

# Test-Driven Development Using React.js and ES6 (ES2015)

copyright 2016, Chris Minnick  
version 1.5, December 2016

# Contents

## **Chapter 0: Introduction - 4**

## **Chapter 1: Development Ecosystem - 16**

1. Node.js - 19
2. Git - 28
3. Command Prompt - 35

## **Chapter 2: Reproducible Builds - 37**

1. NPM - 40
2. Gulp - 46

## **Chapter 3: Static Code Analysis - 56**

1. Configuring JSHint - 59
2. Browser Development Tools - 65

## **Chapter 4: Introduction to TDD / BDD - 66**

1. Assertions - 68
2. Jasmine - 70
3. Karma - 82

## **Chapter 5: Modularity - 84**

1. CommonJS - 86
2. RequireJS - 87
3. ES6 Modules - 88

## **Chapter 6: Building and Refactoring - 92**

# Contents

## **Chapter 7: ES2015 (ES6) - 98**

## **Chapter 8: The Document Object Model (DOM) - 139**

## **Chapter 9: Introduction to React.js - 150**

1. Props vs. State - 156
2. Virtual DOM - 159
3. React Development Process - 168

## **Chapter 10: JSX - 175**

1. What is JSX? - 176
2. Using JSX - 177

## **Chapter 11: React Components - 186**

1. Styles in React - 200
2. Forms - 203
3. Testing React Components – 231
4. React Router - 236

## **Chapter 12: Flux and Redux - 237**

1. Flux - 238
2. Redux - 244

## **Chapter 13: Advanced Topics - 259**

1. Server-side React - 263
2. Using React with Other Libraries - 266
3. Optimizing React Performance - 267

# Introduction

## Objectives

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

# Chris Minnick

- Author
  - Coding JavaScript For Dummies
  - JavaScript For Kids
  - Beginning HTML5 and CSS3 For Dummies
  - Adventures in Coding
  - XHTML
  - WebKit For Dummies
- Web / Front-end Developer
  - Since 1996
- Trainer
  - Since 2008
- Swimmer, winemaker, musician

# 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?
- Favorite food?

# Daily Schedule

- 08:30 - 10:00
- 15 minute break
- 10:15 - 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
  - ES6
  - React.js



# Professional Front-End Development

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

# What is ES6?

- Now officially called ES2015
- First update to language since 2009
- New features include:
  - Arrow functions
  - Classes
  - Block-scoped binding constructs (let and const)
  - Iterators
  - Modules
  - Promises
- Supported in Node.js
- Beginning to be supported by browsers

# 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.

# Who Uses React?

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

**and many more!**

# Chapter 1:

# Development Ecosystem

Objectives:

