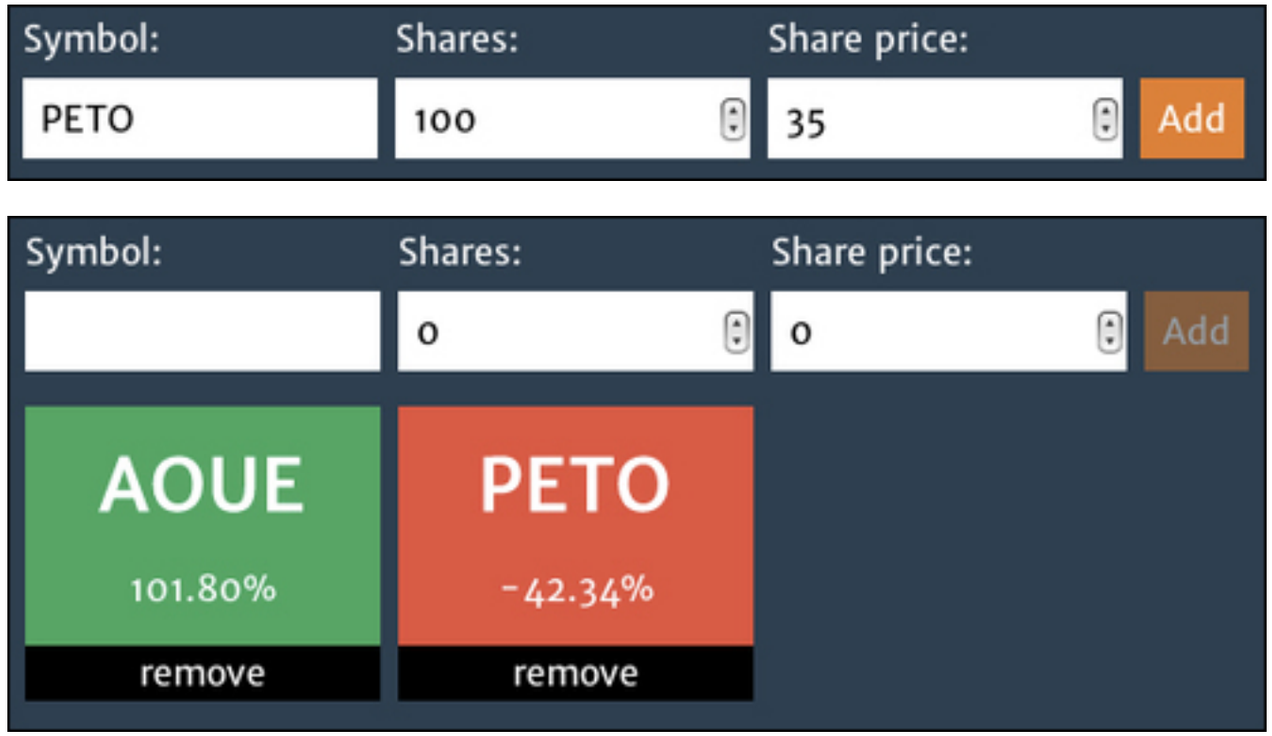
- How do we know when we have tested “enough”?
  - Did we test all of the features we created?
  - Did we test all possible values for those features?

Behavior Driven Development

- Establish *specifications* that say what an app should do
- We write our spec *before* writing the code!
- Only write code if it's to make a spec work
- Provide a mapping between those specifications, and some observable application functionality
- This way, we can have a clear map from specifications to tests

Investment Tracker

- Users make investments by entering a ticker symbol, number of shares, and the price that the user paid per share
- Once the investment is inputted, the user can see the current status of their investments
- How do we test this?



Investment Tracker

- What's an investment for our app?
  - Given an investment, it:
    - Should be of a stock
    - Should have the invested shares quantity
    - Should have the share paid price
    - Should have a current price
    - When its current price is higher than the paid price:
      - It should have a positive return of investment
      - It should be a good investment

A JavaScript Testing Framework

Jasmine is a Behavior Driven Development testing framework for JavaScript. It does not rely on browsers, DOM, or any JavaScript framework. Thus it's suited for websites, Node.js projects, or anywhere that JavaScript can run.

Documentation & guides live here: http://jasmine.github.io For a quick start guide of Jasmine 2.0, see the beginning of http://jasmine.github.io/2.0/introduction.html

Upgrading from Jasmine 1.x? Check out the 2.0 release notes for a list of what's new (including breaking interface changes). You can also read the upgrade guide.

