Test-Driven Development Using React.js and ES2018

version 3.2, May 2019

Contents

Contents

Introduction

Objectives

- Who am I?
- Who are you?
- Daily Schedule
- Course Schedule and Syllabus

About Me

Introductions

- What's your name?
- What do you do?
- JavaScript level (beginner, intermediate, advanced)?
- What do you want to know at the end of this course?
- What do you like to do for fun?
- Favorite food?

Daily Schedule

- 08:30 - 10:30
- 15 minute break
- 10:45 - 12:00
- 1 hour lunch break
- 1:00 - 2:00
- 15 minute break
- 2:15 - 3:15
- 15 minute break
- 3:30 - 4:30

The Big Picture

- Topics Covered in this Course
  - Professional Front-End Development
  - Modern JavaScript
  - React.js

Lab 0: Course Homepage

- https://watzthisco.github.io/tdd-react-labs-v3.x/
- This is the homepage for the course, which contains links to further reading on topics covered in the course.
- Please bookmark and check it out as you have time
- If you find an article that should be there, post it to the course issues
  - https://github.com/watzthisco/tdd-react-labs-v3.x/issues

Professional Front-End Development

- Four best practices
  - Version control
    - “Be safe"
  - Automation
    - “Be lazy"
  - Reproducible build
    - “Be verifiable"
  - Test-Driven Development
    - “Be flexible"

What's New in JavaScript (aka ECMAScript)?

- ES2015 (aka ES6) introduced many new features, including:
  - Arrow functions
  - Classes
  - Block-scoped binding constructs (let and const)
  - Iterators
  - Modules
  - Promises
- Since 2015, a new version is released annually
- React uses the new syntax (classes, modules, etc)

What is React.js?

- A Library for Building User Interfaces
- Created by Facebook
  - Open sourced in 2014
- Abstracts the Document Object Model (Virtual DOM)
- Implements one-way data flow
- Component-based
- Can be rendered on the server
- Plays well with other libraries / frameworks

What is React NOT?

- React is not a framework.
- React doesn't have AJAX capabilities.
- React has no data layer.

When can you use React?

- Complex single-page applications (SPAs) can be built entirely using React.
- React can be used as a substitution for views in a traditional MV* framework.
- React can generate static HTML on the server.
- React can be used to create native mobile apps.
  - React Native

Who Uses React?

- AddThis
- Angie's List
- Airbnb
- Atlassian
- BBC
- Codecademy
- Coursera
- Dropbox
- Expedia
- Facebook
- IMDb
- Imgur
- Instagram
- Intuit
- Liberty Mutual
- Lyft
- Netflix
- NFL
- OkCupid
- Paypal
- Reddit
- Salesforce
- Squarespace
- Tesla Motors
- The New York Times
- Trulia
- Trunk Club
- Twitter
- Uber
- Visa
- WhatsApp
- Wired
- Wix
- Wolfram Alpha
- Wells Fargo
- WordPress
- Yahoo
- Zendesk

and many more!

React QuickStart

Objectives

- Install CreateReactApp
- Create a React App
- Test and Run a React App with CRA

What is CreateReactApp

- Creates React App with No Build Configuration
- Simplifies setup for single-page apps and learning
- Sacrifices flexibility for simplicity
- Not a substitute for understanding the tools, but great when you just want to write React code quickly
- Other options offer more flexibility, less simplicity
  - nwb
  - Neutrino

Getting Started with CreateReactApp

1. Create Your App
   - `npx create-react-app my-react-app`
2. Start It
   - `cd my-react-app`
   - `npm start`
   - `Go to http://localhost:3000 in browser`

Lab – Exploring CreateReactApp

- Before we get into how to configure the necessary tools for React manually and before you learn how React works, try to complete the following challenges.
  1. Change the default ‘Welcome to React’ message.
  2. Create a new file named Footer.js in the src directory
  3. Type the code below into Footer.js

```
import React from 'react';

function Footer(){
    return (<p>this is the footer.</p>)
}

export default Footer;
```

  4. Add the following to App.js (under the other import statements)

```
import Footer from './Footer.js';
```

  5. Add the following inside the <div> in App.js

```
<Footer />
```

Lab – Exploring React

- Using what you learned from creating Footer.js, make Header.js and Content.js to replace code in App.js.
- Your finished `return` statement in App.js should match this:

```
return (
  <div className="App">
    <Header />
    <Content />
    <Footer />
  </div>
);
```

Lab – Testing React

- Make copies of App.test.js for Footer.js, Header.js, and Content.js
- Modify the contents of the new files to test the new components.
- Run your tests by entering the following in the command line

```
npm test
```

Chapter 1:
Development Ecosystem

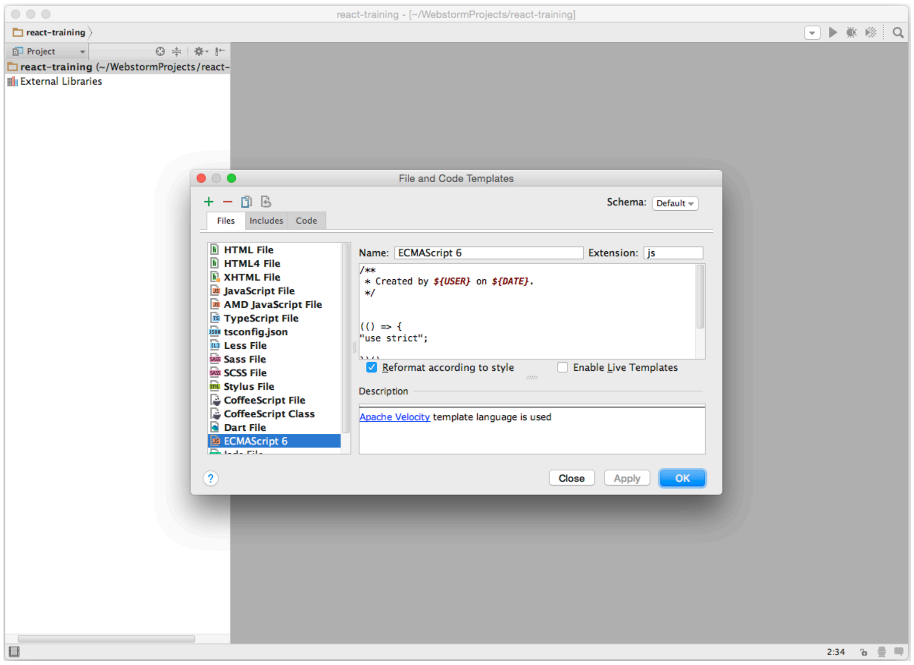
Objectives:

- Configure your IDE
- Use Node.js as a development tool
- Use NPM to install and manage packages
- Use Git for version control
- Use command line dev tools

Code Editors and IDEs

- MS Visual Studio Code
- JetBrains WebStorm
- Atom (atom.io)
- Adobe Brackets
- Eclipse
- Emacs
- IntelliJ IDEA
- Netbeans
- Sublime Text
- vi/Vim

Lab 1: Installing WebStorm or Visual Studio Code



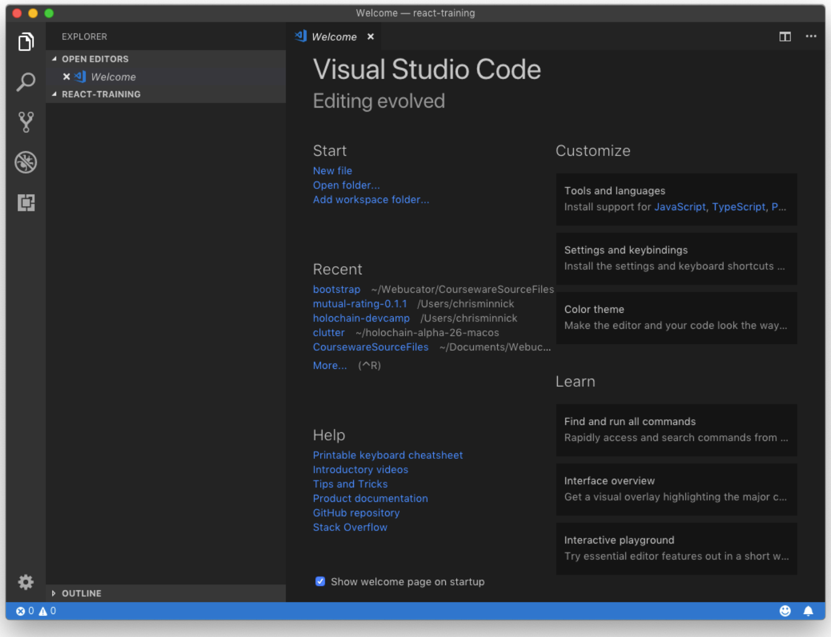
Name: ECMAScript 6 Extension: js

```
/**
 * Created by ${USER} on ${DATE}.
 */


(() => {
"use strict";
```

☑ Reformat according to style ☐ Enable Live Templates

Description

Apache Velocity template language is used



Visual Studio Code
Editing evolved

Start
New file
Open folder...
Add workspace folder...

Recent
bootstrap ~/Webucator/CoursewareSourceFiles
mutual-rating-0.1.1 /Users/chrisminnick
holochain-devcamp /Users/chrisminnick
clutter ~/holochain-alpha-26-macos
CoursewareSourceFiles ~/Documents/Webuc...
More... (^R)

Help
Printable keyboard cheatsheet
Introductory videos
Tips and Tricks
Product documentation
GitHub repository
Stack Overflow

Customize
Tools and languages
Install support for JavaScript, TypeScript, P...
Settings and keybindings
Install the settings and keyboard shortcuts ...
Color theme
Make the editor and your code look the way...

Learn
Find and run all commands
Rapidly access and search commands from ...
Interface overview
Get a visual overlay highlighting the major c...
Interactive playground
Try essential editor features out in a short w...

☑ Show welcome page on startup

Node.js

Objectives

- Install Node
- Test node and get started using npm

What is Node.js?

- JavaScript runtime
- Built on Google Chrome's V8 JavaScript engine
- Open-source
- Event-driven
- Non-blocking
  - Never waits
- Single-threaded
  - Can handle thousands of concurrent connections with minimal overhead
- No buffering
- Used on the server as well as for development automation

How is Node.js Different?

- JavaScript is single-threaded
- A Node application has one event loop, and gives you the benefit of multithreading with async operations
  - Can handle thousands of concurrent connections with minimal overhead
- No buffering
  - Node applications never buffer data.
  - Data is output in chunks.
- Non-blocking
  - Node doesn't wait for data to be returned from APIs.
  - While it's possible to write blocking code in Node.js, it's discouraged.

A Simple Node.js Example

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

EventEmitter

- Node is event driven.
- EventEmitter is a class that lets you listen for events and assign actions when those events occur.
- Very loose way of coupling different parts of your application together.

```
var EventEmitter = require("events").EventEmitter;

var ee = new EventEmitter();
ee.on("someEvent", function () {
    console.log("event has occured");
});

ee.emit("someEvent");
```

Modules Overview

- Node uses CommonJS for loading modules
- Original name was ServerJS
- Provides a way to import dependencies in JavaScript when used outside the browser (such as server side or in desktop applications)
- `module.exports`
  - Encapsulates code into a module
- `require`
  - Imports a module into JavaScript code

CommonJS Example

foo.js

```
var circle = require('./circle.js');
console.log('Area of circle: ' + circle.area(4));
```

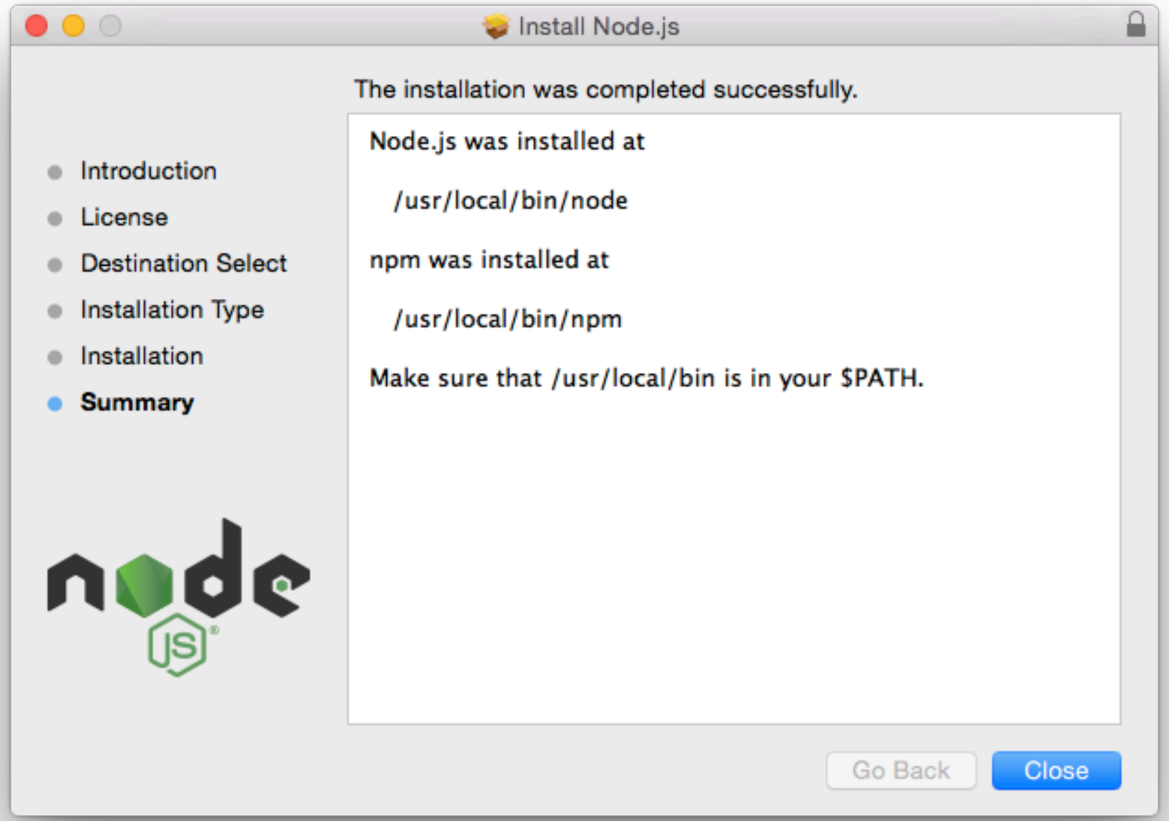
circle.js

```
exports.area = function(r){
  return Math.PI * r * r;
}
```

Front-end Node

- Node has become an essential part of front end development.
- It's used by:
  - Package installers
  - Task runners
  - Testing frameworks
  - CSS compilers
  - JS Transpilers
  - Web servers
  - Web browsers (PhantomJS)
  - and more!

Lab 2 - Getting Started with Node.js



The installation was completed successfully.

Node.js was installed at

/usr/local/bin/node

npm was installed at

/usr/local/bin/npm

Make sure that /usr/local/bin is in your $PATH.

Git

Objectives

- Learn about how Git works
- Learn a typical Git workflow
- Install Git
- Use the most common Git commands

What is Version Control?

- Essential component of any development workflow
- Records changes to files over time
- Allows recalling of specific versions
- Makes collaboration possible
- Makes software development safer

History of Git

- From 2002 - 2005, the Linux Kernel used the proprietary BitKeeper VCS.
- In 2005, after a falling out with BitKeeper's developer, they decided to create their own VCS.
- Goals
  - Speed
  - Simple design
  - Strong support for non-linear development
  - Fully distributed
  - Able to handle large projects

What is Git?

- Version Control System (VCS)
- Different from SVN, CVS, etc
  - Other VCS: store list of changes
  - Git: store a snapshot of files at time of commit
- Doesn't restore unchanged files
- Nearly every operation is local
- Generally only adds data
  - It's difficult to do something that can't be un-done.

3 States of Git

- Git has three states that your files can reside in:
  - Modified
    - Data changed but not committed
  - Staged
    - File is marked to go into your next snapshot
  - Committed
    - Data stored in your local repo

Git Workflow

1. Modify files
2. Stage the files
   - Adds snapshots to the staging area
   - Git add
   - Git add is dual-purpose: it tells git to track new files and it stages changes to existing files.
3. Commit
   - Stores a snapshot permanently in your Git directory
   - Git commit

- You can also skip the staging area by using the -a option with git commit. This will automatically stage all tracked files before doing the commit.

Lab 3 - Version Control With Git

Command Prompt

Objectives

- Use a Unix-style command prompt
- Get familiar with basic commands

Know Your Shell

- Bash shell
- Terminal (Mac)
- Git bash (Windows)

You can use the windows command prompt, but it has its own non-standard commands, so using a bash emulator is recommended.

- `cd` = change directory
- `./` = current directory
- `../` = up one directory
- `ls` = list files
- `ls -la` = long list format and don't hide files starting with .
- `pwd` = print working directory
- `mkdir` = create a new directory
- `cp [source] [dest]` = copy
- `mv [source] [dest]` = move, or rename
- `rm` = remove files or directories
- `help` = get bash commands
- `-help` = get help with a command

Chapter 2: Reproducible Builds

Objectives

- Understand the role of build tools
- Learn about build automation
- Configure and use npm

Why Automate Your Build?

- First step in creating culture of Continuous Delivery and DevOps
  - Continuous Delivery
    - Teams produce software in short cycles, ensuring that software can be reliably released at any time.
  - DevOps
    - Emphasizes communication and collaboration of developers and IT and creates culture of rapid and reliable releases.

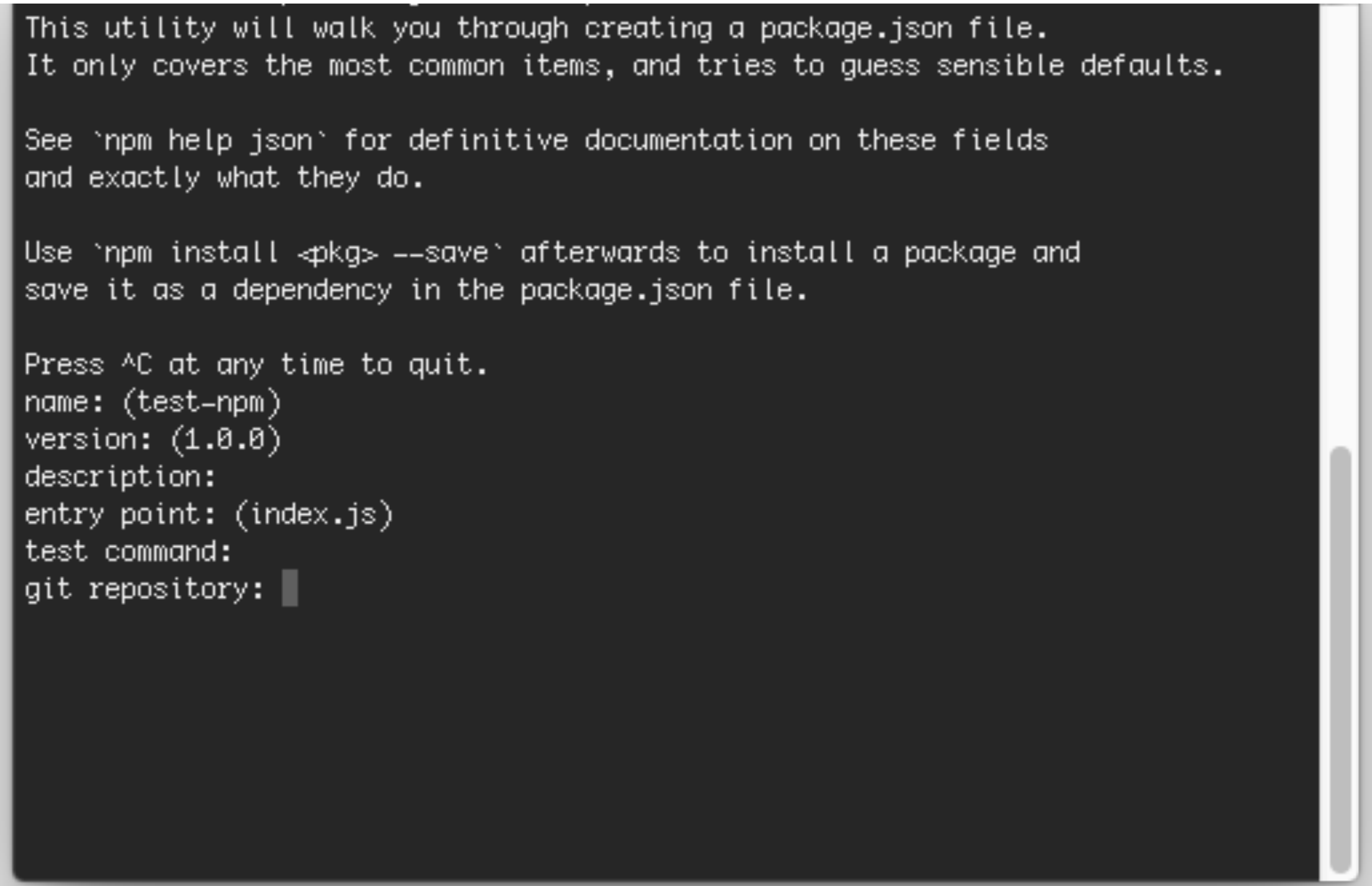
Task Runners

- Grunt
  - emphasizes configuration over code
- Gulp
  - emphasizes code over configuration
- npm as build tool
  - write shell scripts to automate tasks

npm

- Installs, publishes, and manages node programs
- Is bundled and installed with Node
- Allows you to install Node.js applications from the npm registry
- Written in JavaScript

Lab 4 – Initialize npm



```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (test-npm)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
```

node_modules

- Two ways to install npm packages
  - Locally
  - Globally
- When you install packages locally, they're put into the **node_modules** directory in your current directory.
- You should always run npm install from the same directory as your **package.json** file, which should be at the root of your project.
- Run `npm update` to update local packages
- Run `npm outdated` to find out which packages are outdated.

package.json

- Manages locally installed npm packages
- Documents packages your project depends on
- Specifies the versions of each package your project can use
- Makes your build reproducible
- Package versions are specified using Semantic versioning (semver)
- Semver ranges:
  - ~ : patch release - 1.0.x
  - ^ : minor release - 1.x
  - * : major release - x

Npm Install

- Is used to download and install a package
- `-g`: install globally
- `--save`: Package will appear in your dependencies
- `--save-dev`: Package will appear in your devDependencies
- `--save-optional`: Package will appear in your optionalDependencies
- `--save-exact`: Saved dependency will be configured with an exact version rather than the default range operator.