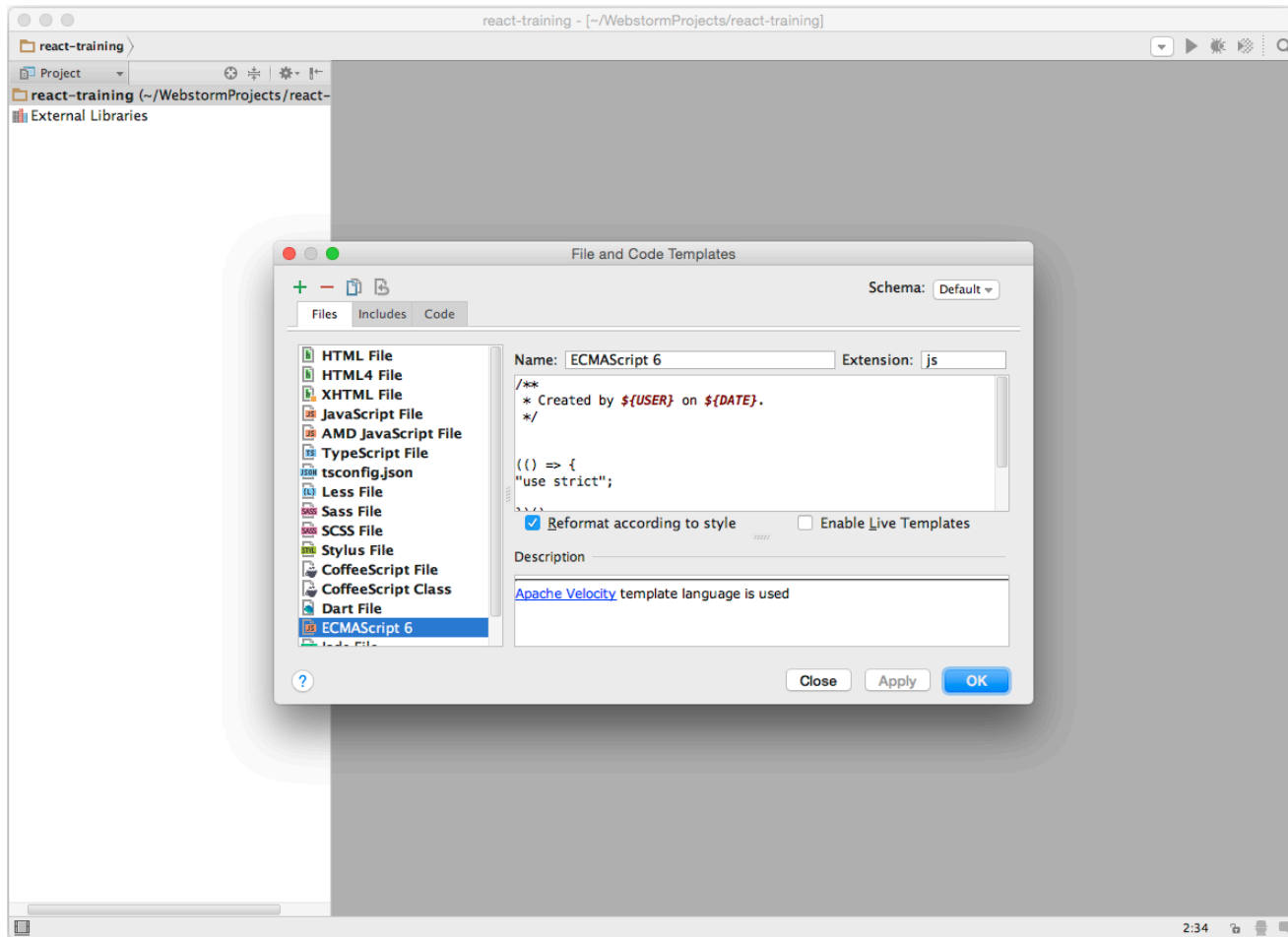
- Configure your IDE
- Use Node.js as a development tool
- Use NPM to install and manage packages
- Use Git for version control
- Use command line dev tools

# Code Editors and IDEs

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



# Lab 1: Installing and Configuring WebStorm IDE



# 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

# 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 occurred");  
});
```

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

# Node Streams

- Streams are unix pipes that let you read data from one source and pipe it to a destination.
- Just an EventEmitter with some additional methods.
- Types of Streams
  - Readable
    - Lets you read data from a source.
  - Writable
    - Lets you write data to a destination.

# Node Modules

- 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).
- `require` imports anything that you wish to use in your code.
- `module.exports` makes modules available to be required.

# 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;  
}
```



# Using Readable Streams

```
var fs = require('fs');  
var readableStream =  
  fs.createReadStream('file.txt');  
var data = '';  
  
readableStream.on('data', function(chunk) {  
  data+=chunk;  
});  
  
readableStream.on('end', function() {  
  console.log(data);  
});
```