Contributing

Please read the contributors' guide

Installation

For the Jasmine NPM module:
https://github.com/jasmine/jasmine-npm

For the Jasmine Ruby Gem:
https://github.com/jasmine/jasmine-gem

For the Jasmine Python Egg:
https://github.com/jasmine/jasmine-py

For the Jasmine headless browser gulp plugin:
https://github.com/jasmine/gulp-jasmine-browser

Jasmine lets you specify behavior in *specs*

- Specs are written in JS
- Key functions:
  - `describe`, `it`, `expect`
- `Describe` a high level scenario by providing a name for the scenario and a function that contains some test information by saying what `it` should be
- Example:

```
describe("Investment", function() {
  it("should be of a stock", function() {
    expect(investment.stock).toBe(stock);
  });
});
```

Writing Specs

- Can specify some code to run before or after checking a spec

```
describe("Investment", function() {
  var stock, investment;

  beforeEach(function() {
    stock = new Stock();
    investment = new Investment({
      stock: stock,
      shares: 100,
      sharePrice: 20
    });
  });

  it("should be of a stock", function() {
    expect(investment.stock).toBe(stock);
  });
});
```

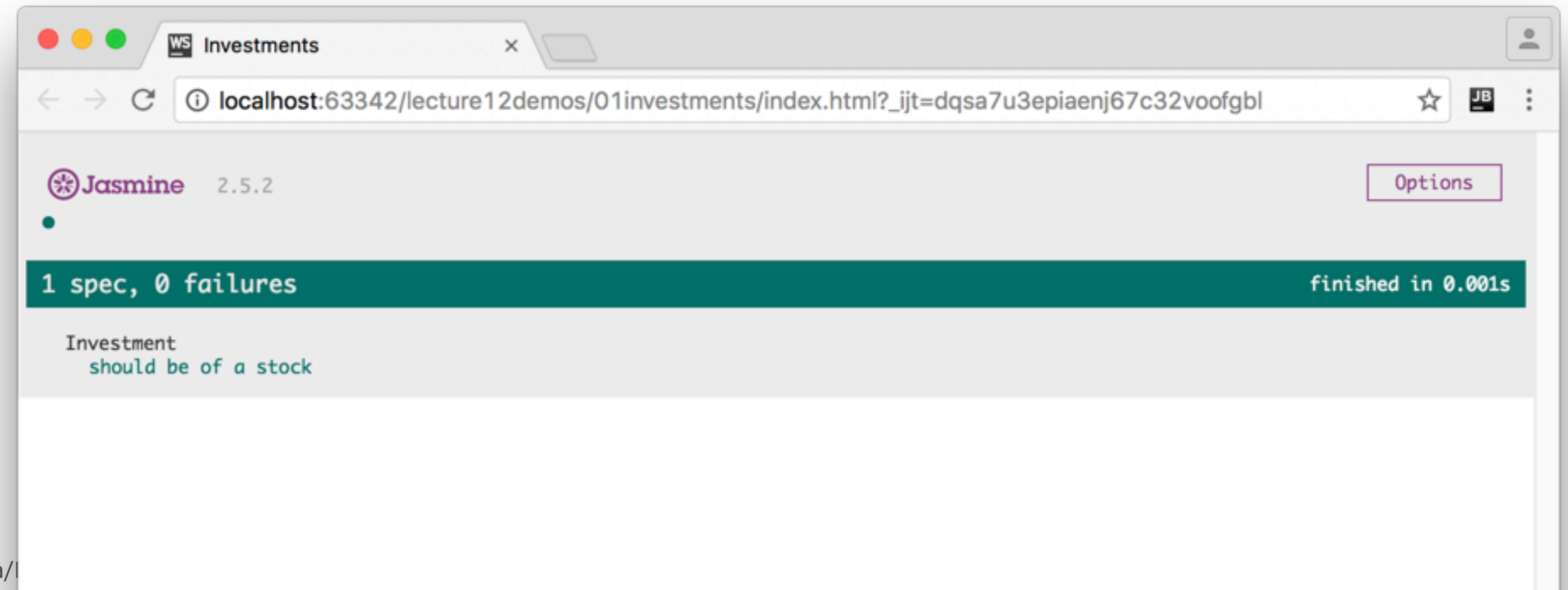
Making it work

- Download jasmine standalone and unpack it.
- Include jasmine in your HTML files

```
<link rel="stylesheet" type="text/css" href="../jasmine/lib/jasmine-2.5.2/jasmine.css">

<script type="text/javascript" src="../jasmine/lib/jasmine-2.5.2/jasmine.js"></script>
<script type="text/javascript" src="../jasmine/lib/jasmine-2.5.2/jasmine-html.js"></script>
<script type="text/javascript" src="../jasmine/lib/jasmine-2.5.2/boot.js"></script>
```