Lab 5 – Using npm as a Build Tool

Lab 6 - Managing External Dependencies

1. Create a script called version-check to check the node version
2. Compare node version against version listed in package.json
3. Fail with helpful message if wrong version is installed
4. Make the default task dependent on version

Chapter 3: Static Code Analysis

Objectives

- Learn about Lint tools
- Use ESLint
- Configure ESLint
- Manual testing with a local web server

Lint tools

- JSLint
  - Created by Douglas Crockford
  - Highly opinionated (like Mr. Crockford)
  - Flags style that conflicts with "The Good Parts" according to D.C.
- JSHint
  - More control
  - Doesn't flag style issues by default
- ESLint
  - Allows developers to create their own rules ("Pluggable")
  - "Agenda free" - doesn't promote any particular style

Configuring ESLint

- Two ways
  - Configuration comments
    - embed configuration info in JS files with comments
    - `/* eslint eqeqeq: "off", curly: "error" */`
  - Configuration files
    - .eslintrc file

ESLint: What Can Be Configured?

- Environments
  - Where is the code running?
  - Includes predefined global variables for each environment.
- Globals
  - Specify additional globals your scripts use.
- Rules
  - Enable rules at different levels.

ESLint Rules

- 3 Levels
  - "off" or 0
    - Rule not applied.
  - "warn" or 1
    - Warn but don't exit.
  - "error" or 2
    - Error and exit.
- Example rules

```
{
    "rules": {
    "eqeqeq": "off",
    "curly": "error",
    "quotes": ["error", "double"]
}
}
```

Lab 7 - Automate Linting

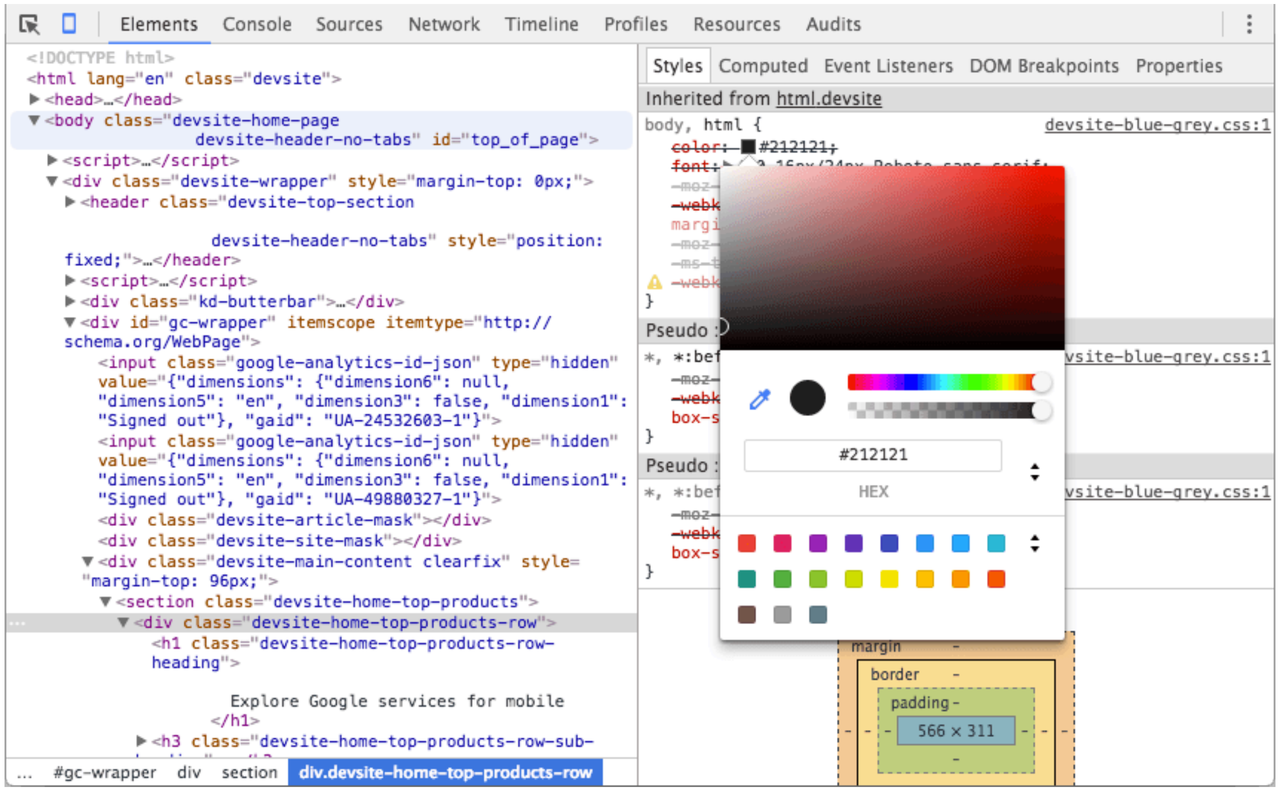
- Install ESLint into project
- Test it
- Create an npm script called "lint"
- Make lint task a dependency of default build task

Lab 8 - Configure a Local Web Server

- Install http-server
- Add webserver task to package.json

Browser Development Tools

- Using Chrome Developer Tools
- Using IE F12 Developer Tools



```
<!DOCTYPE html>
<html lang="en" class="devsite">
▶<head>…</head>
▼<body class="devsite-home-page
              devsite-header-no-tabs" id="top_of_page">
  ▶<script>…</script>
  ▼<div class="devsite-wrapper" style="margin-top: 0px;">
    ▶<header class="devsite-top-section

                devsite-header-no-tabs" style="position:
    fixed;">…</header>
    ▶<script>…</script>
    ▶<div class="kd-butterbar">…</div>
    ▼<div id="gc-wrapper" itemscope itemtype="http://
    schema.org/WebPage">
        <input class="google-analytics-id-json" type="hidden"
        value="{"dimensions": {"dimension6": null,
        "dimension5": "en", "dimension3": false, "dimension1":
        "Signed out"}, "gaid": "UA-24532603-1"}">
        <input class="google-analytics-id-json" type="hidden"
        value="{"dimensions": {"dimension6": null,
        "dimension5": "en", "dimension3": false, "dimension1":
        "Signed out"}, "gaid": "UA-49880327-1"}">
        <div class="devsite-article-mask"></div>
        <div class="devsite-site-mask"></div>
      ▼<div class="devsite-main-content clearfix" style=
      "margin-top: 96px;">
        ▼<section class="devsite-home-top-products">
          ▼<div class="devsite-home-top-products-row">
              <h1 class="devsite-home-top-products-row-
              heading">

                      Explore Google services for mobile
                    </h1>
            ▶<h3 class="devsite-home-top-products-row-sub-
```

```
Inherited from html.devsite
body, html {                          devsite-blue-grey.css:1
   ~~color: #212121;~~
   ~~font:~~
}
Pseudo
*, *:bef                              vsite-blue-grey.css:1
   box-s
}
Pseudo
*, *:bef                              vsite-blue-grey.css:1
   box-s
}
```

#212121

Chapter 4: Test-Driven Development

Objectives

- Learn the TDD Steps
- Write Assertions
- Understand exception handling in JS
- Create tests with Jasmine
- Automate cross-browser testing

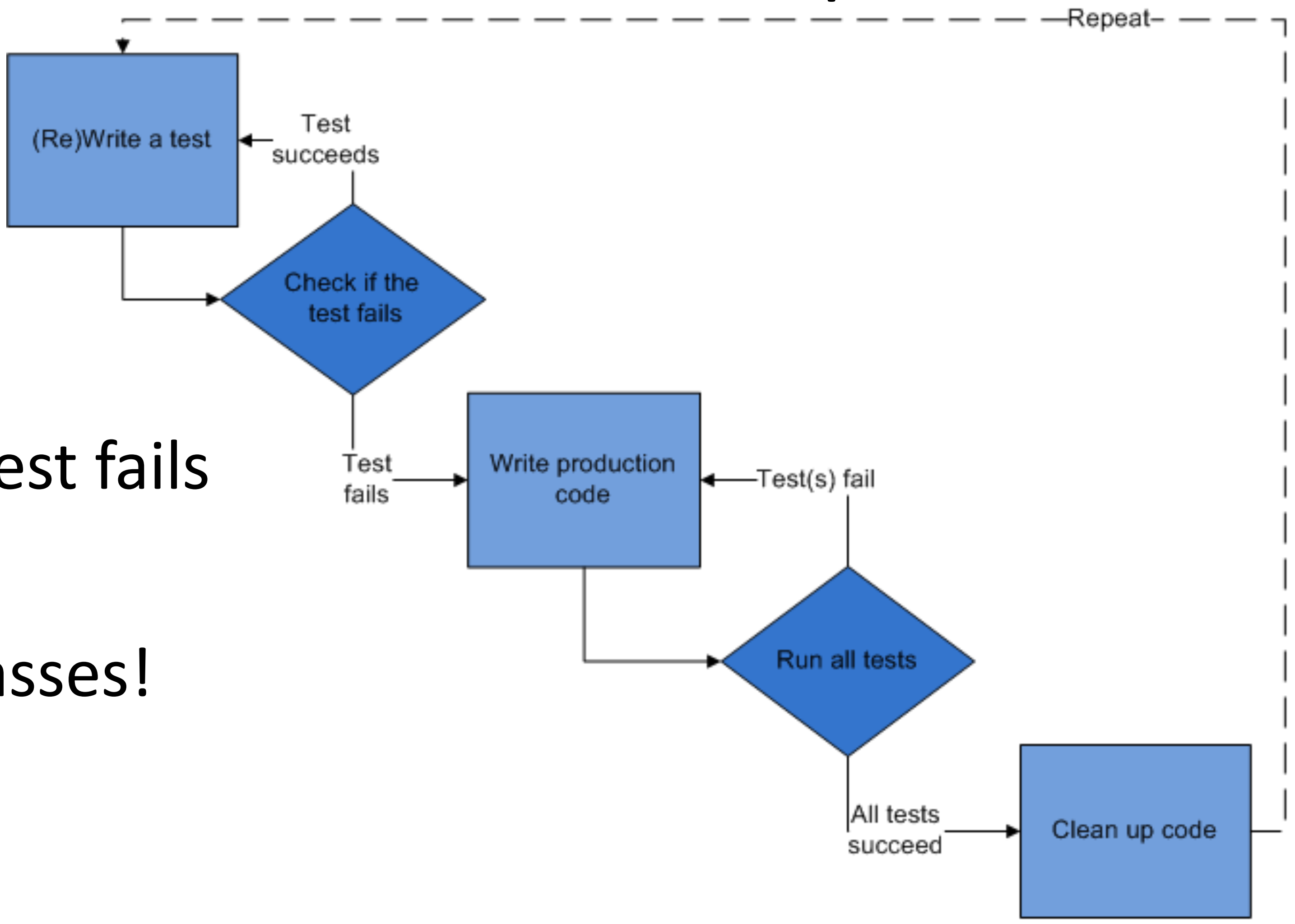
Goal of TDD

- Clean code that works.

The TDD Cycle

- Red
  - write a little test that doesn't work.
- Green
  - make the test work, as quickly as possible.
  - don't worry about doing it right.
- Refactor
  - eliminate duplication created in making the test work.

TDD Steps



- Write a test
- Check that test fails
- Write code
- Run test - passes!
- Refactor
- Repeat

Red

- Write the story.
- Invent the interface you wish you had.
- Characteristics of a good tests:
  - Each test should be independent of the others.
  - Any behavior should be specified in only one test.
  - No unnecessary assertions
  - Test only one code unit at a time
  - Avoid unnecessary preconditions

Green

- Get the test to pass as quickly as possible.
- Three strategies:
  - Fake it
    - Do something, no matter how bad, to get the test to pass.
  - Use an obvious clean solution.
    - But don't try too hard!
  - Triangulation
    - only generalize code when you have two examples or more.
    - When the 2nd example demands a more general solution, then and only then do you generalize.

Refactor

- Make it right.
- Remove duplication.
- Improve the test.
- Repeat.
- Add ideas or things that aren't immediately needed to a todo list.

Assertions

- Expression that encapsulates testable logic
- Assertion Libraries
  - Chai, should.js, expect.js, better.assert
- Examples
  - `expect(buttonText).toEqual('Go!'); // jasmine`
  - `result.body.should.be.a('array'); // chai`
  - `equal($("h1").text(), "hello"); //QUnit`
  - `assert.deepEqual(obj1, obj2); //Assert`

JavaScript Testing Frameworks

- Jasmine
- Mocha
  - doesn't include its own assertion library
- QUnit
  - from JQuery
- js-test-driver
- YUI Test
- Sinon.JS
- Jest

JS Exception Handling

```
function hello(name) {
   return "Hello, " + name;
}
let result = hello("World");
let expected = "Hello, World!";
try {
   if (result !== expected) throw new Error
      ("Expected " + expected + " but got " +
      result);
} catch (err) {
   console.log(err);
}
```

Jasmine Overview

Objectives

- Write test suites
- Create specs
- Set expectations
- Use matchers

How Jasmine Works

- Suites describe your tests
- Specs contain expectations

```
describe("A suite is just a function", function() {
  var a;

  it("and so is a spec", function() {
    a = true;

    expect(a).toBe(true);
  });
});
```

Test Suites

- Created using the describe function
- Contain one or more specs
- 2 params
  - Text description
  - Function

```
describe("Hello", function() {
...
}
```

Specs

- Created using the `it` or `test` function
  - they're the same thing
- Contains one or more expectations
- expectations === assertions

```
describe("Hello", function() {

   it("Concats Hello and a name", function() {
      var expected = "Hello, World!";
      var actual = hello("World");
      expect(actual).toEqual(expected);
   });
});
```

Expectations

- AKA assertions
- Made using `expect` function,
  - Takes a value
- Chained to a Matcher
  - Takes the expected value

```
expect(actual).toEqual(expected);
```

Matchers

- `expect(fn).toThrow(e);`
- `expect(instance).toBe(instance);`
- `expect(mixed).toBeDefined();`
- `expect(mixed).toBeFalsy();`
- `expect(number).toBeGreaterThan(number);`
- `expect(number).toBeLessThan(number);`
- `expect(mixed).toBeNull();`
- `expect(mixed).toBeTruthy();`
- `expect(mixed).toBeUndefined();`
- `expect(array).toContain(member);`
- `expect(string).toContain(substring);`
- `expect(mixed).toEqual(mixed);`
- `expect(mixed).toMatch(pattern);`

TDD vs. BDD

Test-Driven Development

- Focused on being useful for programmers

Behavior-Driven Development

- Focused on documentation for non-programmers
  - Features, not results
  - More verbose

TDD Example

```
suite('Counter', function() {
  test('tick increases count to 1',

   function() {
       var counter = new Counter();
       counter.tick();
       assert.equal(counter.count, 1);
     });

});
```

BDD Example

```
describe('Counter', function() {
  it('should increase count by 1 after calling
tick',
   function() {
       var counter = new Counter();
       var expectedCount = counter.count + 1;
       counter.tick();
       assert.equal(counter.count, expectedCount);
     });
});
```

Lab 9 – Get Started with Jasmine

- Install jasmine
- Jasmine init
- Create a test suite

Lab 10 - TDD Practice

- We need more features!!!
- Split into teams and work together to implement the following features using TDD. Break up each feature into smaller units as needed.
  - It gives an appropriate hello for the time of day
    - Good morning!
    - Good afternoon!
    - Good evening!
  - It displays a login message if no name is provided
  - It speaks German to Germans
  - It refuses to say hello after the 4th time the function is called.

Automated Cross-browser Testing

- Front-end code runs in browsers
- Browsers are all slightly different
- Automated testing in Node isn't enough
- Manual cross-browser testing is tedious
- Karma can run tests in actual browsers

Karma

- A test runner for JavaScript
- Test on real devices
- Testing Framework Agnostic
- How it works
  - Spawns a web server
  - Executes source code against test code for each browser connected
  - Results are displayed on the command line
- Preprocessors can be configured to do work to files before they get served to browsers.
- Can launch browsers automatically, or you can do it manually

Lab 11 - In-browser Testing with Karma

- Install karma
- Run karma
- Integrate karma into your build
- Testing with multiple browsers / devices

Chapter 5: Modularity

Objectives

- Explain modularity
- Learn different methods of using modules in JS
- Understand methods of front-end module management

Why is Modularity Important?

- Individual modules can be tested
- Allows distributed development
- Enables code reuse
- Reduce coupling
- Increase cohesion

CommonJS

- Modularity for JavaScript outside of the browser
- Node.js is a CommonJS module implementation
- uses `require` to include modules

- export an anonymous function

hello.js
```
    module.exports = function () {
      console.log("hello!");
    }
```
app.js
```
    var hello = require("./hello.js");
```

- export a named function

hello.js
```
    exports.hello = function () {
      console.log("hello!");
    }
```
app.js
```
    var hello =
    require("./hello.js").hello;
```

RequireJS

- File and module loader for in-browser use
- Can also be used in other environments (like Node)
- Uses Asynchronous Module Loading (AMD)

- index.html

```
<script data-main="scripts/main"
src="scripts/require.js">
</script>
```

- main.js

```
require(["purchase"],
function(purchase){
  purchase.purchaseProduct();
});
```

- purchase.js

```
define(["credits","products"],
function(credits,products) {
  return {
    purchaseProduct: function() {
      …
```

ES6 Modules

- Compromise between AMD and CommonJS
  - compact syntax
  - support for asynchronous loading
- 2 Types
  - named exports
    - multiple per module
  - default exports
    - 1 per module

- Named export
- lib.js

```
export function square(x) {
  return x * x;
}
```

- main.js

```
import {square} from 'lib';
```

- Default export
- myFunc.js

```
export default function() {
…
};
```

- main.js

```
import myFunc from 'myFunc';
```

Front-end Modules

- Modules that will be part of the deployed code
- Examples include: jQuery, React, Modernizr, Backbone
- Can be managed "manually" or using npm

Manage Modules Manually

- To manage front-end modules manually, download them to your `src` directory.
- Important to keep them separate from your source code
- A common approach is to use a 'vendor' directory.