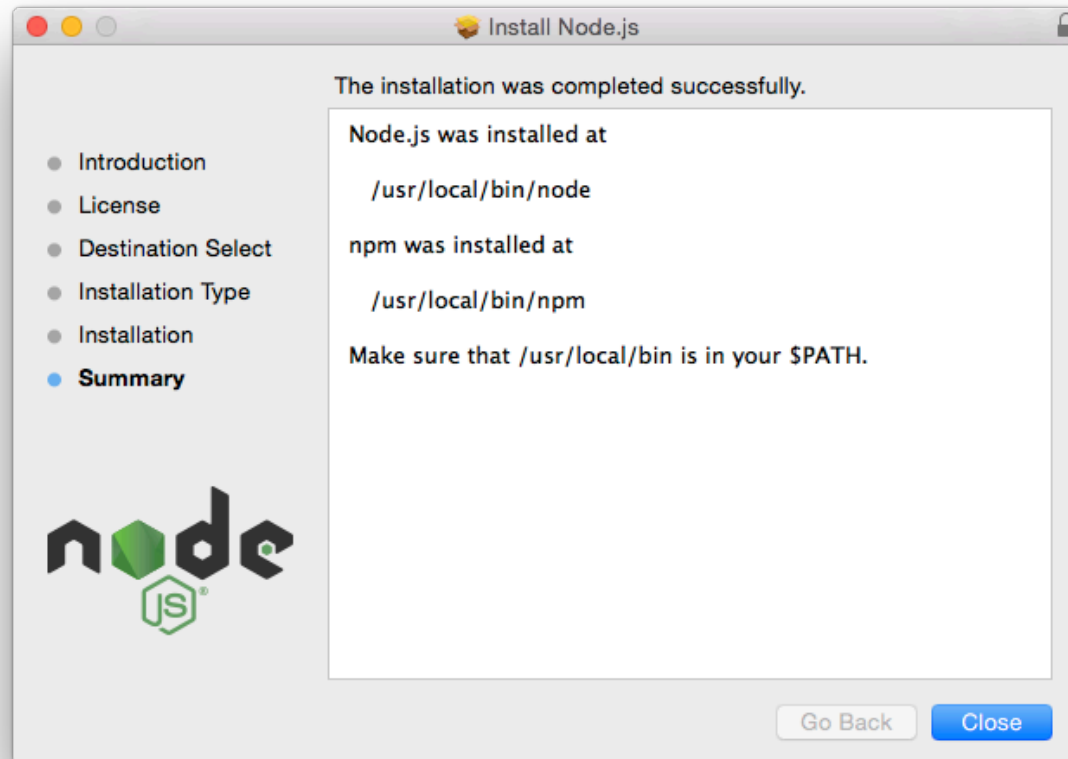
# Using Writable Streams

```
var fs = require('fs');  
var readableStream =  
  fs.createReadStream('file1.txt');  
var writableStream =  
  fs.createWriteStream('file2.txt');  
  
readableStream.setEncoding('utf8');  
  
readableStream.on('data', function(chunk) {  
  writableStream.write(chunk);  
});
```

# 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



# 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:
  - Committed
    - Data stored in your local repo
  - Modified
    - Data changed but not committed
  - Staged
    - File is marked to go into your next snapshot

# 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
- Create tasks with Gulp

# 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.

# Build Requirements

- Return helpful error messages
- Platform independant
- Dependency resolution
- Command-line processing
- Self-documentation
- Simple



# 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.

# Task Runners

- Grunt
  - Configuration over code
- Cake
  - Ships with and requires Coffeescript.
  - Code over configuration
- Jake
  - Code over configuration
  - Simple
- Gulp
  - Code over configuration
  - Pipes
- Broccoli
  - Code over configuration
  - Filters

# Gulp

- Streaming build system
- Uses node's streams
- It's all code, so very flexible
- File manipulation is done in memory
- Data is passed and piped through multiple steps and plugins

# Lab 5 - Set Up a Task Runner

- `npm install -g "gulpjs/gulp-cli#4.0"`
- `npm install --save-dev --ignore-scripts "gulpjs/gulp-cli#4.0"`
- `git add .`
- `git commit -am "installed gulp"`
- `npm rebuild`
- `git status`





# gulpfile.js

```
/* File: gulpfile.js */

// include gulp packages
const gulp = require('gulp');

// default task
gulp.task('default', function(done) {
  console.log('BUILD OK');
  done();
});
```

# Gulp API

- Gulp top-level functions
  - `gulp.task`
    - defines tasks
  - `gulp.src`
    - points to files to use
  - `gulp.dest`
    - point to the output folder
  - `gulp.watch`
    - watches files and triggers tasks
  - `gulp.series`
    - run tasks in series
  - `gulp.parallel`
    - run tasks in parallel

# gulp.task

- Defines a task
- Takes 3 arguments
  - name
    - a string that will be used to run the task
  - fn
    - the stuff to do after dependant tasks

# gulp.src

- Indicates the files we want to use.
- Uses `.pipe` for chaining its output into plugins
- Example:

```
gulp.src('source/*.html')  
    .pipe(gulp.dest('public'));
```

`gulp.dest`

- Points to the output folder to write files to.

# gulp.watch

- Watches indicated files for changes
- Runs the task when changes happen.
- Example:

```
gulp.watch("src/js/**/*.js", ['jshint']);
```

- This watches JavaScript files in `src/js` and runs the `jshint` task when they change.