- Include your specs
- Open browser to page:



Multiple Specs

- Simply keep saying what “it” is

```
describe("Investment", function() {
  var stock, investment;

  beforeEach(function() {
    stock = new Stock();
    investment = new Investment({
      stock: stock,
      shares: 100,
      sharePrice: 20
    });

  it("should be of a stock", function() {
    expect(investment.stock).toBe(stock);
  });
  it("should have the invested shares quantity", function() {
    expect(investment.shares).toEqual(100);
  });
  it("should have the share payed price", function() {
    expect(investment.sharePrice).toEqual(20);
  });
  it("should have a cost", function() {
    expect(investment.cost).toEqual(2000);
  });
```

Nesting Specs

- "When its current price is higher than the paid price:
  - It should have a positive return of investment
  - It should be a good investment"
- How do we describe that?

```
describe("Investment", function() {
  var stock, investment;
  beforeEach(function() {
    stock = new Stock();
    investment = new Investment({
      stock: stock,
      shares: 100,
      sharePrice: 20
    });
  });
 describe("when its current price is higher than the paid price", function() {
    beforeEach(function() {
      stock.sharePrice = 40;
    });
    it("should have a positive return of investment", function() {
      expect(investment.roi()).toBeGreaterThan(0);
    });
    it("should be a good investment", function() {
      expect(investment.isGood()).toBeTruthy();
    });
  });
});
```

Matchers

- How does Jasmine determine that something is what we expect?

```
expect(investment.roi()).toBeGreaterThan(0);
expect(investment).isGood().toBeTruthy();
expect(investment.shares).toEqual(100);
expect(investment.stock).toBe(stock);
```

- These are “matcher” for Jasmine - that compare a given value to some criteria
- Basic matchers are built in:
  - toBe, toEqual, toBeTruthy, toBeNaN, toBeNull, toBeUndefined, >, <, >=, <=, !=, regular expressions
- Can also define your own matcher

toEqual vs toBe

```
describe("toEqual", function() {
  it("should pass equal numbers", function() {
    expect(1).toEqual(1);
  });

  it("should pass equal strings", function() {
    expect("testing").toEqual("testing");
  });

  it("should pass equal booleans", function() {
    expect(true).toEqual(true);
  });

  it("should pass equal objects", function() {
    expect({a: "testing"}).toEqual({a: "testing"});
  });

  it("should pass equal arrays", function() {
    expect([1, 2, 3]).toEqual([1, 2, 3]);
  });
});
```

```
describe("toBe", function() {
  it("should pass equal numbers", function() {
    expect(1).toBe(1);
  });

  it("should pass equal strings", function() {
    expect("testing").toBe("testing");
  });

  it("should pass equal booleans", function() {
    expect(true).toBe(true);
  });

  it("should pass same objects", function() {
    var object = {a: "testing"};
    expect(object).toBe(object);
  });

  it("should pass same arrays", function() {
    var array = [1, 2, 3];
    expect(array).toBe(array);
  });

  it("should not pass equal objects", function() {
    expect({a: "testing"}).not.toBe({a:
"testing"});
  });

  it("should not pass equal arrays", function() {
    expect([1, 2, 3]).not.toBe([1, 2, 3]);
  });
});
```

Truthiness

```
describe("toBeTruthy", function() {
  it("should pass the true boolean value", function() {
    expect(true).toBeTruthy();
  });

  it("should pass any number different than 0", function() {
    expect(1).toBeTruthy();
  });
  it("should pass any non empty string", function() {
    expect("a").toBeTruthy();
  });

  it("should pass any object (including an array)", function() {
    expect([]).toBeTruthy();
    expect({}).toBeTruthy();
  });
});
```

Custom Matchers

- We can define a matcher however we want: return true if the value is OK, false if not

```
describe("Investment", function() {
  beforeEach(function() {
    this.addMatchers({
      toBeAGoodInvestment: function() {
        return investment.isGood();
      }
    });
...
      it("should be a bad investment", function() {
        expect(investment).toBeAGoodInvestment();
      });
...
  });
```