Front End Package Management with npm

- Manage packages with npm
- Port to browser using webpack

Chapter 6: Building and Refactoring

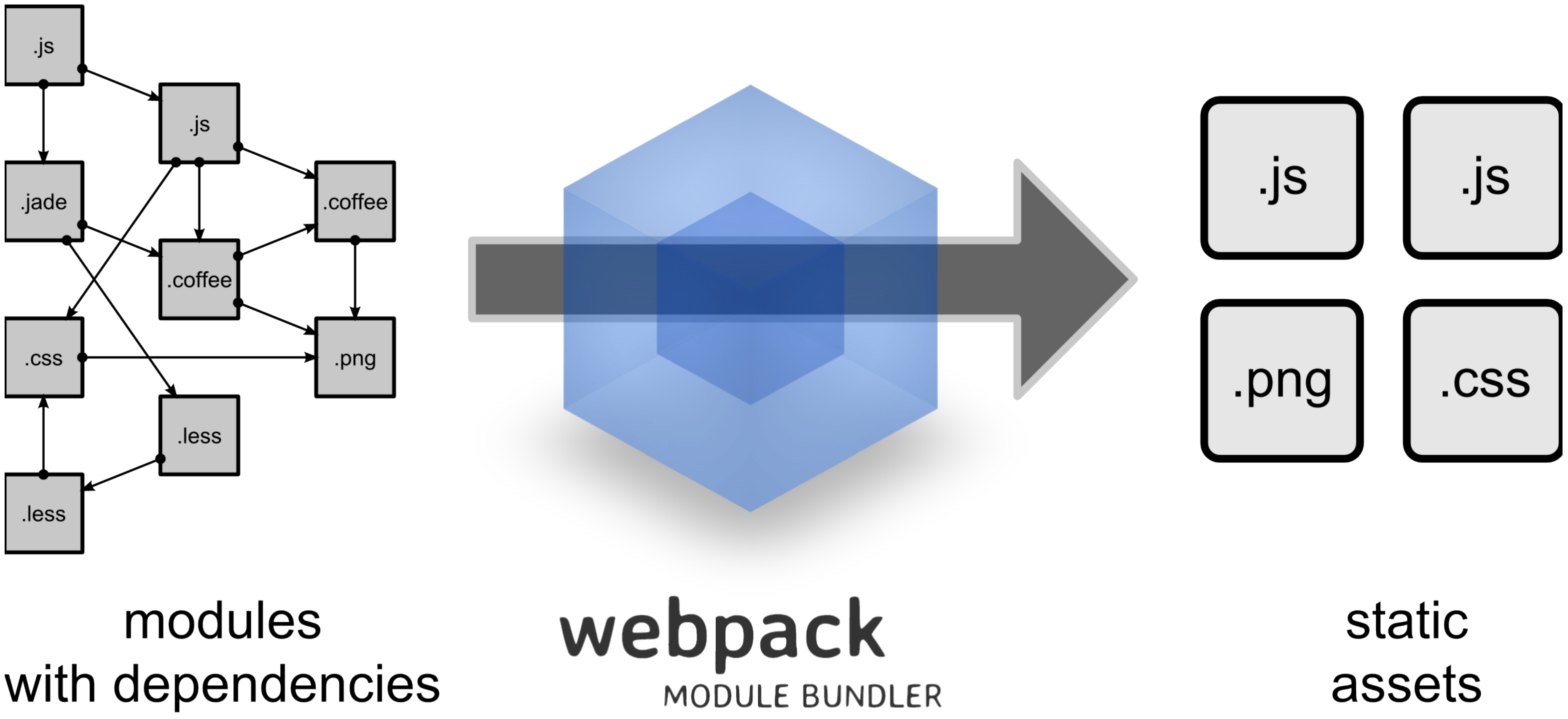
Objectives

- Use Webpack to build your production files
- Integrate Webpack into the automated build

Building the `dist` directory

- Use webpack to bundle everything
- Use babel to convert ES6 code to ES5
- Use run webpack and create the `dist` directory.

webpack


.js
.js
.jade
.coffee
.coffee
.css
.png
.less
.less
modules
with dependencies
webpack
MODULE BUNDLER
.js
.js
.png
.css
static
assets

How webpack Works

- Treats all assets as modules.
- Loads modules using most module styles (CommonJs, AMD, ES6) and creates a 'bundle'.
- Uses loaders to transform non-JS resources into JS
  - Loaders transform individual files
- Plugins extend webpack's capabilities
  - Plugins work on the entire bundle

Lab 12: Deploying with Webpack

Lab 13: README Update and Refactoring

Chapter 7: ES2015 (ES6) and Beyond

Objectives

- Learn what's new
- Use arrow functions and block-scoped variables
- Create generator functions
- Use classes and modules
- Transpile ES6 code to ES5 with Babel

Variable Scoping with `const` and `let`

- `const` creates constants
  - "immutable variables”
  - Cannot be reassigned new content
  - The assigned content isn't immutable, however,
    - If you assign an object to a constant, the object can still be changed.
- `let` creates block-scoped variables
  - Main difference between `let` and `var` is that the scope of `var` is the entire enclosing function.
  - Redeclaring a variable with `let` raises a syntax error
  - No hoisting
    - Referencing a variable in the block before the declaration results in a ReferenceError.

`let` vs. `var`

`var`

```
var a = 5;
var b = 10;

if (a===5) {
  var a = 4;
  var b = 1;
}
console.log(a); // 4
console.log(b); // 1
```

`let`

```
let a = 5;
let b = 10;

if (a===5) {
  let a = 4;
  let b = 1;
}
console.log(a); // 5
console.log(b); // 10
```

Block-scoped Functions

ES5

```
(function () {
  var foo = function () {
  return 1;
  }
  console.log(foo()); // 1

  (function () {
    var foo = function() {
    return 2;
    }
    console.log(foo()); // 2
    })();

  console.log(foo()); // 1
})();
```

ES6

```
{
  function foo () { return 1; }
  console.log(foo()); // 1
  {
    function foo () {
    return 2;
    }
    console.log(foo()); // 2

  }
    console.log(foo()); // 1
}
```

Arrow Functions

- aka "fat arrow" functions
- A more concise syntax for writing functions
  - ES5 (old way)

```
function increment(v){
   return v+1;
}
```

  - Arrow function

```
increment = (v) =>{v+1};
```

Arrow Function Parameters

- Surround parameter names with parentheses

```
(param1,param2,param3) => { statements }
```

- Parentheses are optional when there's only one parameter name

```
singleParam => { statements }
```

Arrow Functions (cont.)

- More intuitive handling of current object context.
  - ES6

```
this.nums.forEach((v) => {
  if (v % 5 === 0)
    this.fives.push(v);
});
```

  - ES5

```
var self = this;
this.nums.forEach(function (v) {
  if (v % 5 === 0)
    self.fives.push(v);
});
```

Default Parameter Handling

- ES6

```
function myFunc (x, y = 0, z = 13) {
  return x + y + z;
}
```

- ES5

```
function f (x, y, z) {
    if (y === undefined)
        y = 0;
    if (z === undefined)
        z = 13;
    return x + y + z;
};
```

Rest Parameter

- Aggregation of remaining arguments into single parameter of variadic functions.

```
function myFunc (x, y, ...a) {
    return (x + y) * a.length;
}
console.log(myFunc(1, 2, "hello", true, 7));
```

- http://jsbin.com/pisupa/edit?js,console

Spread Operator

- Spreading of elements of an iterable collection (like an array or a string) into both literal elements and individual function parameters.

```
var params = [ "hello", true, 7 ];
var other = [ 1, 2, ...params ];
console.log(other); // [1, 2, "hello", true, 7]

console.log(MyFunc(1, 2, ...params));

var str = "foo";
var chars = [ ...str ]; // [ "f", "o", "o" ]
```

- http://jsbin.com/guxika/edit?js,console

Template Literals

- String Interpolation

```
var customer = { name: "Penny" }
var order = { price: 4, product: "parts", quantity: 6 }
message = `Hi, ${customer.name}. Thank you for your order
of ${order.quantity} ${order.product} at ${order.price}.`;
```

- http://jsbin.com/pusako/edit?js,console

Template Literals (cont)

- Raw String Access

Allows you to access the raw template string content (without interpreting backslashes)

```
function tag(strings, ...values) {
  console.log(strings.raw[0]);
  // "string text line 1 \\n string text line 2"
}
tag`string text line 1 \n string text line 2`;
```

- http://jsbin.com/donibif/edit?js,console

Enhanced Object Properties

- Property Shorthand
  - Shorter syntax for properties with the same name and value
- ES5

```
obj = { x: x, y: y};
```

- ES6

```
obj = { x,y };
```

Enhanced Object Properties

- Computed names in object property definitions

```
let obj = {
  customer: "Nigel",
  [ "order" + getOrderNum() ]: 10
};
```

- http://jsbin.com/wejuqe/edit?js,console

Method notation in object property definitions

ES6

```
obj = {
    foo (a,b) {
    },
    bar (x,y) {
    }
};
```

ES5

```
obj = {
    foo: function(a, b) {
    },
    bar: function(x, y) {
    }
};
```

Array Matching

- Intuitive and flexible destructuring of Arrays into individual variables during assignment

```
var list = [ 1, 2, 3 ];
var [ a, , b ] = list; // a = 1 , b = 3
[ b, a ] = [ a, b ];
```

- http://jsbin.com/yafage/edit?js,console

Object Matching

- Flexible destructuring of Objects into individual variables during assignment

```
var { a, b, c} = {a:1, b:2, c:3};
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

- http://jsbin.com/kuvizu/edit?js,console

Symbol Primitive

- A **unique** and **immutable** data type.
- May be used as an identifier for object properties.
- Examples:
  - var sym1 = Symbol();
  - var sym2 = Symbol("foo");
  - var sym3 = Symbol("foo"); // Symbol("foo") !== Symbol("foo")
- Well-known Symbols
  - Built-in symbols, for example Symbol.iterator, which returns the default iterator for an object.

User-defined Iterators

- Customizable iteration behavior for objects
- In order to be iterable, an object must implement the iterator method -- the object, or one of the objects up its prototype chain) must have a property with a Symbol.iterator key.

```
var myIterable = {}
myIterable[Symbol.iterator] = function* () {
    yield 1;
    yield 2;
    yield 3;
};
[...myIterable] // [1, 2, 3]
```

For-Of Operator

- Convenient way to iterate over all values of an iterable object

```
let fibonacci = {
  [Symbol.iterator]() {
    let pre = 0, cur = 1
    return {
      next() {
        [ pre, cur] = [cur, pre + cur]
        return { done: false, value: cur }
        }
      }
    }
}
for (let n of fibonacci) {
    if (n > 1000)
      break;
    console.log(n);
}
```

- http://jsbin.com/nururoz/edit?js,console

Creating and Consuming Generator Functions

- A generator is a special type of function that works as a factory for iterators.

```
function* idMaker(){
  var index = 0;
  while(true)
    yield index++;
}
var gen = idMaker();
console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
```

- http://jsbin.com/pozite/edit?js,console

Class Definition

- ES6 introduces more OOP-style classes
- Can be created with Class declaration or Class expression

Class Declaration

```
class Square {
  constructor (height, width) {
    this.height = height;
    this.width = width;
    }
}
```

Class Expressions

- Can be unnamed

```
var Square = class {
  constructor(height,width) {
    this.height = height;
    this.width = width;
  }
};
```

- Or named

```
class Square {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

Class Inheritance

```
class Rectangle extends Shape {
    constructor (id, x, y, width, height) {
      super (id, x, y);
      this.width = width;
      this.height = height;
    }
}
class Circle extends Shape {
    constructor (id, x, y, radius) {
      super (id, x, y);
      this.radius = radius;
    }
}
```

Beyond ES2015

- ES2016
  - aka ES7
  - New Features
    - Exponentiation operator (**)
    - Array.prototype.includes
- ES2017
  - aka ES8
  - New Features
    - async/await
- ES.Next
  - Dynamic name for upcoming version

this, promises, map

ADVANCED JAVASCRIPT TOPICS

"use strict"

- Introduced with ES5
- Tells browsers to check for common errors and bad practices at run time
- Opts in to restricted variant of JavaScript
- To invoke:
  - put `"use strict;"` before any other statements
- Can also be invoked for functions:

```
function myStrictFunction(){
    'use strict';
    …
}
```

- Is implied in ES6 Modules and Classes
- Is redundant in properly written ES6 code.

Understanding `this`

- Allows functions to be reused with different context.
- Which object should be focal when invoking a function.

4 Rules of `this`

- Implicit Binding
- Explicit Binding
- New Binding
- Window Binding

What is `this`?

- When was function invoked?
- We don't know what `this` is until the function is invoked.

Implicit Binding

- `this` refers to the object to the left of the dot.

```
var author = {
  name: 'Chris',
  homeTown: 'Detroit',
  logName: function() {
    console.log(this.name);
  }
});

author.logName();
```

Explicit Binding

- .call, .apply, .bind

```
var logName = function() {
  console.log(this.name);
}

var author = {
  name: 'Chris',
  homeTown: 'Detroit"
}
logName.call(author);
```

Explicit Binding with .call

- Calls a function with a given `this` value and the arguments given individually.

```
var logName = function(lang1) {
  console.log(this.fname + this.lname + lang1);
};
var author = {
  fname: "Chris",
  lname: "Minnick"
};
var language = ["JavaScript","HTML","CSS"];
logName.call(author,language[0]);
```

Explicit binding with .apply

- Calls a function with a given `this` value and the arguments given as an array

```
logName = function(food1,food2,food3) {
  console.log(this.fname + this.lname);
  console.log(food1, food2,
    food3);
};
var author = {
  fname: 'Chris',
  lname: 'Minnick'
};

var favoriteFoods= ['Tacos','Soup','Sushi'];

logName.apply(author, favoriteFoods);
```

Explicit Binding with .bind

- Works the same as .call, but returns `new` function rather than immediately invoking the function

```
logName = function(food) {
  console.log(this.fname +' ' + this.lname +
        '\'s Favorite Food was ' + food);
};
var person = {
  fname: 'George',
  lname: 'Washington'
};
var logMe = logName.bind(person,'Tacos');

logMe();
```

- http://jsbin.com/xikuzog/edit?js,console

`new` Binding

- When a function is invoked with the `new` keyword, then this keyword inside the object is bound to the new object.

```
var City = function (lat,long,state,pop) {
  this.lat = lat;
  this.long = long;
  this.state = state;
  this.pop = pop;
};
var sacramento = new City(38.58,121.49,'CA',480000);
console.log (sacramento.state);
```

`window` Binding

- What happens when no object is specified or implied
- `this` defaults to the `window` object

```
var logName = function() {
  console.log(this.name);
}
var author = {
  name: 'Chris',
  homeTown: 'Detroit"
}
logName(author); //undefined(error in 'strict' mode)
window.author = 'Harry';
logName(author); // 'Harry'
```

`Array.map()`

- **Array.map()**
  - Creates a new array with the results of calling a provided function on every element in this array.
- Syntax
  - var new_array = arr.map(callback)
- Parameters passed to the callback
  - `currentValue`
    - The current element being processed
  - `index`
    - The index (number) of the current element
  - `array`
    - The array map was called upon

PROMISES

What Are Promises?

- An abstraction for asynchronous programming
- Alternative to callbacks
- A promise represents the result of an async operation
- Is in one of three states
  - pending – the initial state of a promise
  - fulfilled – represents a successful operation
  - rejected – represents a failed operation

Promises vs. Event Listeners

- Event listeners are useful for things that can happen multiple times to a single object.
- A promise can only succeed or fail once.
- If a promise has succeeded or failed, you can react to it at any time.
  - `readJSON(filename).then(success,failure);`

Why Use Promises?

- Chain them together to transform values or run additional async actions
- Cleaner code
  - Avoid problems associated with multiple callbacks
    - Callback Hell
    - Christmas Tree
    - Tower of Babel
    - Etc

Demo: Callback vs. Promise

- Callback

  -
    ```
    fs.readFile('text.txt', function(err, file){
          if (err){
                //handle error
          } else {
                console.log(file.toString());
          }
    });
    ```
- Promise

  -
    ```
    readText('text.txt')
          .then(function(data) {
                      console.log(data.toString());
                }, console.error
          );
    ```

Using Promises

```
const fs = require('fs');

function readFileAsync (file, encoding) {
    return new Promise(function (resolve, reject) {
        fs.readFile(file, encoding, function (err, data) {
            if (err) return reject(err);
            resolve(data);
        })
    })
}
```

```
readFileAsync('myfile.txt')
    .then(console.log, console.error);
```

Async / Await

Simplifies using promises.

```
async function f() {
    return 1;
}
```

An async function always returns a promise.

```
f().then(alert);
```

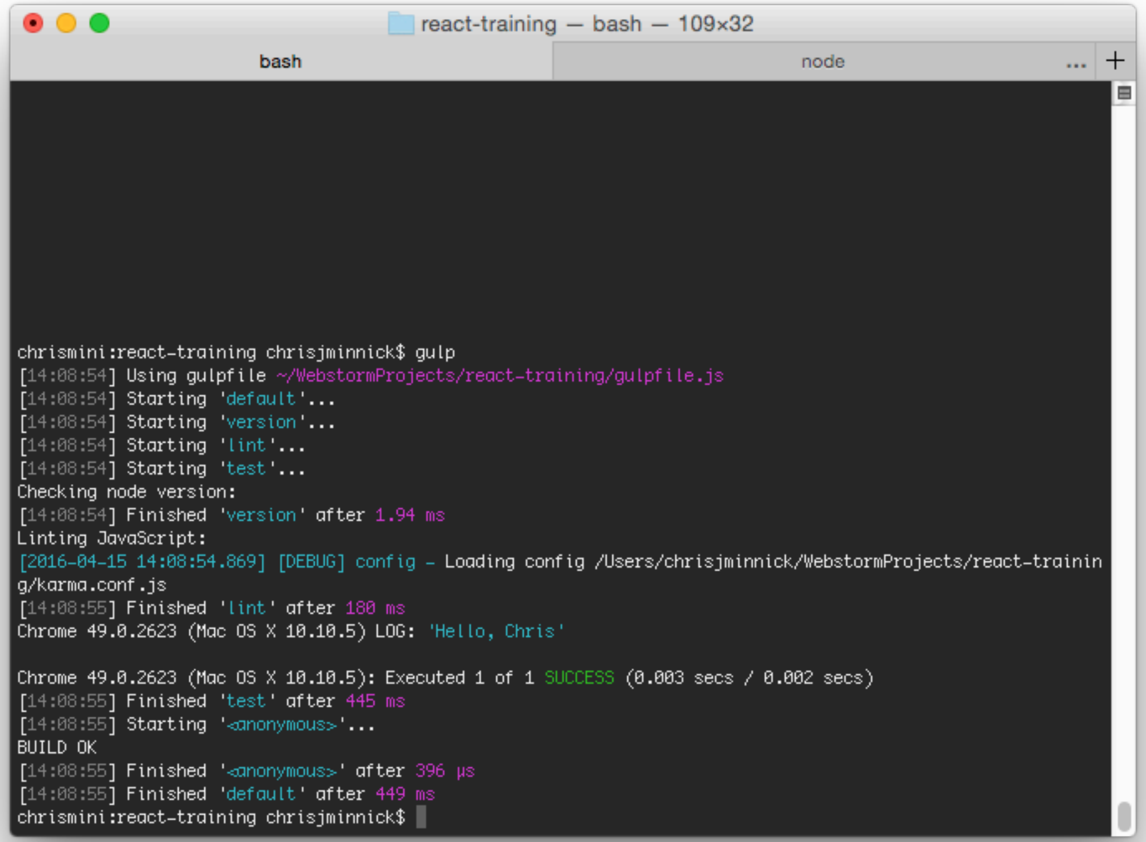
You can use the await keyword inside an async function to wait until the promise resolves.

```
async function getUser() {
    let response = await fetch('http://someurl.com/user.json');
    return response;
}
```

Babel

- Babel converts ES6 code into its equivalent ES5 so that it will run in today's browsers.

Lab 14: Transpiling with Babel



```
chrismini:react-training chrisjminnick$ gulp
[14:08:54] Using gulpfile ~/WebstormProjects/react-training/gulpfile.js
[14:08:54] Starting 'default'...
[14:08:54] Starting 'version'...
[14:08:54] Starting 'lint'...
[14:08:54] Starting 'test'...
Checking node version:
[14:08:54] Finished 'version' after 1.94 ms
Linting JavaScript:
[2016-04-15 14:08:54.869] [DEBUG] config - Loading config /Users/chrisjminnick/WebstormProjects/react-trainin
g/karma.conf.js
[14:08:55] Finished 'lint' after 180 ms
Chrome 49.0.2623 (Mac OS X 10.10.5) LOG: 'Hello, Chris'

Chrome 49.0.2623 (Mac OS X 10.10.5): Executed 1 of 1 SUCCESS (0.003 secs / 0.002 secs)
[14:08:55] Finished 'test' after 445 ms
[14:08:55] Starting '<anonymous>'...
BUILD OK
[14:08:55] Finished '<anonymous>' after 396 μs
[14:08:55] Finished 'default' after 449 ms
chrismini:react-training chrisjminnick$
```

Lab 15: Converting to ES6

```
export function greet(name) {

    return 'Hello, ' + name;

};


import * as sayHello from './sayHello.js';
```

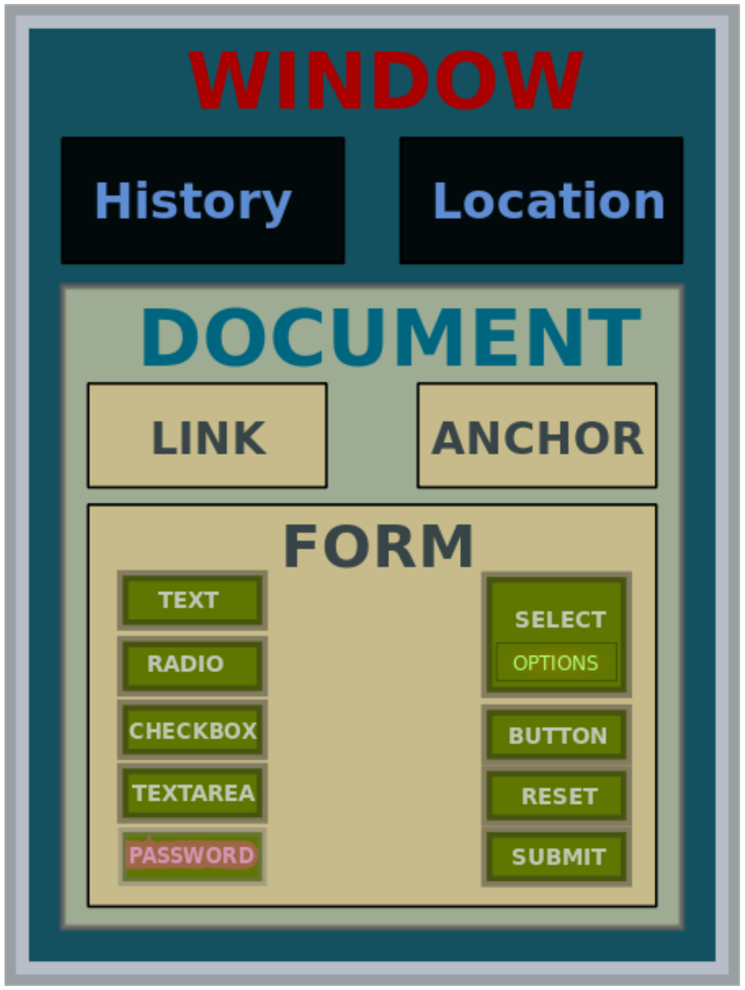
Chapter 8: The Document Object Model

Objectives

- Understand how the DOM works
- Select DOM nodes
- Manipulate the DOM with JavaScript

What is the DOM?

- JavaScript API for HTML documents
- Represents elements as a tree structure
- Objects in the tree can be addressed and manipulated using methods.



JohnManuel [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

Understanding Nodes

- DOM interfaces that inherit from Node
  - Document
  - Element
  - CharacterData
    - Text
    - Comment
  - DocumentType
- Nodes inherit properties from EventTarget

EventTarget

- An interface implemented by objects that can receive events (aka event targets)
- Examples:
  - Element
  - Document
  - Window
- Many event targets support setting event handlers.

DOM Events

- Things that happen in the DOM
  - `abort`
  - `beforeinput`
  - `blur`
  - `click`
  - `compositionend`
  - `compositionupdate`
  - `dblclick`
  - `error`
  - `focus`
  - `focusin`
  - `focusout`
  - `input`
  - `keydown`
  - `keyup`
  - `load`
  - `mousedown`
  - `mouseenter`
  - `mouseleave`
  - `mousemove`
  - `mouseout`
  - `mouseover`
  - `mouseup`
  - `resize`
  - `scroll`
  - `select`
  - `unload`
  - `wheel`

Other Events

- Many interfaces implement events or inherit events.
- Visit *http://developer.mozilla.org/en-US/docs/Web/Events.*

Element

- Interface for elements within a Document
- Inherits properties and methods from Node and EventTarget
- Most common properties:
  - `innerHTML`
  - `attributes`
  - `classList`
  - `id`
  - `tagName`
- Most common methods
  - `getElementById`
  - `addEventListener`
  - `querySelectorAll`

Manipulating HTML with the DOM

- You can get and set properties of HTML elements with JavaScript through the DOM

```
//starting HTML and DOM Element
<p id="favoriteMovie">The Matrix</p>

<script>
getElementById("favoriteMovie")
   .innerHTML = "The Godfather";
</script>

//updates DOM, which updates the browser
<p id="favoriteMovie">The Godfather</p>
```

Manipulating HTML with the DOM

```
<ol id="favoriteSongs">
  <li class="song"></li>
  <li class="song"></li>
  <li class="song"></li>
</ol>

<script>
var mySongs=document
   .querySelectorAll("#favoriteSongs .song");
mySongs[0].innerHTML = "My New Favorite Song";
</script>
```

Manipulating HTML with JQuery

```
<ol id="favoriteSongs">
  <li class="song"></li>
  <li class="song"></li>
  <li class="song"></li>
</ol>

<script>
$("#favoriteSongs .song").first()
   .html("My New Favorite Song");
</script>
```

Manipulating HTML with React

```
<div id="favoriteSongs"></div>

<script>
class FavoriteSongs extends React.Component({
  render() {
    return (
    <ol>
      <li className="song">{this.props.song}</li>
      <li className="song"></li>
      <li className="song"></li>
    </ol>
    );
  }
});
ReactDOM.render(<FavoriteSongs song="My New Favorite Song" />,
    document.getElementById('favoriteSongs'));
</script>
```

Chapter 9: Introduction to React.js

Objectives

- Understand React
- Explore the Virtual DOM
- Explain one-way data binding
- Create React Components
- Manage state

What is React.js?

- A JavaScript library for building UIs
- Wraps an imperative API (DOM) with a declarative one
- Is comperable to Angular Directives
- Can be plugged into a framework's component technology
- Doesn't have to be used with MVC

Imperative API vs. Declarative API

- Imperative
  - Focuses on the steps to complete a task
  - Example:
    - Walk to the stairs
    - Walk down stairs
    - Go to the kitchen
    - Open refrigerator
    - Take out salami, cheese, mustard
    - Put salami, cheese, mustard on bread
- Declarative
  - Focuses on what to do without saying how
    - Bring me a sandwich.

Imperative vs. Declarative Screen Updates

- Imperative
  - Change the greeting to: "Dear Mr. Smith,”
  - Change the beginning of the first paragraph to "We are pleased to”
  - Changing the closing to "Sincerely,”
- Declarative
  - Make it look like this:

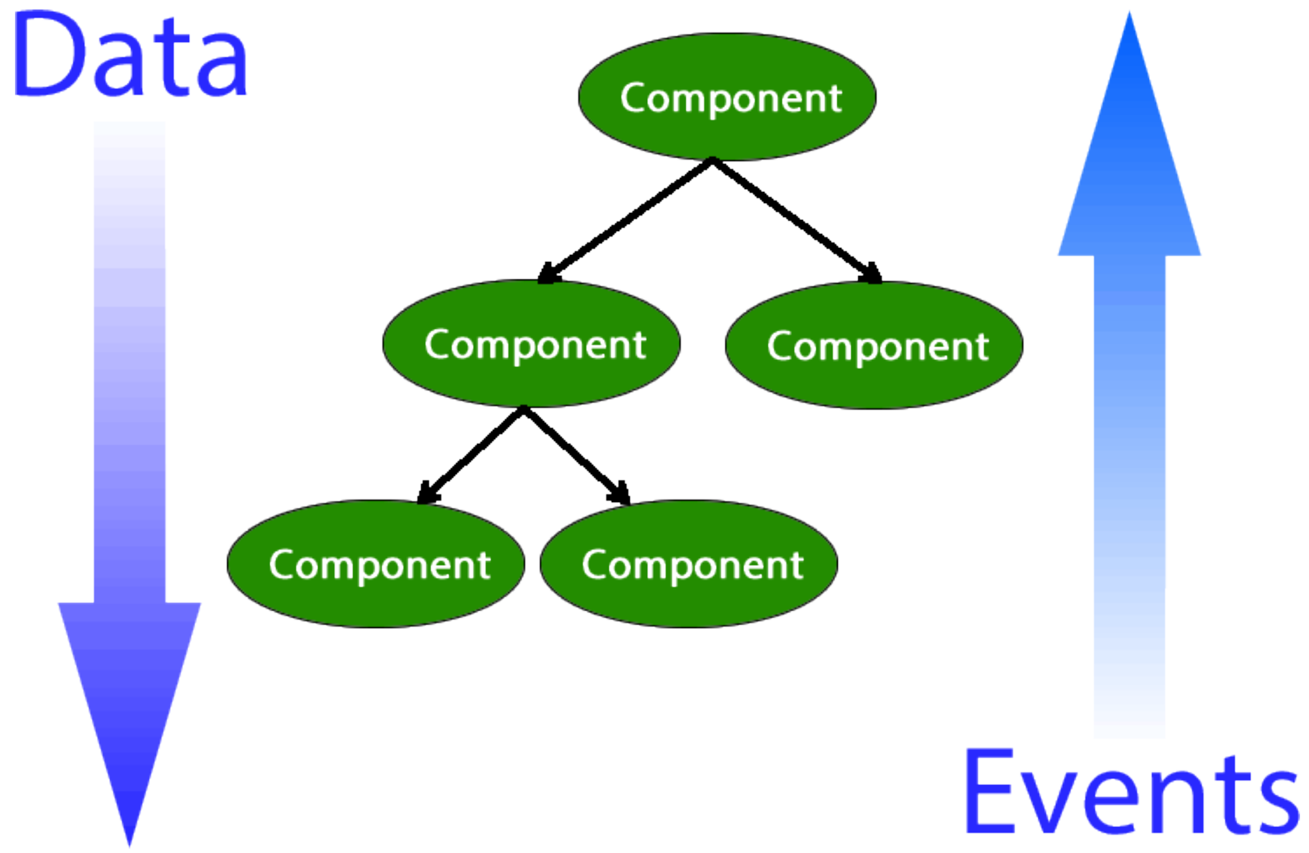
```
Dear Mr. Smith,
We are pleased to inform you that you have been
selected!
Sincerely,
The Management
```

Key Points

- One-way data flow
  - A component can't modify properties passed to it.
- Virtual DOM
  - Manages HTML DOM updates
- JSX
  - Easy XML template language.
- Not just for browser output
  - Architecture can apply to native apps, canvas

One-way Data Flow

- Each UI element represents one component
- All data flows from owner to child



Virtual DOM

1. Virtual DOM is updated (in memory) as the **state** of the data model changes.
2. React calculates the difference between the Virtual DOM and the real DOM.
3. React updates only what needs to be updated in the DOM.
4. Batches changes

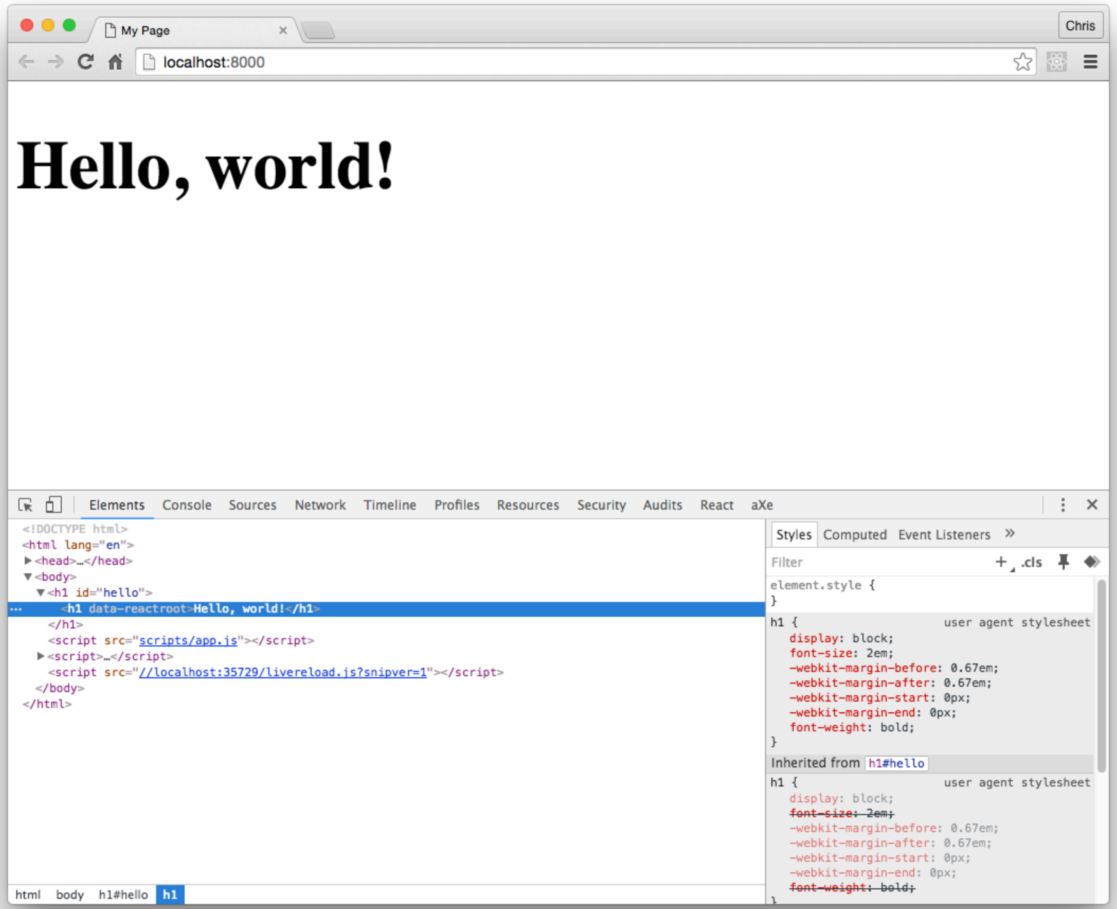
Virtual DOM vs. HTML DOM

- Virtual DOM is a local and simplified copy of the HTML DOM
- (It's an abstraction of an abstraction)
- The goal of the Virtual DOM is to only re-render when the **state** changes.
  - This makes it more efficient than direct DOM manipulation.
- Developers can write code as if the entire tree is being re-rendered.
  - This makes it easier to understand.
- Behind the scenes, React/Virtual DOM works out the details and creates a patch for the HTML DOM, which causes the browser to re-render the changed part of the scene.

State Machines

- React thinks of UIs as being simple state machines.
- A state machine has:
  - An initial state or record of something stored someplace
  - A set of possible input events
  - A set of new states that may result from the input
  - A set of possible actions or output events that result from a new state

Lab 16, Part 1: Hello React!



Hello, world!

```
<!DOCTYPE html>
<html lang="en">
▶<head>…</head>
▼<body>
  ▼<h1 id="hello">
      <h1 data-reactroot>Hello, world!</h1>
    </h1>
    <script src="scripts/app.js"></script>
  ▶<script>…</script>
    <script src="//localhost:35729/livereload.js?snipver=1"></script>
  </body>
</html>
```

```
element.style {
}
h1 {                          user agent stylesheet
   display: block;
   font-size: 2em;
   -webkit-margin-before: 0.67em;
   -webkit-margin-after: 0.67em;
   -webkit-margin-start: 0px;
   -webkit-margin-end: 0px;
   font-weight: bold;
}
Inherited from h1#hello
h1 {                          user agent stylesheet
   display: block;
   ~~font-size: 2em;~~
   -webkit-margin-before: 0.67em;
   -webkit-margin-after: 0.67em;
   -webkit-margin-start: 0px;
   -webkit-margin-end: 0px;
   ~~font-weight: bold;~~
}
```

Understanding Components

- Created by extending `React.Component`
  - Returns a single element OR an Array of elements (as of v16)
    - May contain nested elements, however.
  - You can have multiple instances of components
    - For example, you might have components called NavButton or Invitee
  - An element describes a component and tells React what you want to see on screen.
- Components should be reusable as well as composable.

`React.render()`

- `fn(d) = V`
- Give the function data and you get a View.
- Return value must
  - be a single element:
    - `return (<div><p><em></em></p></div>);` = correct
    - `return (<div></div><p></p>);` = error
  - or a fragment

    -
      ```
      return (<React.Fragment>
                  <h1>this is cool</h1>
                  <h2>this too</h2>
                  as of React 16.2</React.Fragment>
              );
      ```
  - or an array with unique key values

    -
      ```
      return [
            <li key="1">item 1</li>
            <li key="2">item 2</li>
      ];
      ```
  - or a string
    - `return "this is totally valid too";`

React.Fragment

- Allows a component to return multiple elements by grouping them together without added extra nodes to the DOM.

```
render() {
  return (
    <React.Fragment>
      <td>table row</td>
      <td>Another row</td>
    </React.Fragment>
    );
}
```

React Fragment Shorthand

```
render() {
  return (
    <>
      <td>table row</td>
      <td>Another row</td>
    </>
    );
}
```

ReactDOM

- React package for working with the DOM
  - Generally only needed at the top level of your application.
- Methods
  - `findDOMNode`
  - `render`
  - `unmountComponentAtNode`
    - Removes a mounted component from the DOM and cleans up its event handlers and state.
  - react-dom/server
    - For rendering static markup on the server.
    - `ReactDOMServer.renderToString()`
    - `ReactDOMServer.renderToStaticMarkup()`

`ReactDOM.findDOMNode`

- `findDOMNode(component)`
- If the component has been mounted, returns the corresponding native browser DOM element

ReactDOM.unmountComponentAtNode

- Removes a mounted component from the DOM and cleans up its event handlers and state

```
React.unmountComponentAtNode
    (document.getElementById('container'));
```

`ReactDOM.render`

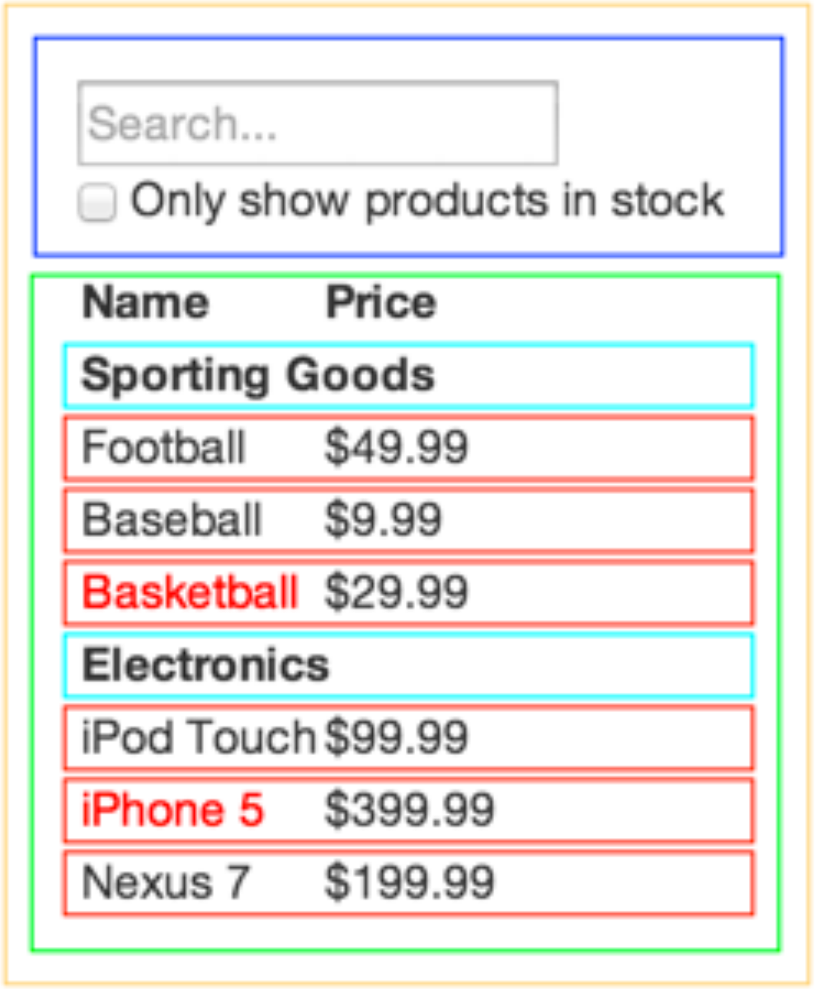
- `ReactDOM.render(reactElement, domContainerNode)`
- Renders a `reactElement` into the DOM in the supplied container
- Replaces any existing DOM elements inside the container node when first called
- Later calls using DOM diffing algorithm for efficient updates

React Development Process

1. Break the UI into a component hierarchy
2. Build a static version in React using props
   – Props pass data from parent to child
3. Identify the minimal representation of UI state
4. Identify where your state should live
5. Add inverse data flow

Step 1 - Break up the UI

```
var PRODUCTS = [
  {category:
'Sporting Goods',
price: '$49.99',
stocked: true, name:
'Football'},
  {category:
'Sporting Goods',
price: '$9.99',
stocked: true, name:
'Baseball'}
];
```



Name	Price
Sporting Goods	
Football	$49.99
Baseball	$9.99
Basketball	$29.99
Electronics	
iPod Touch	$99.99
iPhone 5	$399.99
Nexus 7	$199.99

Step 2 - Static Version

- Render the UI from the data model with no interactivity
- Goal is to have a library of reusable components that render the data model

```
class ProductCategoryRow extends React.Component({
  render () {
    return (
    <tr>
      <th colSpan="2">
        {this.props.category}
      </th>
    </tr>
    );
  }
});
```

Step 3 - Minimal UI State

- Is it state?
  - Is it passed from the parent via props?
    - Probably not state
  - Does it change over time?
    - Might be state
  - Can you compute it based on any other state or props in your application?
    - Probably not state

Step 4 – Where Should Your State Live?