Testing Asynchronous Code

- When we need to get some data asynchronously then use it, we structure it so that we get our data in a beforeEach
- And change our beforeEach to take a parameter: done. Then when we are done, call done()

```
beforeEach(function(done){
    //do something async and on its completion call done()
}
```

- No “it” statements will run until done() is called (default timeout: 5 seconds)

Testing Asynchronous Code

- Example: Assume our stock object from the investment example has a “fetch” function to update its price using AJAX
- Test that we can fetch the price, and then see the new price

```
describe("Stock", function(){
  describe("should be able to update its share price", function () {
    var fetched = false;
    beforeEach(function(done){
        stock.fetch({
            success: function () {
                fetched = true;
                done();
            }
        });
    });
    it("will get the updated price eventually", function(){
        expect(stock.sharePrice).toEqual(23.67);
    });
  });
});
```

Spies

- Sometimes, when you are testing, you don't want to deal with external components
- For instance: in the investment app - maybe don't care about HOW stock.fetch() gets the stock price - just care about it updating its state
- Solution: Mocks (Jasmine: spies)
- Spies *replace* existing methods on objects
- Spies track the parameters sent to those methods

Spies

- Can also say that a spy should return a specific value
- Or say that it should instead call a specific function
- …or so that it can also let the original function be called
- Really, really powerful

Spies - Example

- Make a spy to remove the async fetch from our investment:

```
describe("should be able to update its share price", function () {
    var fetched = false;
    beforeEach(function(done){
        spyOn(stock,"fetch").and.callFake(function(param)
        {
            this.sharePrice = 23.67;
            done();
        });
        stock.fetch({
            success: function () {
                fetched = true;
                done();
            }
        });

    });
    it("will get the updated price eventually", function(){
        expect(stock.sharePrice).toEqual(23.67);
    });
});
```

Testing Frontend Code

- How do we test our interface?
- We can describe them
- It's a lot easier with components
- We'll cover how you can test React components with Jasmine
  - Docs: https://facebook.github.io/react/docs/test-utils.html
  - Make sure to include react-with-addons (and not just react) in your pages

Testing React Components

- High level:
  - Render a component (but don't put it into the page)
  - Expect certain things about that component
- Example
- TodoApp
  - Has a new button
  - Has a TodoList component
  - The new item button:
    - Causes fireBase push to be called
  - The TodoList:
    - Updates firebase when text is changed
    - Removes items from firebase when delete is clicked