- For each piece of state:
  - Identify every component that renders something based on that state.
    - Find a common owner component
    - The common owner or a component higher in the hierarchy should own the state.
    - If you can't find a common owner, create a new component higher up in the hierarchy just for holding the state.

Step 5 – Add Inverse Data Flow

- Use a function to update state in components higher up in the hierarchy.

Props vs. State

Props

- Passed to the child within the render method of the parent
- Immutable
- Better performance

State

- State of the parent becomes prop of child
- Mutable

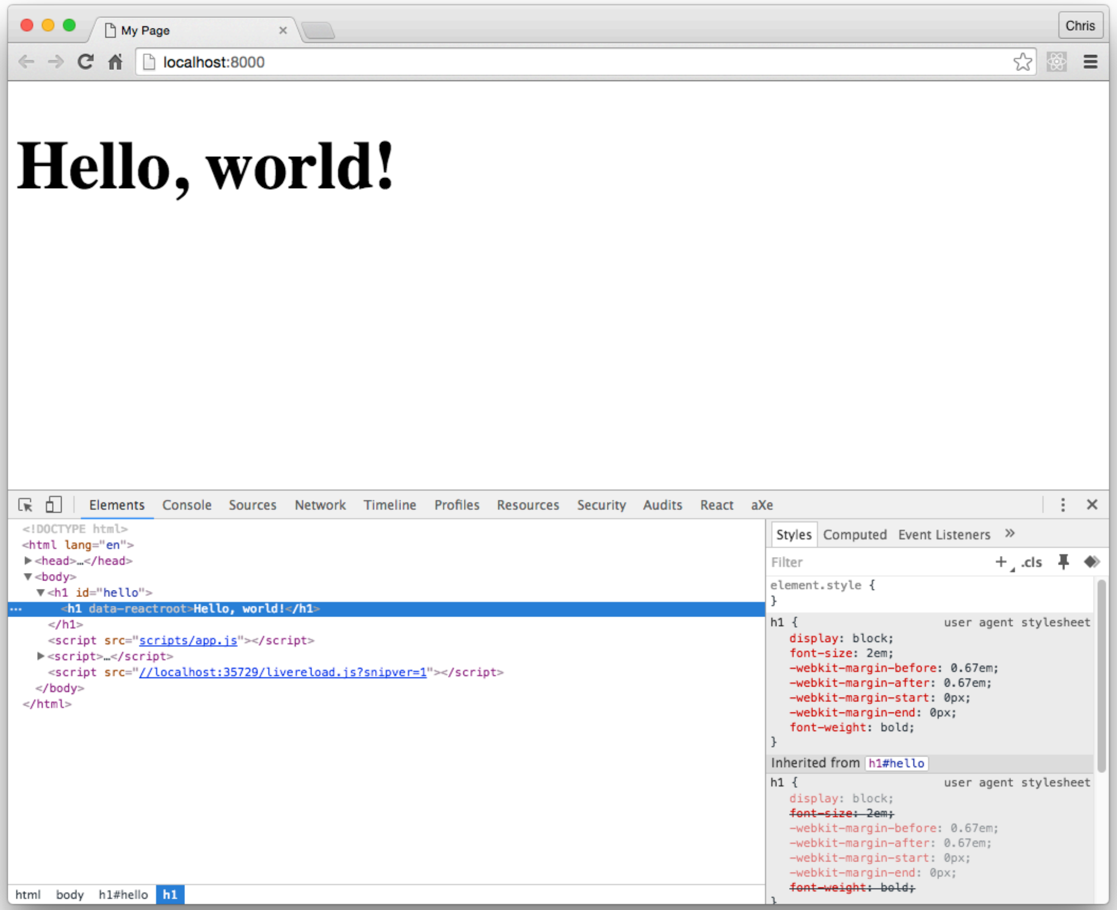
Setting Initial State

```
class MyComponent extends React.Component {
  constructor() {
      this.state = { /* some initial state */ }
  }
```

super()

- Used for calling methods on the parent.
- ES6 classes must call `super()` if they are subclasses.
- If you don't have a constructor, you don't need to call `super()`. React will automatically make `this.props` available to the subclass.
- If you want to use `this.props` in your constructor, you need to call `super(props)` in the constructor.

Lab 16, Parts 2-3: Your first component



My Page
Chris
localhost:8000
Elements Console Sources Network Timeline Profiles Resources Security Audits React aXe
Styles Computed Event Listeners
Filter
.cls
html body h1#hello h1

Hello, world!

```
<!DOCTYPE html>
<html lang="en">
▶<head>…</head>
▼<body>
  ▼<h1 id="hello">
      <h1 data-reactroot>Hello, world!</h1>
    </h1>
    <script src="scripts/app.js"></script>
  ▶<script>…</script>
    <script src="//localhost:35729/livereload.js?snipver=1"></script>
  </body>
</html>
```

```
element.style {
}
h1 {                          user agent stylesheet
   display: block;
   font-size: 2em;
   -webkit-margin-before: 0.67em;
   -webkit-margin-after: 0.67em;
   -webkit-margin-start: 0px;
   -webkit-margin-end: 0px;
   font-weight: bold;
}
Inherited from h1#hello
h1 {                          user agent stylesheet
   display: block;
   ~~font-size: 2em;~~
   -webkit-margin-before: 0.67em;
   -webkit-margin-after: 0.67em;
   -webkit-margin-start: 0px;
   -webkit-margin-end: 0px;
   ~~font-weight: bold;~~
}
```

Chapter 10:
JSX

Objectives

- Know how to write JSX
- Use React with JSX
- Use React without JSX

What is JSX?

Preprocessor that adds XML syntax to JavaScript.

Takes XML input and produces native JavaScript.

JSX is not exactly HTML

- You can use HTML entities within JSX text
  - `<div>&copy; all rights reserved</div>`
- React will not render properties on HTML elements that don't exist in the HTML spec.
  - preface custom properties with data-
    - `<div data-custom-attribute="foo" />`
  - Arbitrary attributes are supported on custom elements
    - `<x-my-component custom-attribute="foo" />`

JSX is not exactly HTML (cont)

- XML syntax required
  - Elements must be closed
- Attributes use DOM property names
  - `className` instead of `class`
- Attributes become props in the child
- React components start with upper-case
- HTML tags start with lower-case

Using React with JSX

```
class LoginBox extends React.Component{
render() {
  return (
  <div>
  <label>Log In <input type="text" id="username"
      placeholder={this.props.placeholderText} />
  </label>
  </div>
  );
}
};
ReactDOM.render(<LoginBox placeholderText="Enter
Your Email" />, document.getElementById("login"));
```

Using React without JSX

```
...
  render() {
    return React.createElement("div",null,
      React.createElement("label",null,"Log In",
        React.createElement("input",
         { type: "text", id: "username" })
      )
    );
  }
});
ReactDOM.render(React.createElement(LoginBox, null),
document.getElementById("login"));
```

Expressions in JSX

- Use curly braces around JavaScript expressions to include them in JSX.

```
return(
  <HelloWorld
    name={this.props.firstname +
      this.props.lastname} />

  <div>
    {isLoggedIn ? <Logout /> : <Login />}
  </div>;
  );
```

Expressions in JSX

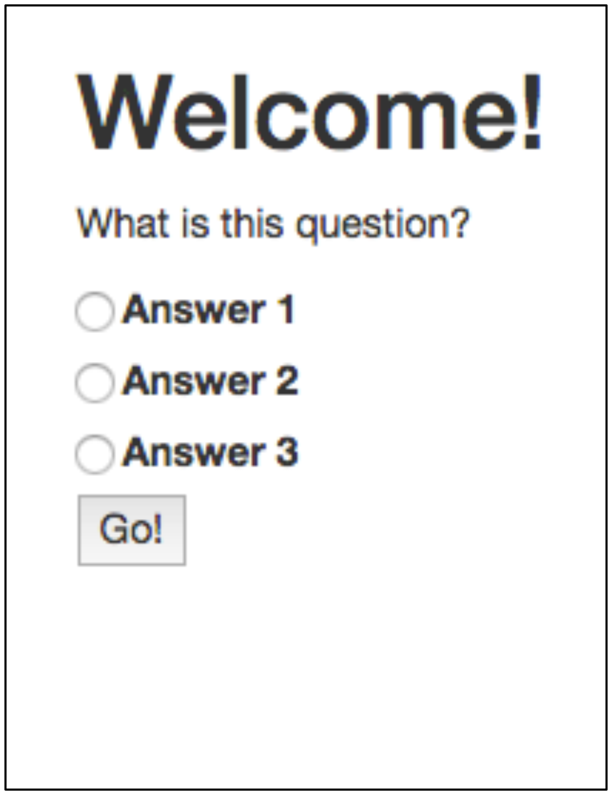
```
class Logout extends React.Component{
  render() {
    return (<div>logout</div>);
  }
};

class Login extends React.Component{
  render() {
    return(<div>login</div>);
  }
};

class Container extends React.Component{
  render() {
    return(
    <div>
    {this.props.isLoggedIn ? <Logout /> : <Login />}
    </div>);
  }
};

ReactDOM.render(<Container isLoggedIn={false} />, document.getElementById("app"));
```

Precompiled JSX

- Two ways to use JSX
  - Compile to JavaScript during build
    - Use Babel
  - Serve JSX and compile in the browser
    - Use JSXTransformer
      - Intended only for prototypes
      - **Deprecated**

Lab 17 - HTML to JSX

- Convert an HTML mockup to static React components



Chapter 11: React Components

Objectives

- Understand component life-cycle
- Use events and dispatching
- Communicate between components
- Test React components

Creating Components

- Two techniques
  - `React.createClass`
  - `React.Component`
- `React.Component` is the ES6 way.
- `React.createClass` is deprecated as of v15.5.0.

`React.createClass`

`createClass(object specification)`

- Creates a component, given a specification.
- Implements a `render` method
  - `render` method is required.
    - `render` returns one single child.
      - Can be a virtual representation of a DOM component or another component that you've created
        - » may have a deep child structure
- Is still available for React 16+ as create-react-class

React.Component

- Base class for React Components when defined using ES6 classes.

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="Ann" />, mountNode);
```

React.createElement

```
createElement(
  string/ReactClass type,
  [object props],
  [children …]
)
```

- Create and return a new ReactElement of the given type.
- Type argument can be an HTML tag name string (`'div'`, `'span'`, etc.), or a React Class (created via `React.createClass`).
- JSX gets compiled to React.createElement

Communication Between Components

- Parent to Child
  - pass props
  - Ref functions
    - Call a method in a child component from the parent component
    - `this.refs.<ref name>.<function name>`

props

- Props are passed from the parent component to its children
- JSX attributes become a single object in the child.

- Parent:

```
<Hello name="Chris" />
```

- Child:

```
class Hello extends React.Component {
  render() {
    return (<h1>Hello, {this.props.name}</h1>);
  }
}
```

Communication Between Components (cont.)

- Child to Parent
  - Callback Functions
    - Parent passes a function to a child
      - `<MyChild myFunc={this.handleChildFunc.bind(this)} />`
    - Child calls the function
      - `this.props.myFunc();`
  - Event Bubbling
    - Parent can capture DOM events that happen in children.

```
class ParentComponent extends React.Component {
  render() {
    return (
      <div onKeyUp={this.handleKeyUp.bind(this)}>
        // Any number of child components can be added here.
      </div>
    );
  }
  handleKeyUp(event) {
    // This function will be called for the 'onkeyup' event in any <input/>
    // fields rendered by any of my child components.
  }
}
```

Communication Between Components (cont.)

- Between Siblings
  - Use a parent component.
- Any to Any
  - Observer Patterns
    - Components subscribe to messages.
    - Other components publish messages to subscribers.
    - Subscribe in componentDidMount
    - Unsubscribe in componentWillUnmount
    - Can use a library such as EventEmitter (https://github.com/Olical/EventEmitter)
  - Context
    - Provides data to an entire subtree
    - https://facebook.github.io/react/docs/context.html
  - ~~Global Variables~~
    - Best not to use.

Using refs

- refs can be used to get access to a DOM node or to an instance of a component.

```
...
update(e){
    this.setState(
        {message:this.refs.message.value}
    );
}
...
render(){
  return(
<div>
    <input type="text" ref="message"
        onChange={this.update.bind(this)} />
    </div>
);
}
```

Ref Callback

- A special attribute you can attach to any component
- Takes a callback function
- The callback will be executed immediately after the component mounts or unmounts.
- When used on an HTML element, the ref callback receives the underlying DOM node as its argument.

Communicating Parent > Child with Ref

```
class TheChild extends React.Component {
  myFunc() {
    return 'hello';
  }
}
class TheParent extends React.Component {
  render() {
    return (
      <TheChild ref='foo' />
    );
  }

  componentDidMount() {
    var x = this.refs.foo.myFunc();
    // x is now 'hello'
  }
}
```

React.createRef

- As of React 16.3, you can create refs using React.createRef

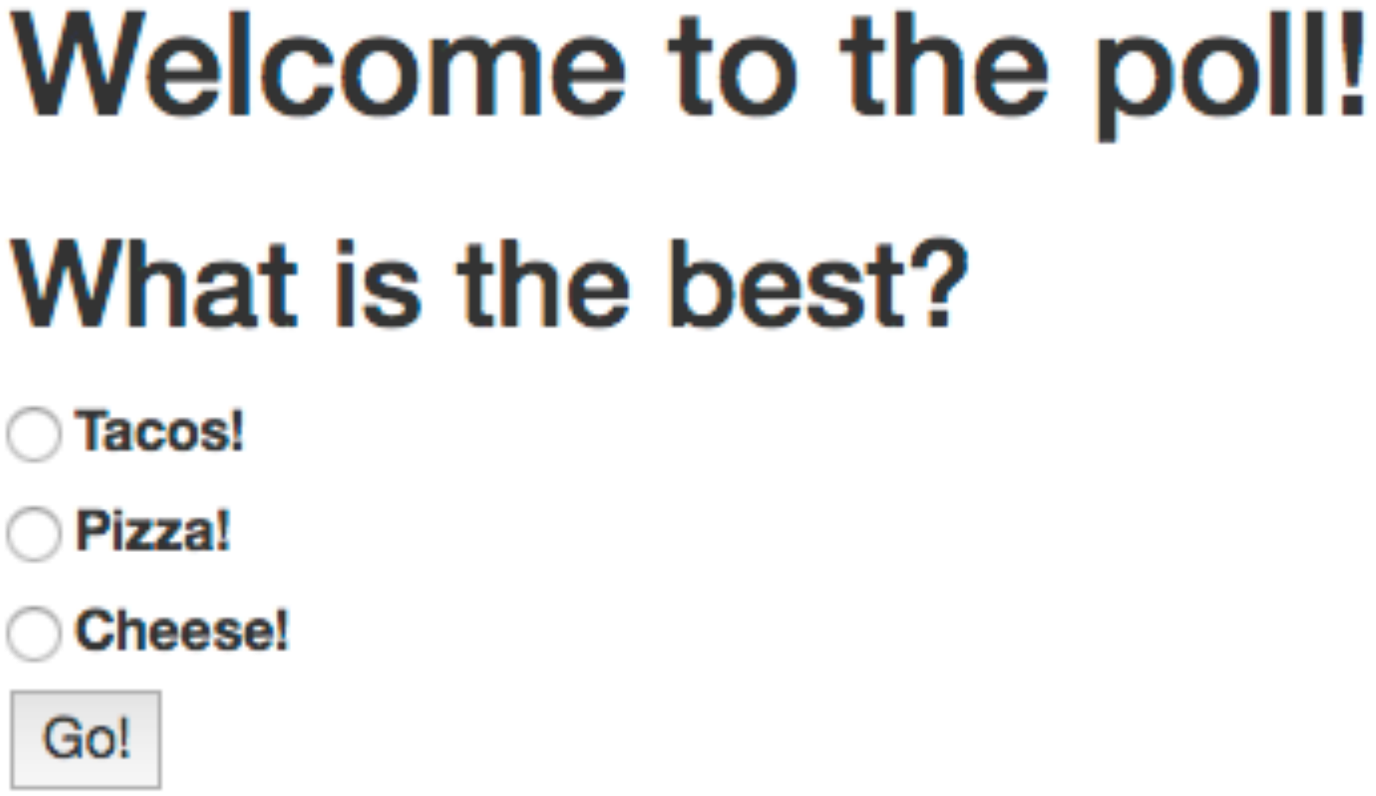
```
class MyComponent extends React.Component {
    constructor(props) {
        super(props);
            this.myRef = React.createRef();
        }
        render() {
            return <div ref={this.myRef} />;
    }
 }
```

When to Use Refs

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.

Avoid using refs for anything that can be done declaratively.

Lab 18: Passing Props



Welcome to the poll!

What is the best?

Styles in React

- Facebook recommends using inline styles, set using JavaScript.
- Pass styles into JSX using objects containing style properties.
- Properties are camelCased versions of the CSS properties.

```
var headingStyle = {
  color: "blue",
  textTransform: "uppercase"
};
return (<h1 style={headingStyle}>
        Welcome!</h1>);
```

Styled Components

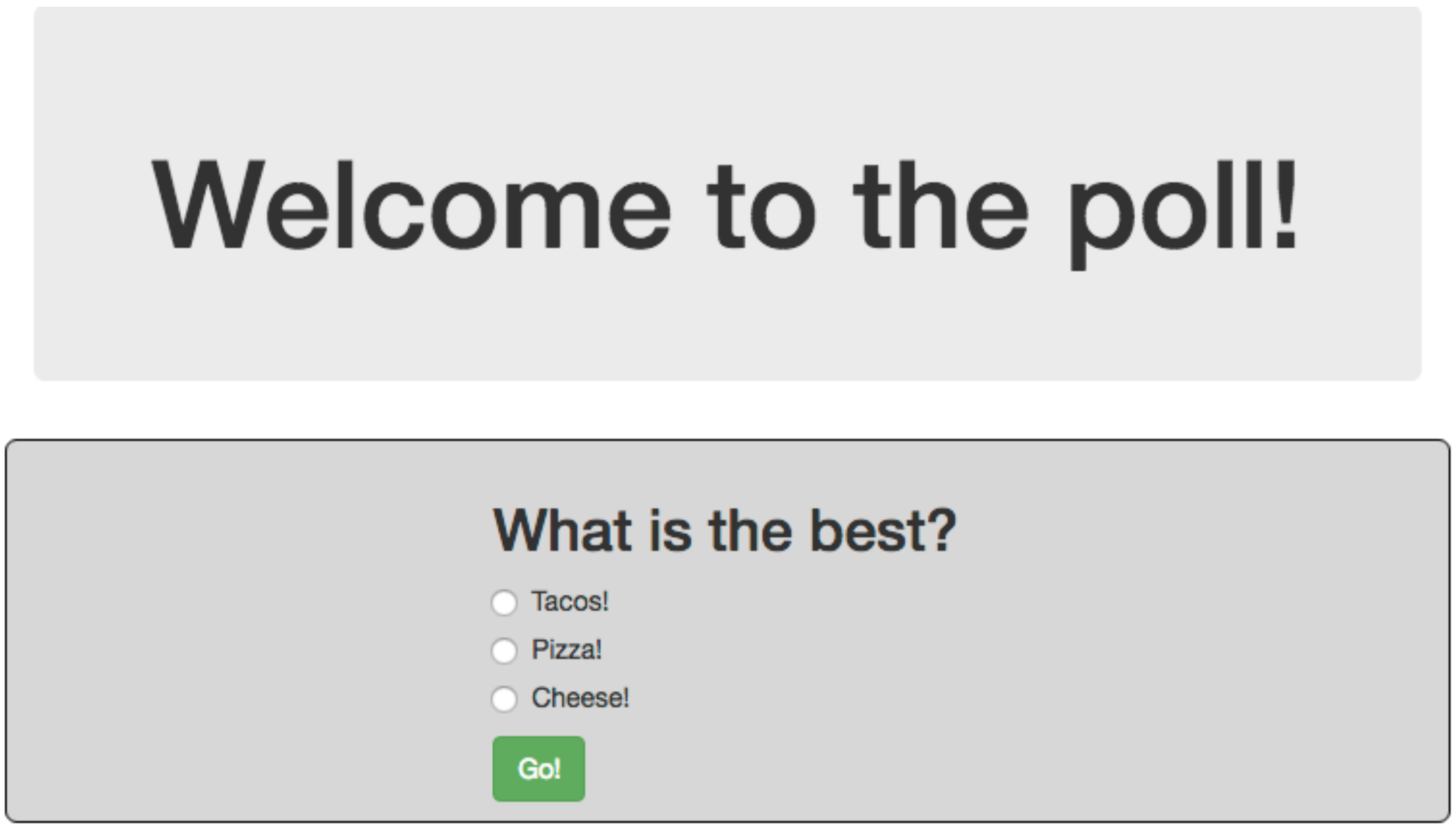
1. import styled from 'styled-components';
2. Call styled.[object] function, passing in style info using Tagged Template Literal Notation.

```
const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`;

<Title>This is a styled component</Title>
```

Styles in React

- Other ideas
  - Radium
    - Resolves nested objects and allows use of media queries and other complicated styling needs.
  - CSS Modules
    - Create separate style modules and import and use them in components.
  - Use a CSS library (Bootstrap) for layout, inline for presentation.
- How to do CSS in React is still very much being worked out.
  - Lots of opinions
- Big question: Should we even need to know CSS?

Lab 19: Style in React



Forms

Objectives

- Use Controlled Components
- Use Uncontrolled Components

Forms Have State

- They are different from other native components because they can be mutated based on user interactions.
- Properties of Form components
  - value
    - supported by `<input>` and `<textarea>`
  - checked
    - supported by `<input type="checkbox | radio" />`
  - selected
    - supported by `<option>`
- `<textarea>` should be set with value attribute, rather than children in React

Form Events

- Form components allow listening for changes using onChange
- The onChange prop fires when:
  - The value of `<input>` or `<textarea>` changes.
  - The checked state of `<input>` changes.
  - The selected state of `<option>` changes.

Controlled Components

- A controlled `<input>` is one with a value prop.
- User input has no effect on the rendered element.
- To update the value in response to user input, you can use the onChange event.
- Controlled vs. Uncontrolled Demo:
  - https://goo.gl/wMMbVc

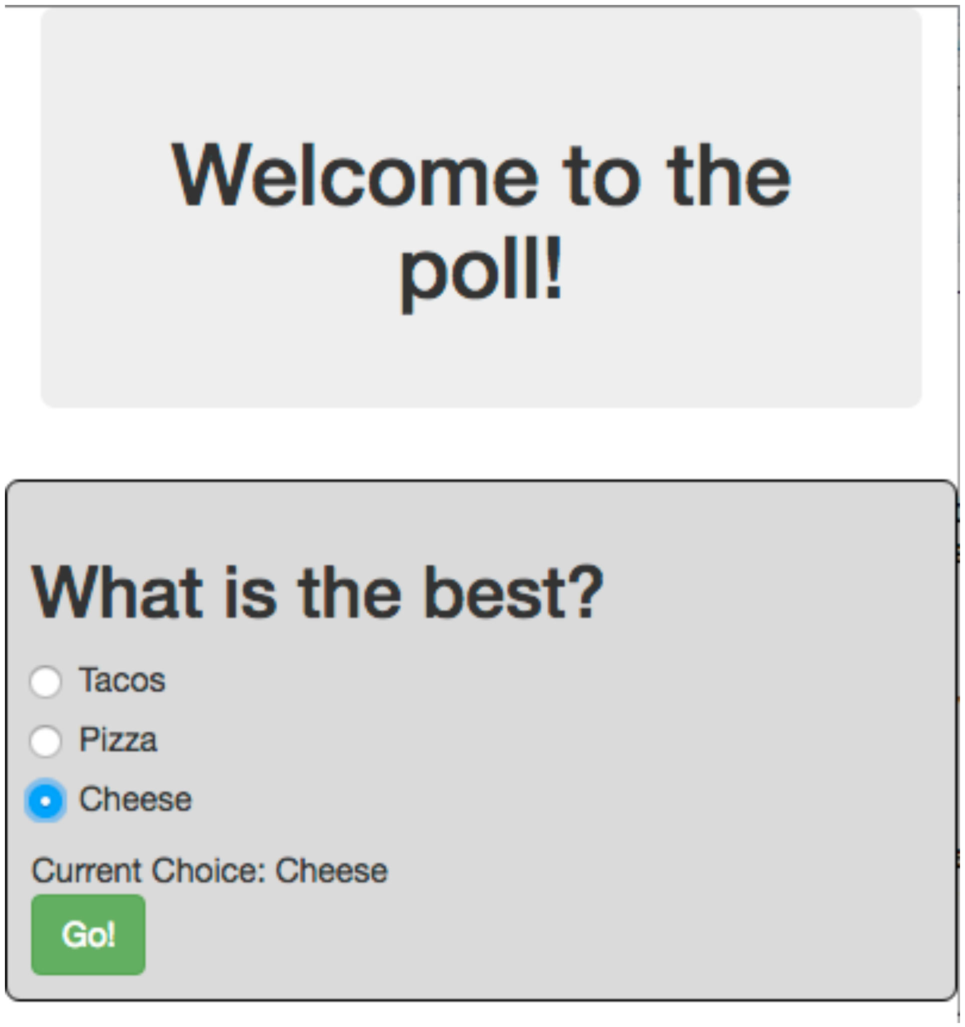
Uncontrolled Components

- An `<input>` without a value property is an uncontrolled component.

```
render() {
  return <input type="text" />;
}
```

- User input will be reflected immediately by the rendered element.
- Maintains its own internal state

Lab 20: Controlling the Form



Welcome to the poll!

What is the best?

Tacos [✓ Cheese] Pizza

Current Choice: Cheese

COMPONENT DESIGN

F.I.R.S.T.

- React Components should be:
  - **F**ocused
  - **I**ndependent
  - **R**eusable
  - **S**mall
  - **T**estable

Single Responsibility

- A component should only do one thing.
- If it ends up growing, it should be decomposed into smaller subcomponents.
- A responsibility is a "reason to change."
- Single responsibility makes components more robust.

"A class should have only one reason to change.

-Robert C. Martin

When a component is a result of props alone (no state), it can be written as a **pure function**.

Pure Functions

- Pure functions always return the same result given the same arguments.
- Pure function's execution doesn't depend on the state of the application.
- Pure functions don't modify the variables outside of their scope (no side effects).

Function Comparison

slice()

```
var toppings =
['cheese','pepperoni','mushrooms'];

toppings.slice(0,2);
// ["cheese", "pepperoni"]
toppings.slice(0,2);
// ["cheese", "pepperoni"]
toppings.slice(0,2);
// ["cheese", "pepperoni"]
```

- Always returns the same result given the same arguments
- Doesn't depend on the state of the application
- Doesn't modify variables outside its scope
- **It's a Pure Function!**

splice()

```
var toppings =
['cheese','pepperoni','mushrooms'];

toppings.splice(0,2);
// ["cheese", "pepperoni"]
toppings.splice(0,2);
// ["mushrooms"]
toppings.splice(0,2);
// []
```

- **Not Pure!**

Benefits of Pure Functions

- Easy to test
- Easy to reason about
- Easy to reuse
- Easy to reproduce the results

React.PureComponent

- If your Component returns the same result given the same props and state, use React.PureComponent
- PureComponent does a shallow state and prop comparison and doesn't update if the component is unchanged.
- May give a performance boost

```
class MyComponent extends React.PureComponent {
  ...
}
```

Stateless Functional Components

- If your component only has a render method and optional props, you can just create a normal JavaScript function.

```
function HelloWorld(props){
    return (
        <p>Hello {props.name}</p>
    );
}
reactDOM.render(<HelloWorld name='Chris' />,
document.getElementById("app"));
```

React.memo

- Works the same as React.PureComponent, but for functional components.

```
const MyComponent = React.memo(
   function MyComponent(props) {
      ...
   });
```

Lab 21: Refactoring the App

Component Life-Cycle Events

- 2 Categories
  - Mount / Unmount
  - When component receives new data

Life-Cycle Methods

- Methods of components that allow you to hook into views when specific conditions happen.

Mount/Unmount

- Mount and unmount methods are called when components are added to the DOM (Mount) and removed from the DOM (Unmount).
- Each is invoked only once in the lifecycle of the component
- Used for:
  - establish default props
  - set initial state
  - make AJAX request to fetch data for component
  - set up listeners
  - remove listeners

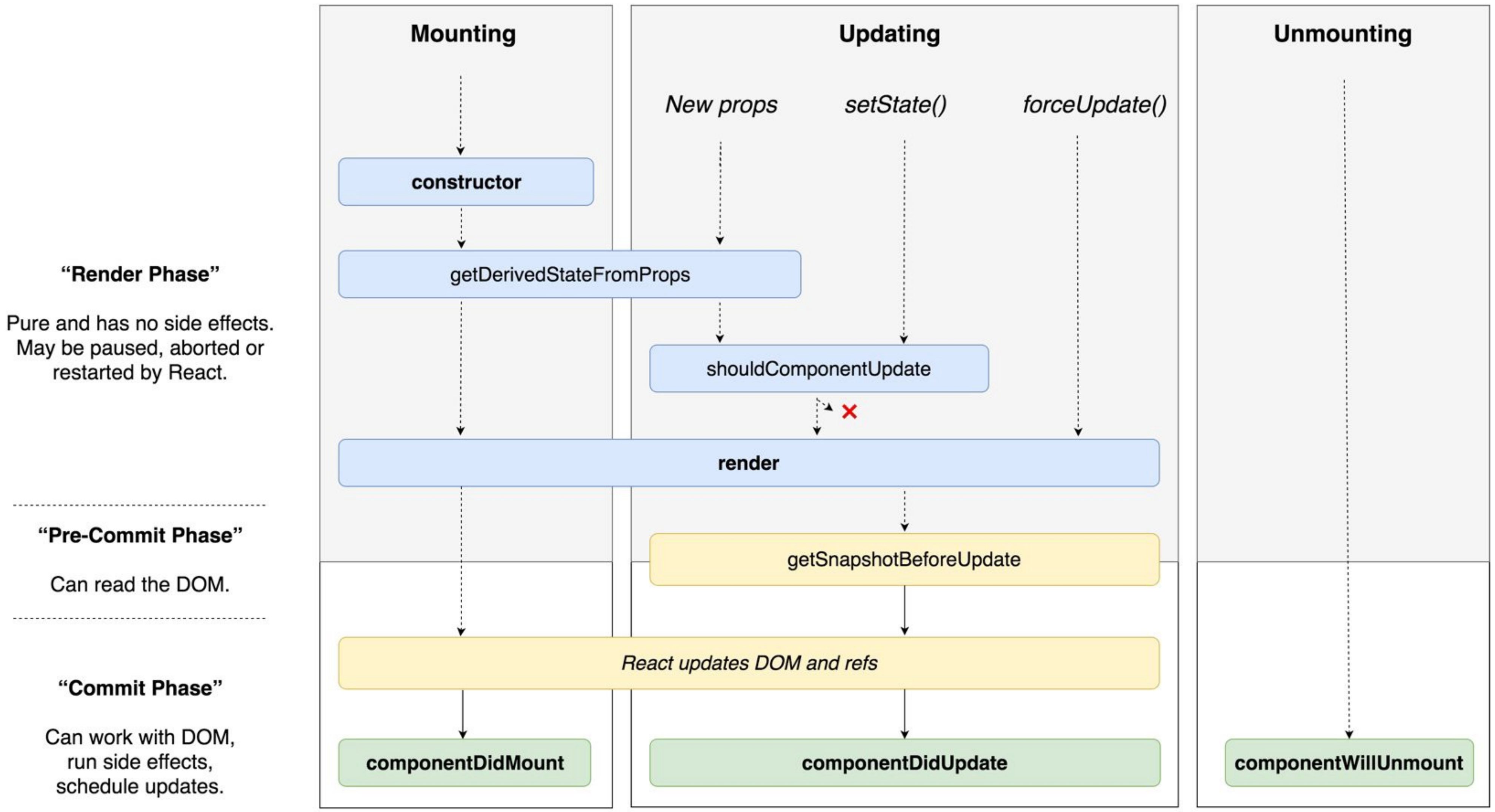
Mount/Unmount Life-Cycle Methods

- `constructor`
- `componentDidMount`
- `componentWillUnmount`

Data Life-Cycle Methods

- `getDerivedStateFromProps`
- `shouldComponentUpdate`
- `getSnapshotBeforeUpdate`
- `componentDidUpdate`

Component Life Cycle

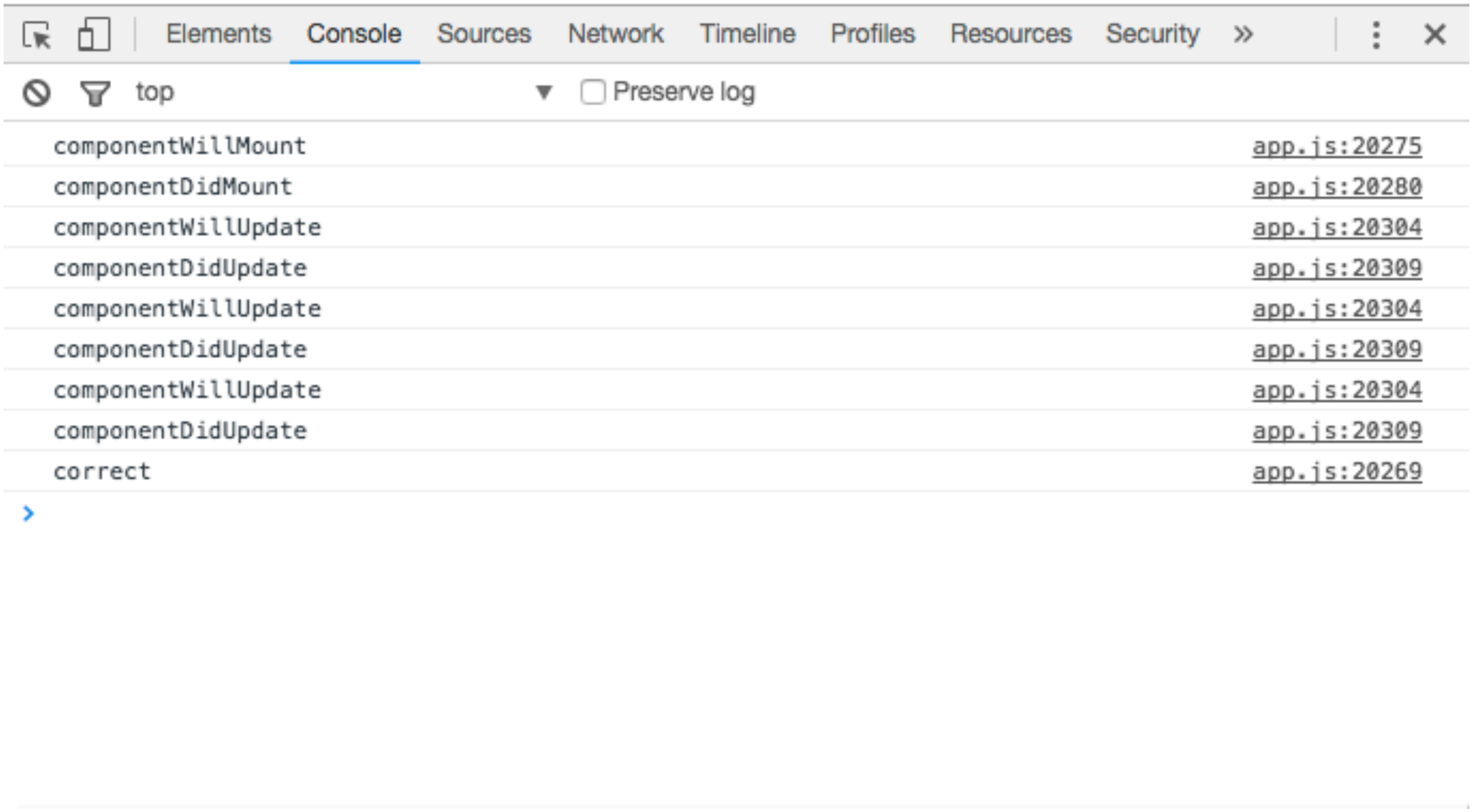


Events

- SyntheticEvent
  - A cross-browser wrapper around the browser's native event
  - Same interface as browser's native event

```
class ShareButton extends Component ({
  onButtonClick (evt) {
    alert("wow!");
  }
  render () {
    return (
    <div onClick={this.onButtonClick}>Share!</div>
    );
  }
});
```

Lab 22: Life Cycle and Events



```
componentWillMount                                   app.js:20275
componentDidMount                                    app.js:20280
componentWillUpdate                                  app.js:20304
componentDidUpdate                                   app.js:20309
componentWillUpdate                                  app.js:20304
componentDidUpdate                                   app.js:20309
componentWillUpdate                                  app.js:20304
componentDidUpdate                                   app.js:20309
correct                                              app.js:20269
>
```

Higher Order Functions

- A **function** that can take another **function** as an argument and/or that returns a **function** as a result.

```
const multiplyBy = (multiplier) => (number) => number *
multiplier

const double = multiplyBy(2); // returns (number) => number * 2

double(10) // returns 20
```

Higher Order Components

- A **function** that takes (wraps) a component and returns a new component.
- Allow us to abstract over actions, not just values.

```
const EnhancedComponent =
higherOrderComponent(WrappedComponent);
```

Context API

- Context allows parents to pass data implicitly to children, no matter how deep the component tree is.

Using Context with Provider

```
const ColorContext =
React.createContext('color');
class ColorProvider extends
React.Component {
    render(){
        return (
        <ColorContext.Provider
             value={'red'}>

         { this.props.children }

         </ColorContext.Provider>
        )
    }
}
```

```
class Parent extends React.Component
{
    render(){
        return (
            <ColorProvider>
                <Child />
            </ColorProvider>
        );
    }
}
```

Render Props

- Allows you to use props to share code between two components.

```
class App extends React.Component {
   getChildContext() {
      return {
         color: 'red'
      }
   }
   render() {
      return <Button />
   }
}
App.childContextTypes = {
   color: React.PropTypes.string
}
```

```
// Hook 'Color' into 'App' context
class Color extends React.Component {
   render() {
      return this.props.render(this.context.color);
   }
}

Color.contextTypes = {
   color: React.PropTypes.string
}

class Button extends React.Component {
   render() {
      return (
         <button type="button">
            {/* Return colored text within Button */}
            <Color render={ color => (
               <Text color={color} text="Button Text" />
            ) } />
         </button>
      )
   }
}

class Text extends React.Component {
   render(){
      return (
         <span style={{color: this.props.color}}>
            {this.props.text}
         </span>
      )
   }
}

Text.propTypes = {
   text: React.PropTypes.string,
   color: React.PropTypes.string,
}
```

Composition

- Composition is combining smaller components to form a larger whole.

Reusable Components

- Break down the common design elements (buttons, form fields, layout components, etc.) into reusable components with well-defined interfaces.
- The next time you need to build some UI, you can write much less code.
- This means faster development time, fewer bugs, and fewer bytes down the wire.

Presentational Components

- Components that just output presentation
- Contain no logic

Container Components

- Wrap presentational components
- Contain the logic and state

PropTypes

- Allows you to add type to props
- Useful for debugging, documentation
- Types:
  - `array`
  - `object`
  - `string`
  - `number`
  - `bool (not boolean)`
  - `func (not function)`
  - `node`
  - `element`

Using PropTypes

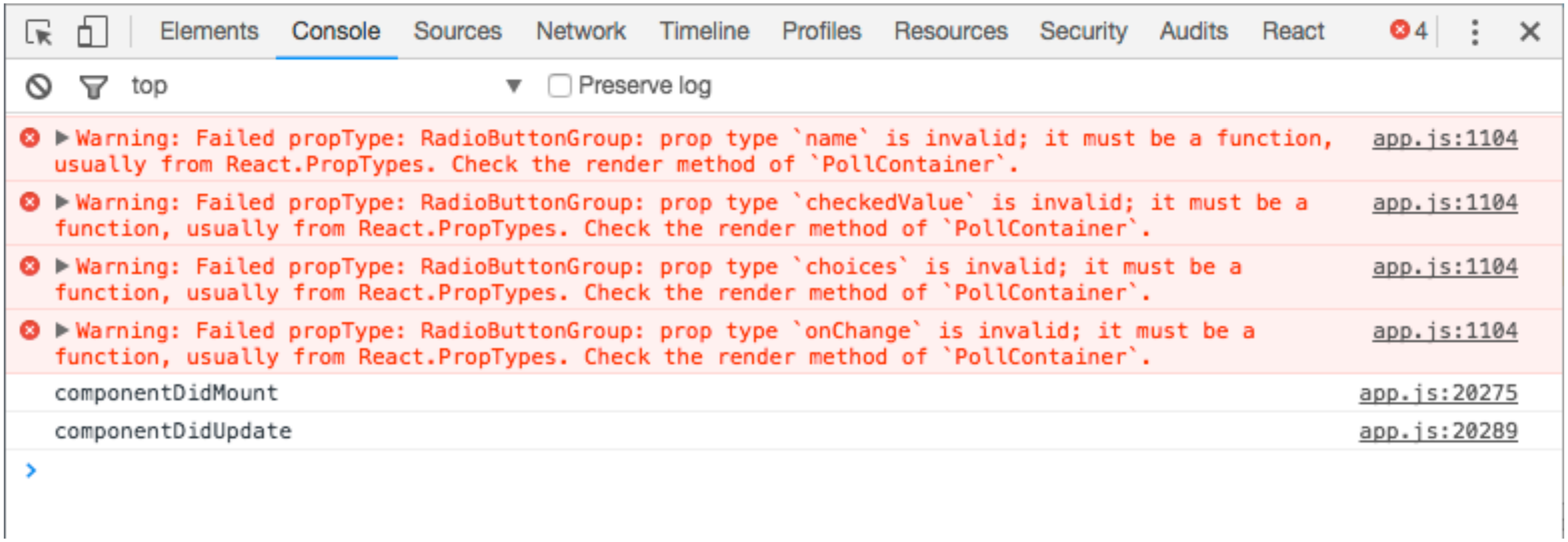
```
import React from 'react';
import PropTypes from 'prop-types';

class Component extends React.Component {
...
}

Component.propTypes = {
    name: PropTypes.string.isRequired,
    size: PropTypes.number.isRequired,
    color: PropTypes.string.isRequired,
    style: PropTypes.object
};
```

defaultProps

- Is part of React
  - not a separate library
- Default props will be used unless the parent overrides them.