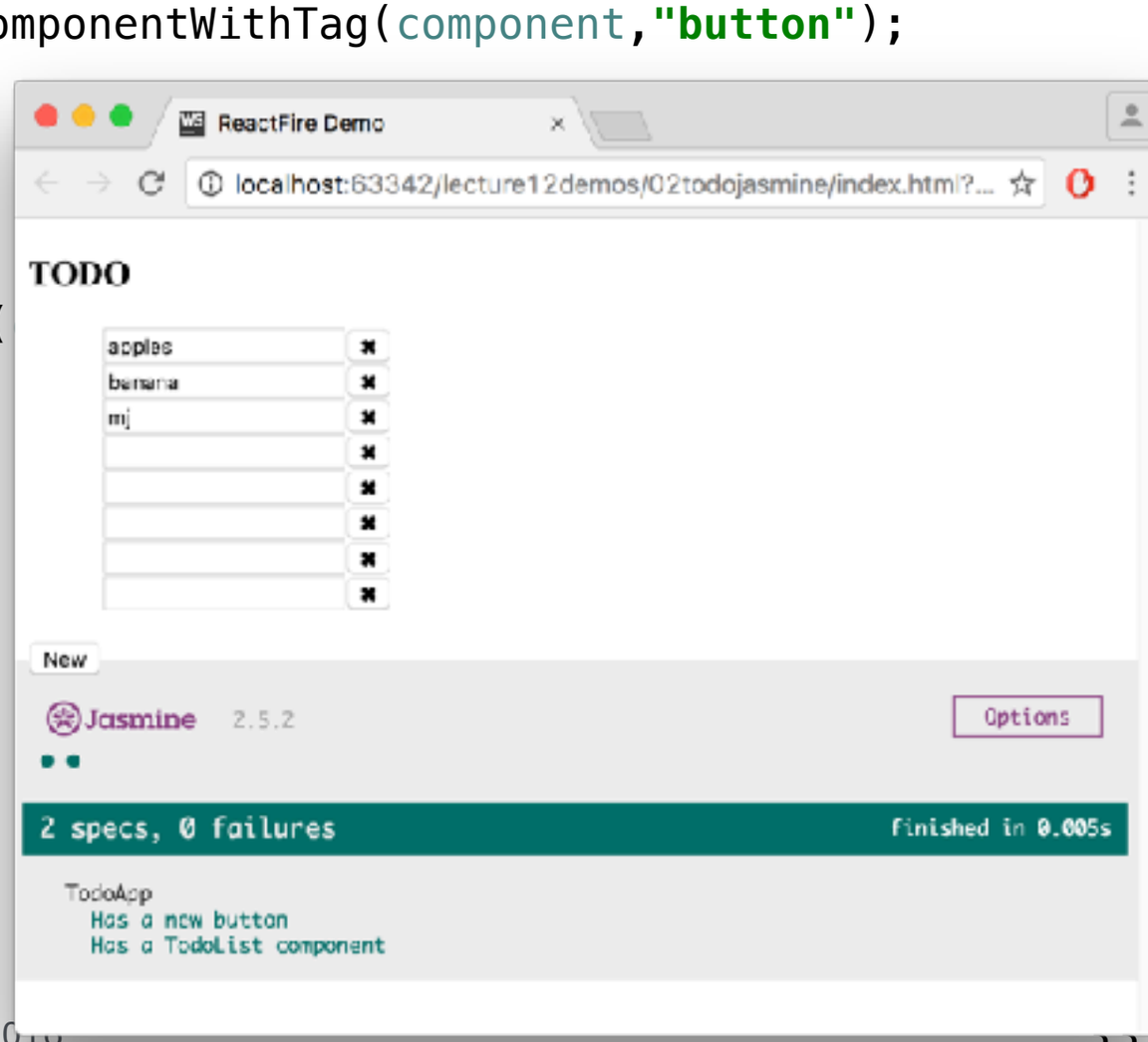
React TestUtilities

- **renderIntoDocument**
  - Renders a component into the DOM but does not attach it to the page
- **Simulate**
  - Simulates an event
- findAllInRenderedTree
  - Finds any components that match a function provided
- scryRenderedDOMComponents
  - Return all DOM components rendered by CSS class
- findRenderedDOMComponentWithClass
  - Return the only DOM component rendered w/ the given CSS class, errors if more than 1 or less than 1.
- scryRenderedDOMComponentsWithTag
  - Return all DOM components rendered with a given type
- findRenderedDOMComponentWithTag
  - Return single DOM component rendered with a given tag, error if < or > 1
- findRenderedComponentWithType
  - Return single React component rendered with a given type, error if < or > 1
- Plus a whole lot more: https://facebook.github.io/react/docs/test-utils.html

Testing React Components

- Todo:

```
describe('TodoApp', function() {
    var TestUtils = React.addons.TestUtils;
    var component, element, renderedDOM;
    beforeEach(function(){
        element = React.createElement(TodoApp);
        component = TestUtils.renderIntoDocument(element);
    });
    it("Has a new button", function(){
        let button = TestUtils.findRenderedDOMComponentWithTag(component,"button");
         expect(button).not.toBeUndefined();
         expect(button.innerHTML).toBe("New");
    });
    it("Has a TodoList component", function(){
       expect(function(){
        TestUtils.findRenderedComponentWithType(
       }).not.toThrow();
    });
});
```



How do we test the new button?

Testing Events

```
TestUtils.Simulate.eventType(eventTarget, params)
```

- Test the new button:

```
it(“Can click on new button", function(){
    let button = TestUtils.findRenderedDOMComponentWithTag(todoAppComponent,"button");
    TestUtils.Simulate.click(button);
});
```

- Problem: We trust that Firebase works. Just need to make sure button works. But this code will actually create a new item….
- Solution: spies!

```
describe("New item button", function(){
   beforeEach(function(){
       spyOn(todoAppComponent.fireRef,"push");
   });
    it("Causes fireBase push to be called", function(){
        let button = TestUtils.findRenderedDOMComponentWithTag(todoAppComponent,"button");
        TestUtils.Simulate.click(button);
        expect(todoAppComponent.fireRef.push).toHaveBeenCalledWith({"text": ""});
    });
});
```

Big huge Todo

Jasmine Example

https://gmu-swe432.github.io/lecture12demos/02todojasmine/

Performance Best Practices

- CDNs
  - Server might be closer to client than yours, clients can cache
- Minification
  - Reduce size of JS being transferred
  - Pre-fetch data
  - Profile using Chrome Developer Tools
  - PageSpeed Insights:
    - https://developers.google.com/speed/pagespeed/insights/