```
class Component extends React.Component {
...
}
Component.defaultProps = {
  name: "Default Name",
  style: {backgroudColor: "blue"}
}
```

Lab 23: PropTypes



```
▶ Warning: Failed propType: RadioButtonGroup: prop type `name` is invalid; it must be a function,    app.js:1104
usually from React.PropTypes. Check the render method of `PollContainer`.
▶ Warning: Failed propType: RadioButtonGroup: prop type `checkedValue` is invalid; it must be a      app.js:1104
function, usually from React.PropTypes. Check the render method of `PollContainer`.
▶ Warning: Failed propType: RadioButtonGroup: prop type `choices` is invalid; it must be a           app.js:1104
function, usually from React.PropTypes. Check the render method of `PollContainer`.
▶ Warning: Failed propType: RadioButtonGroup: prop type `onChange` is invalid; it must be a          app.js:1104
function, usually from React.PropTypes. Check the render method of `PollContainer`.
componentDidMount                                                                                   app.js:20275
componentDidUpdate                                                                                  app.js:20289
>
```

Testing React Components

Objectives

- Learn about different rendering modes
- Learn about Jest
- Write Unit Tests with Jest

What to Test in a React Component

- Does it render?
- Does it render correctly?
- Test every possible state / condition
- Test the events
- Test the edge cases

Jest

- Facebook's Testing Framework
- Runs any tests in __tests__ directories, or named .spec.js, or named .test.js
- Simulates browser environment with jsdom
- Vastly improved and simplified in recent versions

Mocking

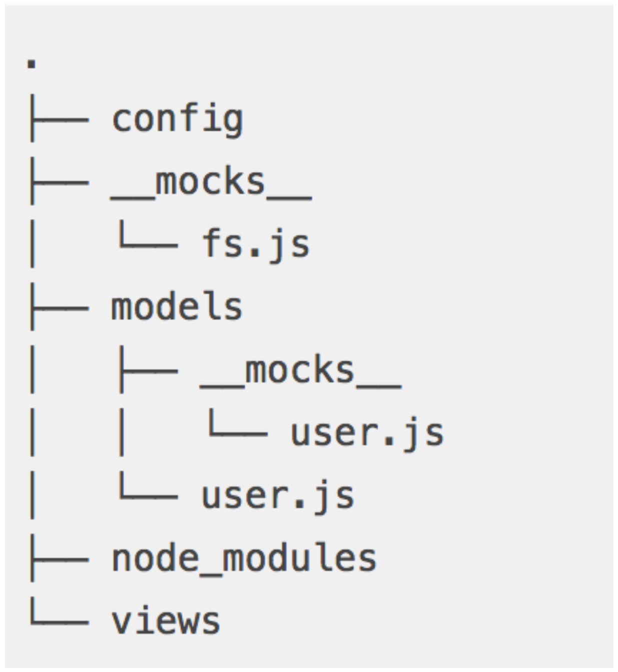
- Mock Function - erase the implementation of a function
- Manual Mocking - stub out functionality with mock data
- Timer Mocking - swap out native timer functions

Mock Function

```
const mockCallback = jest.fn();
forEach([0, 1], mockCallback);

// The mock function is called twice
expect(mockCallback.mock.calls.length).toBe(2);

// The first argument of the first call to the function was 0
expect(mockCallback.mock.calls[0][0]).toBe(0);

// The first argument of the second call to the function was 1
expect(mockCallback.mock.calls[1][0]).toBe(1);
```

Manual Mock

- Ensures tests will be fast and reliable by mocking external data and core modules.
- Define in a __mocks__ subdirectory adjacent to the module



```
.
├── config
├── __mocks__
│   └── fs.js
├── models
│   ├── __mocks__
│   │   └── user.js
│   └── user.js
├── node_modules
└── views
```

Manual Mocks (cont)

- Recommended practice is to create an automatic mock and then override it.
- See https://github.com/facebook/jest/tree/master/examples/manual_mocks for examples of manual mocks and tests that use them.

Automocking

- Disabled by default, but can be used on a per-module basis
  - `jest.mock(moduleName)`
- Enable automock by default in jest.config
  - `automock:true`

Snapshot Testing

1. Renders a components
2. Creates a "snapshot file" on first run
3. Compares subsequent runs with first and fails test if different.

Sample Snapshot Test

```
import React from 'react';
import Link from '../Link.react';
import renderer from 'react-test-renderer';

it('renders correctly', () => {

const tree = renderer.create( <Link
   page="http://www.facebook.com">Facebook</Link>
   ).toJSON();

expect(tree).toMatchSnapshot(); });
```

TestUtils

- `renderIntoDocument()`
- `mockComponent()`
- `isElement()`
- `isElementOfType()`
- `isDomComponent()`
- `isCompositeComponent()`
- `isCompositeComponentWithType()`
- `findAllInRenderedTree()`
- `more...`
- **NOTE**: `TestUtils has been moved into react-dom as of v15.5.0 (April 2017)`

TestUtils Example

```
Old (pre-15.5.0):
import TestUtils from 'react-addons-test-utils';

New (15.5.0):
import TestUtils from 'react-dom/test-utils';
```

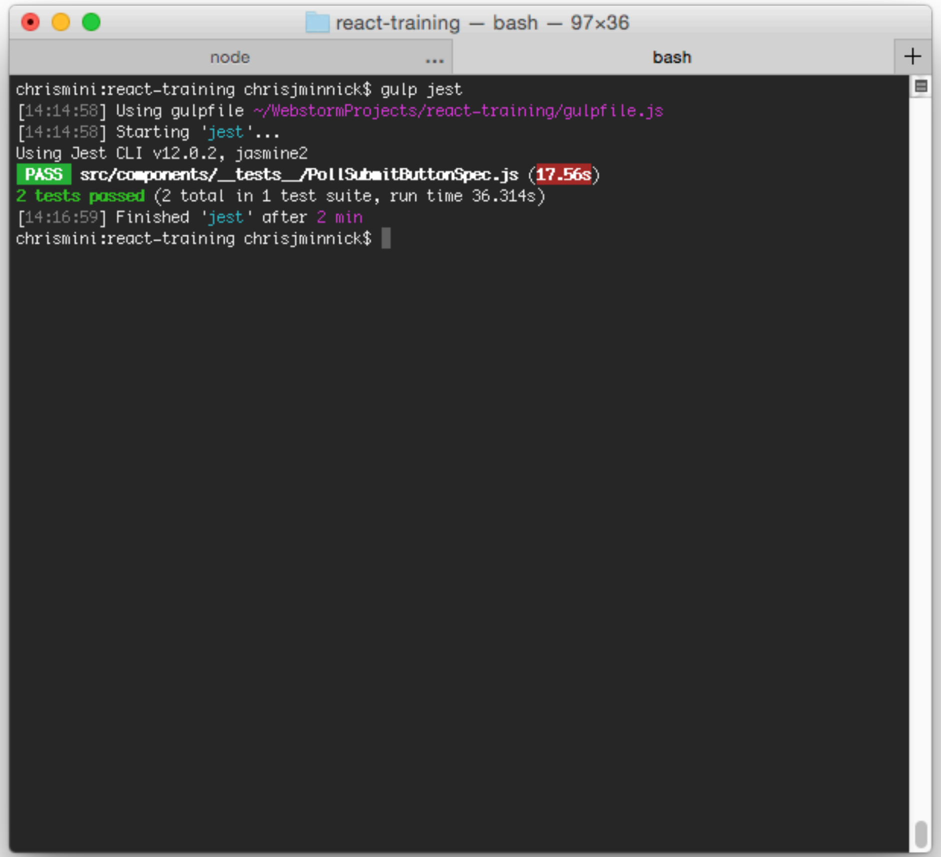
Enzyme

- Testing utility for React, created by AirBnB
- 3 render modes
  - shallow
    - render just the component
  - mount (Full Rendering)
    - For testing where you have components that require the full lifecycle in order to test.
    - Need to run in a browser environment.
  - render
    - renders react components to static HTML
- To use, pass a React component into a mode method.
  - `const wrapper = shallow(<PollSubmitButton />);`

Shallow Rendering

- Renders just a single component, regardless of where it is in the hierarchy.
- Allows you to isolate components for testing.
  - Ensure that your tests are indirectly asserting on behavior of child components.

Lab 24: Testing React Components



```
chrismini:react-training chrisjminnick$ gulp jest
[14:14:58] Using gulpfile ~/WebstormProjects/react-training/gulpfile.js
[14:14:58] Starting 'jest'...
Using Jest CLI v12.0.2, jasmine2
 PASS  src/components/__tests__/PollSubmitButtonSpec.js (17.56s)
2 tests passed (2 total in 1 test suite, run time 36.314s)
[14:16:59] Finished 'jest' after 2 min
chrismini:react-training chrisjminnick$ 
```

Lab 24.5: Testing with Jest and Enzyme

```
npm install --save-dev enzyme enzyme-adapter-react-16
```

- Put this at the top of your test files

```
import { shallow, mount, render } from 'enzyme';
```

- Create src/setupTests.js with the following code:

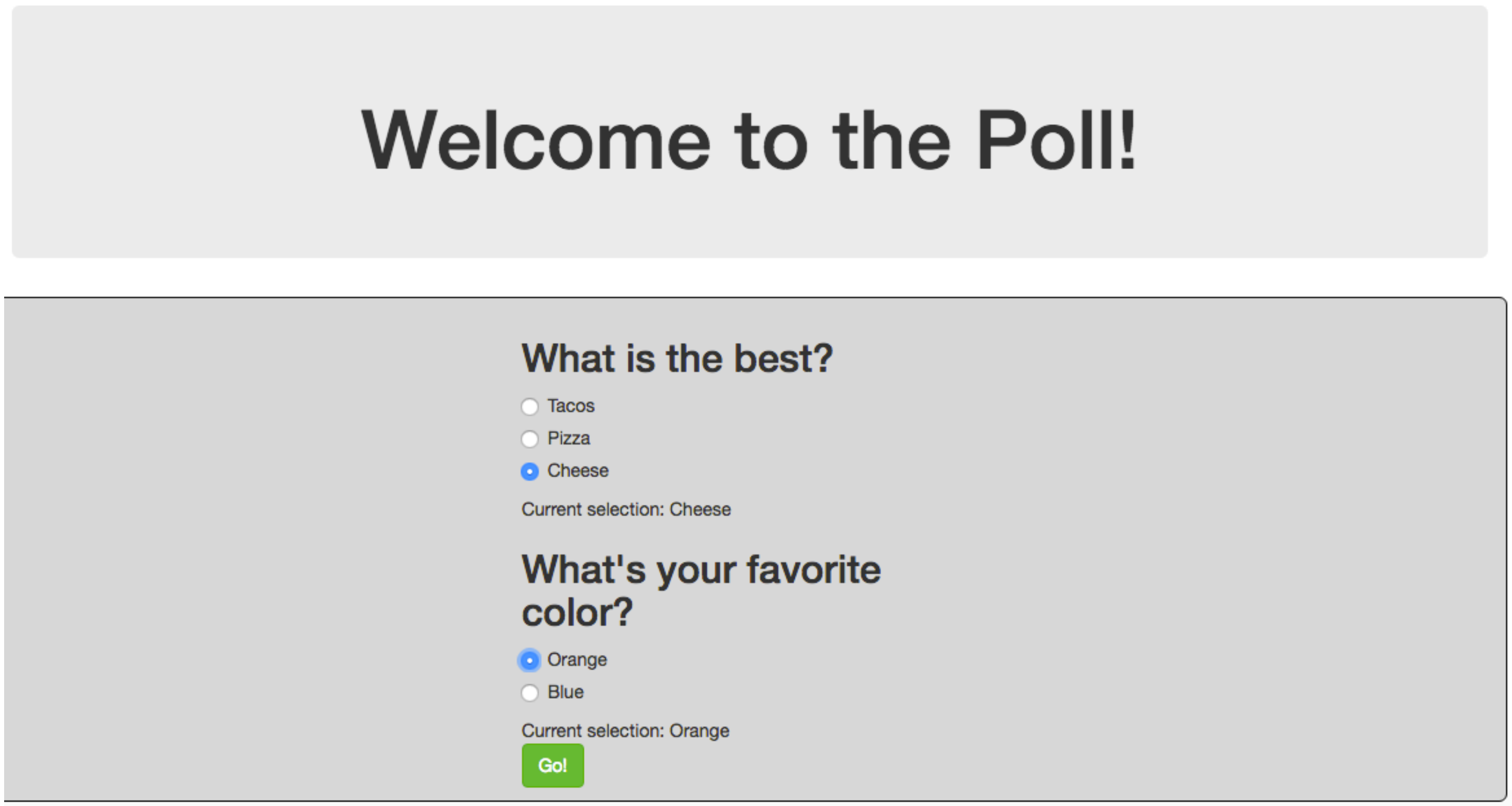
```
import { configure } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
configure({ adapter: new Adapter() });
```

- Add this to package.json

```
"jest": {
  "setupFilesAfterEnv": [
    "<rootDir>/src/setupTests.js"
  ]
}
```

Lab 25: Multiple Components

- Display a list of questions and track the state of each group of radio buttons separately.



Welcome to the Poll!

What is the best?

Tacos
Pizza
[✓ Cheese]

Current selection: Cheese

What's your favorite color?

[✓ Orange]
Blue

Current selection: Orange

React Router

- Declarative way to do routing
- Maps components to URLs
- Dynamic routing (as of v4)
  - Routing takes place as the component is rendering
  - Not in a configuration
  - Almost everything in React Router is a component

Using React Router

- Import the version of React Router for your target environment (i.e. DOM or Native), plus other components

```
import { BrowserRouter, Route, Link } from 'react-router-dom'
```

- Render the Router

```
ReactDOM.render((
<BrowserRouter> <App/> </BrowserRouter> ), holder)
```

- Use <Link> to link to a new location (in `<App>` in this case)

```
<Link to="/dashboard">Dashboard</Link>
```

- Render a Route (also in `<App>`)

```
<Route path="/dashboard" component={Dashboard}/>
```

Router Rendering Example

```
index.js

import { BrowserRouter } from 'react-router-dom';

ReactDOM.render((
  <BrowserRouter>
    <App/>
  </BrowserRouter>
), holder)
```

- `Use BrowserRouter when you have a server`
- `Use HashRouter if you're using a static file server.`

Route Matching

- Route
  - Compares the value of the path prop to the current location's pathname.
  - Renders the component specified by the component prop.
  - Can be used anywhere you want to render based on location.
- Switch
  - Can be (optionally) used for grouping Routes. Will iterate through a group and stop when a match is found.

Navigation

- <Link>
  - Creates links in your application.
  - Inserts an <a> in your HTML
- <NavLink>
  - Can be styled as "active" when it matches the current location.
- <Redirect>
  - Forces navigation

Lab 26: React Router 3.x

- Use React Router to change the UI based on the URL

Lab 27: React Router 4.x

- Use React Router to change the UI based on the URL

Chapter 12: Flux and Redux

Objectives

- Understand the Flux pattern
- Explain Redux's architecture
- Create Redux actions
- Write pure functions
- Use Reducers
- Use Redux with AJAX

Flux

- Flux isn't a library or module.
- It's a design pattern.
- npm install flux installs Facebook's dispatcher.
- It's possible to use Flux design principles without Facebook's module.

Flux Flow

1. Some sort of interaction happens in the view.
2. This creates an action, which the dispatcher dispatches.
3. Stores react to the dispatched action if they're interested, updating their internal state.
4. Stateful view component(s) hear the change event of stores they're listening to.
5. Stateful view component(s) ask the stores for new data, calling setState with the new data.

Flux Action

- An action in flux is what's made when something happens.
- In other words, when you click on something, that's not an action
  - it creates an action. Your click is an interaction.
- Actions should have (but aren't required to have) a type and a payload. Most of the time they will have both, occasionally they'll just have a type.

Flux Dispatcher

- Broadcasts actions when they happen and it lets things tune in to those broadcasts
- Instead of an onClick function using a callback passed to it to set the state of your application, you have it (onClick) use the dispatcher to dispatch a specific action for anyone who's interested to listen for.

Flux Stores

- Represents the ideal state of your application
- If a user enters something into a form, it dispatches an action. If the store is listening for this action, it will update its internal state accordingly.
- Stores don't contain any public setters, just public getters. The only ones who can change the data in a store is the store itself when it hears an action from the dispatcher that it's interested in.

EventEmitter

- Stores emit change events that don't contain data.
- If the view is listening for the particular store's change event, the view should ask the store for the new data that will bring the view back into sync, call setState, and re-render.

Redux

- An implementation of Flux
- Stores state of the app in an object tree in a single store
- The state tree can only be changed by emitting an action
- Specify how the actions transform the state tree using pure reducers

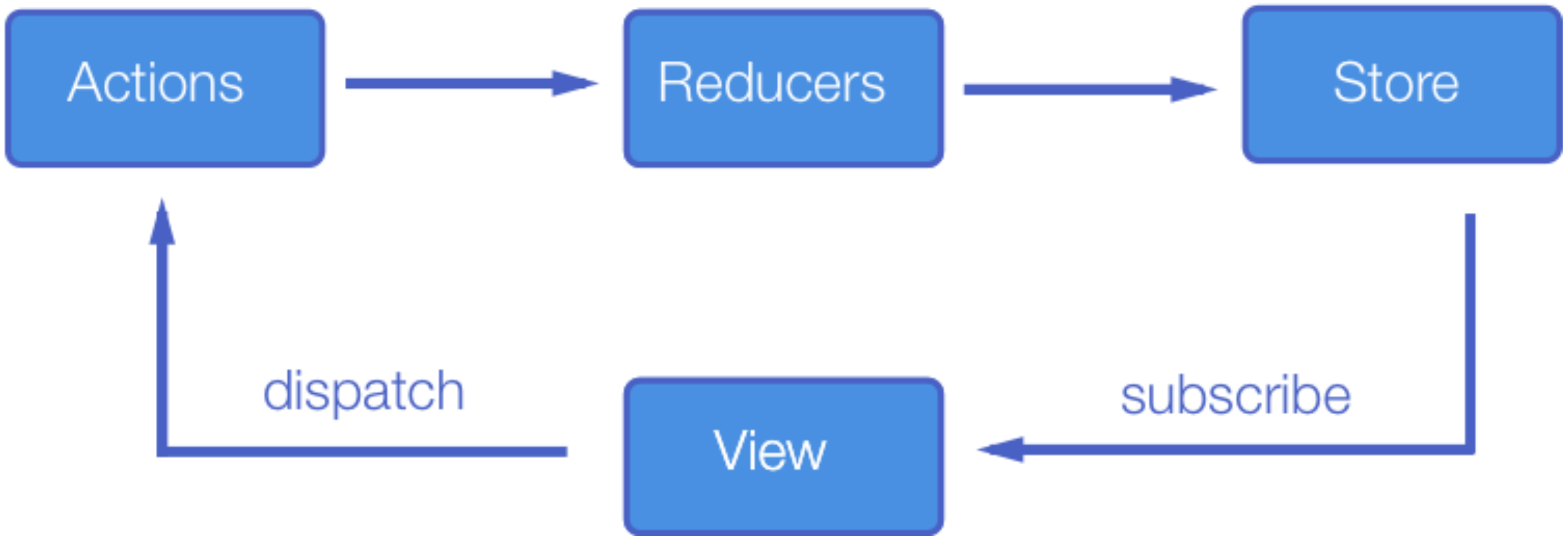
Stores & Immutable State Tree

- The biggest difference between Flux and Redux is that Redux has a single store.
- Redux has a single store with a single root reducing function.
- Split the root reducer into smaller reducers to grow the app.
- Trace mutations by replaying actions that cause them.

Redux Actions

- Payloads of information that send data from the application to the store.
- Actions are the only way to mutate the internal state.
- Use `store.dispatch()` to send them to the store.

```
const ADD_TODO = 'ADD_TODO'
{
  type: ADD_TODO,
  text: 'Build my first Redux app'
}
```


Actions
Reducers
Store
dispatch
View
subscribe

Reducers

- Specifies how the application's state changes in response to something happening (an action).
- A pure function that takes the previous state and an action, and returns the next state.
- `(previousState, action) => newState`

Things You Should Never do in a Reducer

- Mutate its arguments
- Perform side effects like API calls and routing transitions
- Call non-pure functions

Reducer Composition

- The fundamental pattern of building Redux apps
- Split up reducer functions using child reducers.

- Workflow
  - Write top-level reducer to handle a particular function.
  - Break up the top-level reducer into a master reducer that calls smaller reducers to separate concerns.

Reducer Composition Example

```
function checkedValue(state = [], action) {
  switch (action.type) {
      case ‘SELECT_ANSWER’:
        return state
          .slice(0,action.index)
          .concat([action.value])
          .concat(state.slide(action.index+1));
        default:
           return state;
    }
}
export default checkedValue;

function question(state = [], action) {
  return state;
}
export default questions;
```

Reducer Composition Example (cont)

Combine reducers

```
const rootReducer = combineReducers({
  questions,
  checkedValue
});

Export default rootReducer;
```

Redux Store

- Holds the application state
- Allows access to state via `getState()`
- Allows state to be updated via `dispatch(action)`
- Registers listeners via `subscribe(listener)`
- Handles unregistering of listeners via the function returned by `subscribe(listener)`

Redux Store Design

- Many applications use data that is relational or "nested"

```
let orders = [
  {
    id:1,
    customer:{id:20,name:"Wilma",address:""},
    items:[{id:10,name:"Toaster",price:20}
  }
]
```

- Relational data mapped to objects can be complex and cause repetition.

Redux Store Design

- Solution: Normalize part of your store, and treat it like a database.

- **Data normalization:**
- Each type of data gets its own "table" in the state.
- Each "data table" should store the individual items in an object, with the IDs of the items as keys and the items themselves as the values.
- Any references to individual items should be done by storing the item's ID.
- Arrays of IDs should be used to indicate ordering.

Normalized Redux Store Example

```
{
  orders : {
    byId : {
      "order1" : {id : "order1", customer : "customer1",
                  items: ["item1","item2"]},
      "order2" : {id : "order2", customer : "customer1",
                  items: ["item1","item6"]}
    },
    allIds : ["order1", "order2"] },
  customers : {
    byId : {
      "customer1" : { ... }
      ...
}
```

Benefits of Normalizing Store

- Each item is defined in only one place
- Simplifies Reducer logic (flatter structure)
- Simplifies logic for retrieving and updating items
- Because each data type is separated, updates to a single type of data require fewer components to be re-rendered

Redux Pros and Cons

Redux Pros

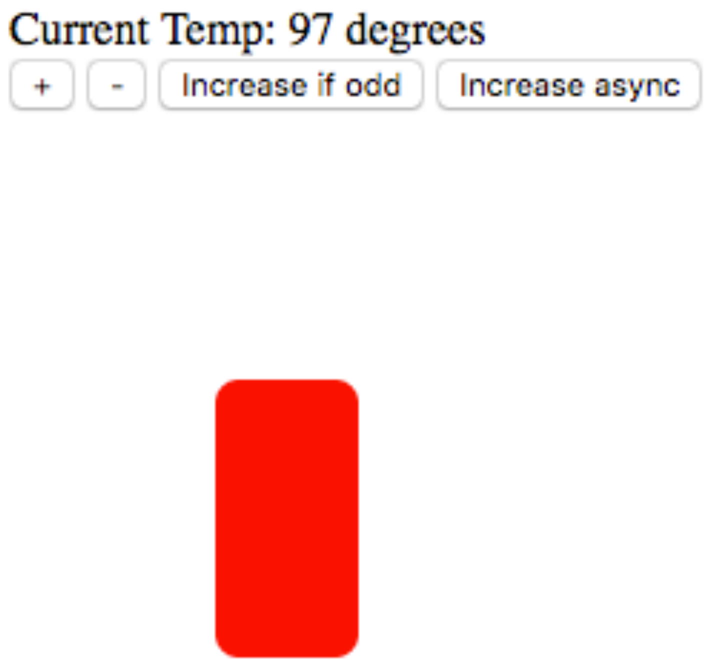
- Declarative
- Immutable state
- Mutation logic separate from views
- Great for testing

Redux Cons

- More complicated than just using plain react components

Lab 28: Redux Thermometer

- Starting with the Redux counter example app (which is in the 'counter' folder inside the Redux examples that come with Redux), convert it into a thermometer / thermostat app with a graphical output.


Current Temp: 97 degrees

Lab 29: Implementing Redux

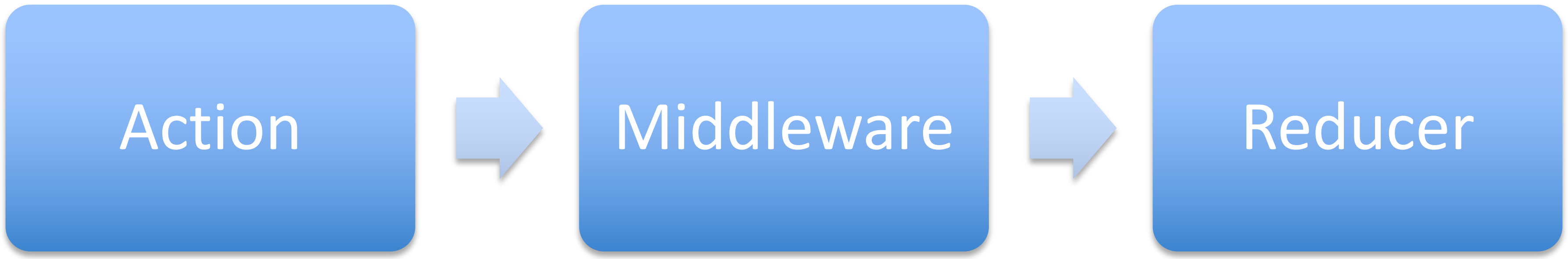
- Implement Redux to change the state of the radio buttons in the Poll Application.

React AJAX Best Practices

- Four Ways
  - Root Component
    - Best for small apps and prototypes.
  - Container Components
    - Create a container component for every presentational component that needs data from the server.
  - Redux Middleware
    - Thunk Middleware
    - Saga
  - Relay
    - Good for large applications
    - Declare data needs of components with GraphQL
    - Relay automatically downloads data and fills out the props

What is Redux Middleware?

- a third-party extension point between dispatching an action, and the moment it reaches the reducer.



What is Middleware Good For?

- Logging actions
- Reporting errors
- Dispatching new actions
- Asynchronous requests

Using Redux Form

- Higher order form for simplifying using forms with Redux.
- Step 1:

```
import { createStore, combineReducers } from 'redux'
import { reducer as formReducer } from 'redux-form'

const rootReducer = combineReducers({
// ...your other reducers here
// you have to pass formReducer under 'form' key,
// for custom keys look up the docs for 'getFormState'
form: formReducer }
)
```

Redux Form, Step 2

```
import React from 'react'
import { Field, reduxForm } from 'redux-form'
let ContactForm = props => {
const { handleSubmit } = props
return <form onSubmit={handleSubmit}>
{/* form body*/}</form>
}
ContactForm = reduxForm({
// a unique name for the form
form: 'contact' })(ContactForm)

export default ContactForm
```

Redux Form, Step 3

- Add <Field /> Components

```
let ContactForm = props => {
const { handleSubmit } = props
   return (
   <form onSubmit={handleSubmit}>
   <div>
   <label htmlFor="firstName">First Name</label>
   <Field name="firstName" component="input"
   type="text" />
   </div>
   <button type="submit">Submit</button>
   </form> )
}
```

Redux Form, Step 4

- React to Submit

```
import React from 'react'
import ContactForm from './ContactForm'
class ContactPage extends React.Component {
   submit = values => {
   // print the form values to the console
   console.log(values) }
   render() {
      return <ContactForm
         onSubmit={this.submit} />
}
}
```

Thunk

- Allows you to write action creators that return a function instead of an object.
- Can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met.
- When an action creator returns a function, that inner function will get executed by the Thunk middleware.
- The inner function receives the store methods `dispatch` and `getState()` as parameters.
- On success, the Thunk action calls a standard action.

Thunk Example

```
//store.js

import { createStore, applyMiddleware, compose } from "redux";
import thunkMiddleware from "redux-thunk";
import rootReducer from "../reducers";

const createStoreWithMiddleware =
    compose( applyMiddleware(thunkMiddleware) )(createStore);

export default function configureStore(initialState) {
    const store = createStoreWithMiddleware(rootReducer);
    return store;
}
```

Thunk Example (cont)

```
//actionCreators.js

export function receiveQuestions(data) {
    return {
        type: 'RECEIVE_QUESTIONS',
        questions: data.poll.questions,
    };
}

export function fetchQuestions() {
    return dispatch => {
        fetch('data/data.json')
            .then(response => {
                const data = response.json();
                dispatch(receiveQuestions(data));
            })
            .catch(error => dispatch({
                type: 'FETCH_FAILED', error
            })
        );
    };
}
```

Redux Saga

- Useful for complex async operations.
- Uses generator functions to complete actions in series.
- redux-saga.js.org/docs/introduction/BeginnerTutorial.html

Using Sagas

```
//install redux-saga
npm install --save redux-saga

//dispatch an action
class UserComponent extends React.Component {
...
onSomeButtonClicked() {
   const { userId, dispatch } = this.props;
   dispatch({
      type: 'USER_FETCH_REQUESTED',
      payload: {userId}});
   }
... }
```

Using Sagas (cont)

```
// create a "worker" saga to do the async action

import { call, put, takeEvery, takeLatest } from 'redux-saga/effects';
import Api from '...';

// worker Saga: will be fired on USER_FETCH_REQUESTED actions

function* fetchUser(action) {
  try {
    const user = yield call(Api.fetchUser, 
action.payload.userId);
    yield put({type: "USER_FETCH_SUCCEEDED", user: user});
} catch (e) {
    yield put({type: "USER_FETCH_FAILED", message: 
e.message});
}
}
```

Using Saga (cont)

```
// create a listener saga to listen for the event

function* mySaga() {
   yield takeEvery("USER_FETCH_REQUESTED", fetchUser);
}

/* Alternatively you may use takeLatest.

function* mySaga() {
   yield takeLatest("USER_FETCH_REQUESTED", fetchUser);
}
*/
// export mySaga
export default mySaga;
```

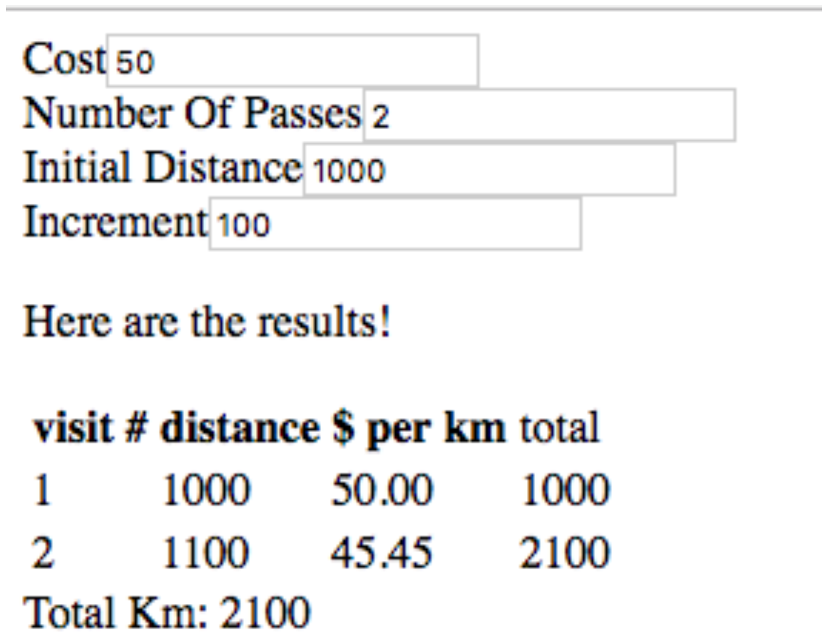
Using Saga (cont)

```
// Connect Saga to the store, using middleware
import { createStore, applyMiddleware } from 'redux';
import createSagaMiddleware from 'redux-saga';
import reducer from './reducers';
import mySaga from './sagas';

// create the saga middleware
const sagaMiddleware = createSagaMiddleware();
// mount it on the Store
const store = createStore( reducer,
applyMiddleware(sagaMiddleware) );
// then run the saga
sagaMiddleware.run(mySaga);
// render the application
```

Demo: Authentication with React and JWT

1. git clone https://github.com/watzthisco/react-jwt-authentication-example
2. cd react-jwt-authentication-example
3. npm install
4. npm start

Lab 30: SwimCalc App



Cost 50
Number Of Passes 2
Initial Distance 1000
Increment 100

Here are the results!

visit #	**distance**	**$ per km**	total
1	1000	50.00	1000
2	1100	45.45	2100

Total Km: 2100

Relay and GraphQL

Objectives

- Retrieve data with Relay

What is Relay?

- Framework for building data-driven react applications
- Simpler than Flux, but may also be used with Flux
- Use GraphQL as a query language.

GraphQL

- Data query language and runtime
- Declarative
- Compositional
- Strongly Typed

GraphQL Example

Query

```
{
  user(id: 3500401) {
    id,
    name,
    isViewerFriend,
    profilePicture(size:
50)  {
      uri,
      width,
      height
    }
  }
}
```

Response

```
{
  "user" : {
    "id": 3500401,
    "name": "Jing Chen",
    "isViewerFriend": true,
    "profilePicture": {
      "uri":
"http://someurl.cdn/pic.jpg"
,
      "width": 50,
      "height": 50
    }
  }
}
```

Relay Pros and Cons

Relay Pros

- Even more declarative
- No custom getter logic
- Tight server integration

Relay Cons

- Requires GraphQL server
- Much more complexity
- Less flexible

Chapter 13: Advanced Topics

Objectives

- Server-side React
- Using React with Other Frameworks / Libraries
- Performance Issues

ACCESSIBILITY AND REACT

Semantic HTML

- HTML tags reinforce meaning of the data
  - <header>, <nav>, <main>, <form>, <footer>, etc.
  - <div> doesn't convey meaning, but is sometimes necessary for hooking scripts and styles into.
  - React.Fragment element can be used to group components without adding extra elements

```
import React, {Fragment} from react;
...
function ListItem({ item }) {
  return (
    <Fragment>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </Fragment>
); }
```

Accessible Forms

- Every form control should have a <label>

```
<label htmlFor="namedInput">Name:</label> <input
id="namedInput" type="text" name="name"/>
```

- Notify users of errors
- If form focus is interrupted (such as by a modal), use Refs to set focus programatically
- Insure that all functionality can be accessed using keyboard events

Testing Accessibility

- Use the ESLINT accessibility plugin
  - eslint-plugin-jsx-a11y
  - is installed and activated by default with create-react-app
- Use The Accessibility Engine to test accessibility in the browser
  - https://www.deque.com/axe/

Server-side React

- Allows you to pre-render components' initial state on the server
- Methods:
  - `renderToString`
    - Returns an HTML string
    - Send the markup down on the initial request for faster page loads and to allow search engines to crawl your pages for SEO purposes.
  - `renderToStaticMarkup`
    - Works the same as `renderToString`, but doesn't add in the extra React DOM elements
    - Good for using React as a static page generator

Using React with Other Libraries

- Use the lifecycle events to attach logic from other libraries.
- `componentDidMount`
  - Attach code that will run when the component is loaded.
  - Can be used similarly to jQuery's $(document).ready()
- `componentDidUpdate`
  - Attach code that will run when a component updates.

Performance Optimization

- Production bundle
- Perf object
- shouldComponentUpdate()

Render Caching

- A technique for making web pages load faster on subsequent visits.

1. Encapsulate load state so that no server call is required to render the initial view
2. Make all API calls before render()
3. Cache locally in the unload handler
4. Restore the last known state on load
5. Render the last known state in react using ReactDOM.hydrate

ReactDOM.hydrate

```
import {render, hydrate} from "react-dom"

if (window.hasRestoredState) {
  hydrate(<MyPage />, renderTarget);
  } else {
  render(<MyPage />, renderTarget);
}
```

Development vs. Production

- Development build
  - Uncompressed.
  - Displays additional warnings that are helpful for debugging.
  - Contains helpers not necessary in production
  - ~669kb
- Production build
  - Compressed.
  - Contains additional performance optimizations.
  - ~147kb

Perf Object

- Indicates expensive parts of your app.
- Only available in development mode.

Perf Object Methods

- Methods
  - `start()` and `stop()`
    - Start and stop measurement
    - Records operations in-between
  - `printInclusive(measurements)`
    - Prints the overall time taken
    - Defaults to the measurements from the last recording
    - Outputs a nice-looking table
  - `printExclusive(measurements)`
    - Doesn't include time taken to mount components, process props, etc.
  - `printWasted(measurements)`
    - "Wasted" time is spent on components that didn't render anything
  - `printOperations(measurements)`
    - Prints the underlying DOM manipulations

Optimization Techniques

- Use `shouldComponentUpdate` hook to add optimization hints to React's diff algorithm

```
shouldComponentUpdate: function() {
   // Let's just never update this component again.
   return false;
 },
```

- Use keys
  - Unique identifier created with `key` attribute.

reactjs.org/hooks

HOOKS

Why Use Hooks?

- Stateful classes need to be written with class syntax
- Classes can be confusing.
- Reusing logic is not possible

Stateful Components with Class

```
Import React from 'react';

export default class Hello extends React.Component {
      constructor(props) {
            super(props);
            this.state = {
                  name: 'World',
            }
            this.handleChange = this.handleChange.bind(this);

      }
      handleNameChange(e) {
            this.setstate({
                  name: e.target.value
            });
      }
      render() {
            return(
                  <input type="text"
                          value={this.state.name}
                          onChange = {this.handleChange} >
            );
      }
}
```

What are Hooks?

- Functions provided by React that let you hook into React features from function components.
- Must be at the top level of the component
  - not inside a condition
- Hook examples
  - useState – Hooks into state
  - useContext – Hooks into Context
  - useEffect – Hooks into lifecycle methods. Runs after initial render and after every update (by default).

useState Hook

```
import React, { useState } from 'react';

export default function Hello(props) {
   const [name, setName] = useState('World');

   function handleChange(e) {
      setName(e.target.value);
   }

   return(
      <input type="text"
             value={name}
             onChange={handleNameChange} />
   );
}
```

useContext Hook

```
import React, { useState, useContext } from 'react';
import { ThemeContext } from './context';

export default function Hello(props) {
    const theme = useContext(ThemeContext);
    const [name, setName] = useState('World');

    function handleChange(e) {
        setName(e.target.value);
    }
    return(
        <section className={theme}>
            <input type="text"
                    value={name}
                    onChange={handleChange}
            />
        </section>
    )
}
```

useEffect Hook

```
import React, { useState, useEffect } from 'react';

export default function Hello(props) {
    const [name, setName] = useState('World');

    useEffect(() = {
        document.title = 'Hello' + ' ' + name;
        return ()=>{
            // optionally return a function
            // to clean up after the effect,
            // such as by removingEventListener or timer set
            // in the effect
    });

...
```

Custom Hooks

- Start with 'use' by convention.
- Create a function outside of the component function.
- Use the function inside the component.

Custom Hook Example

```
export default function Hello(props) {
   const [name, setName] = useState('World');
   useDocumentTitle(name);
   ...
}

function useDocumentTitle(title){
   useEffect(() = {
   document.title = title;
   }
}
```

Should you convert everything to Hooks?

- Probably not right now (Q1 2019).
- Hooks are not yet released (as of 16.7).
- You can try out hooks now with the alpha version.

Using Pre-built Components

- Thousands of React components are shared via npm
  - Searchable through databases like React-Components.com.

Further Study

- Dan Abramov's 30 Free Redux Videos
- Wes Bos’s Redux Course (https://learnredux.com/)

Where to go for help?

- React Documentation
- Stackoverflow

Disclaimer and Copyright