# Run Gulp

```
$ gulp
```

```
$ gulp build
```

```
$ gulp run
```

```
$ gulp build run
```

# Lab 6 - Managing External Dependencies

1. Create a gulp task called version 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 JSHint
- Configure JSHint
- 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

# Lab 7 - Automate Linting

- Install JSHint or ESLint into project
- Test it
- Create gulp task called "lint"
- Make lint task a dependency of default task

# Configuring JSHint

- You can set many different jshint options by passing properties:values in the JSHint configuration object parameter.
- *<http://jshint.com/docs/options/>*

# Recommended JSHint Options

```
.pipe(jshint({
  esversion: 6,
  bitwise: true, // no bitwise operators
  curly: true, // curly braces required around blocks
  eqeqeq: true, // require strict comparison
  forin: true, // require for in loops to filter object's items
  freeze: true, // prohibit overwriting native objects
  latedef: "nofunc", // trying to use variables before defining
  nonbsp: true, // no nonbreaking white space

  // set environments
  node: true,
  browser: true,
  globals: []
}))
```

# 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"]
  }
}
```

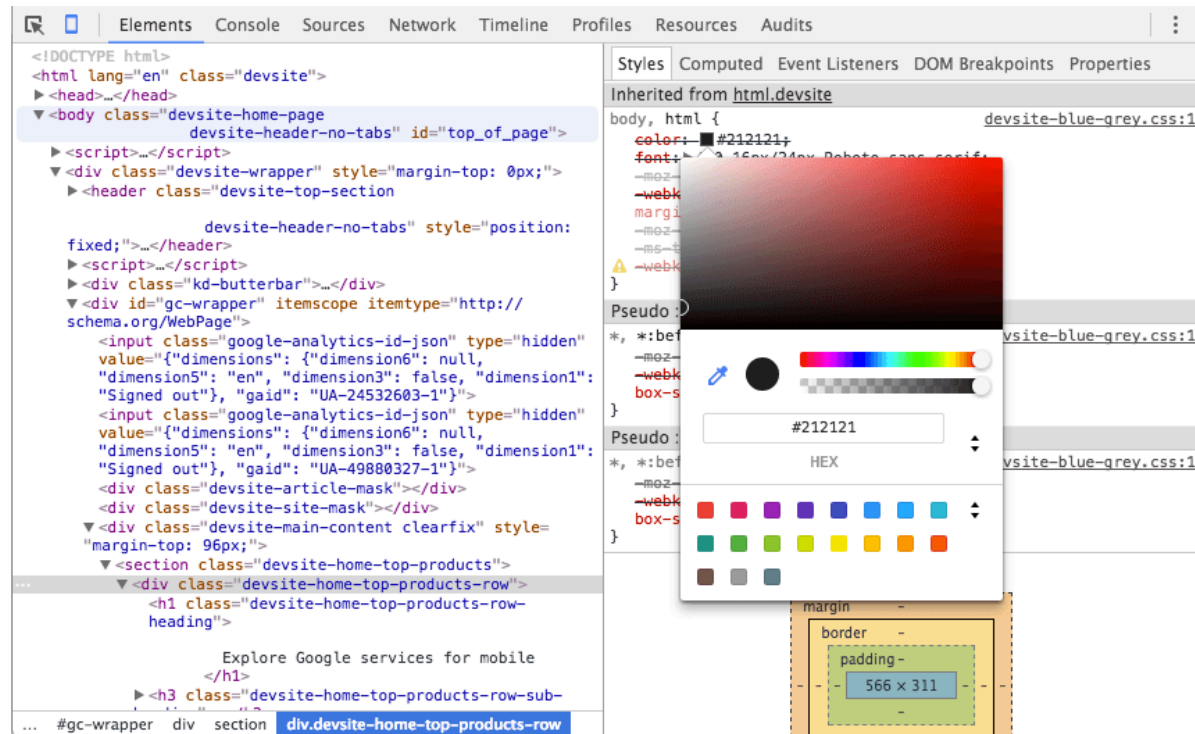


# Lab 8 - Configure a Local Web Server

- Install gulp-webserver
- Add webserver task to gulp

# Browser Development Tools

- Using Chrome Developer Tools
- Using IE F12 Developer Tools



# 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

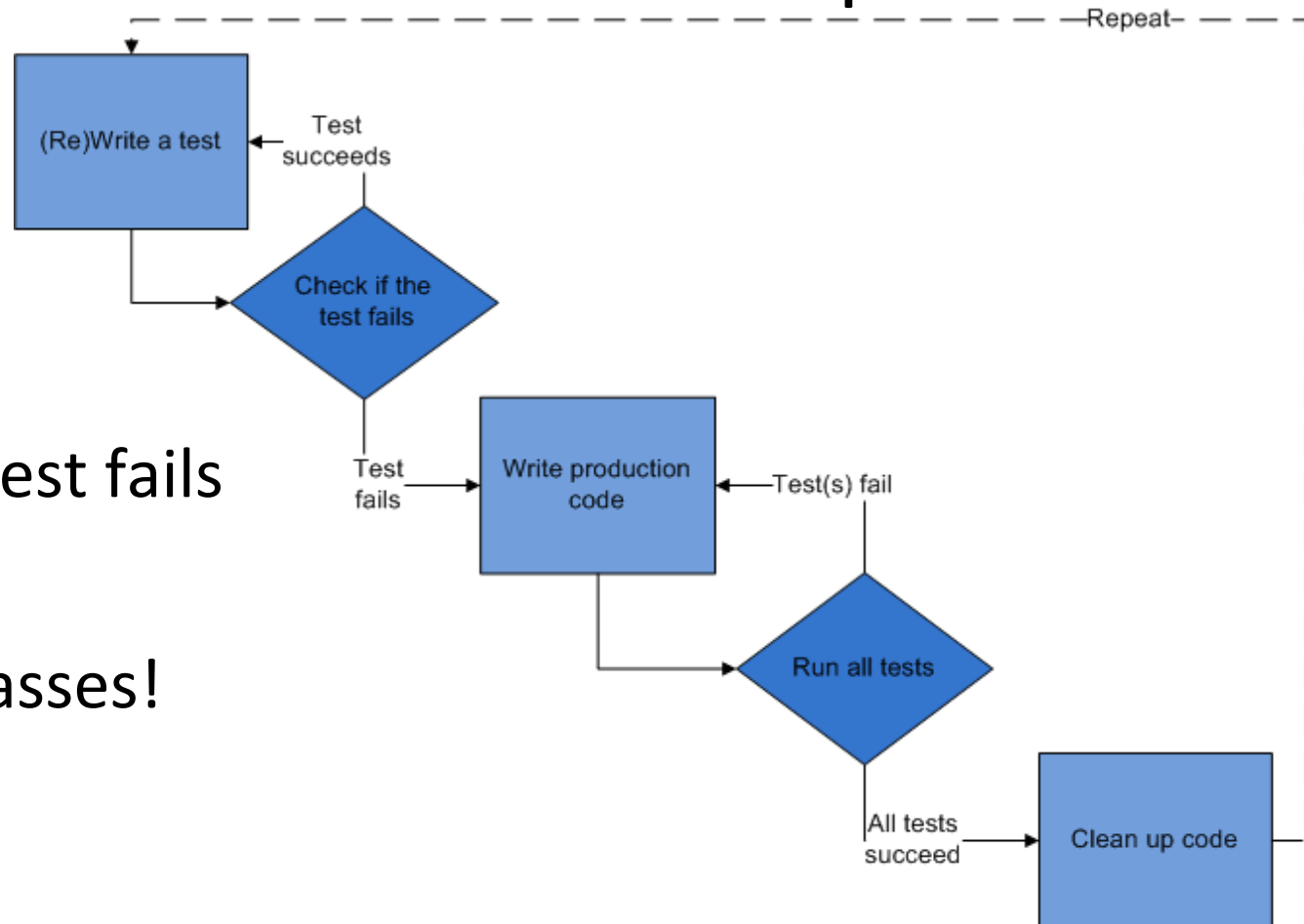
- Clean code that works.

# 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 2<sup>nd</sup> 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
- Integrate Jasmine and Gulp



# 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` functions
- Contains one or more expectations
- `expectations === assertions`

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



# Expectations

- 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 - Integrate Testing into Your Build

- Install gulp-jasmine
- Create jasmine task
- Make jasmine a dependency of default task

# Lab 10.5

- We need more features!!! Pick a feature and implement it using TDD. Break up the 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 4<sup>th</sup> time the function is called.



# Lab 10.75: Course Homepage

- <http://watzthisco.github.io/tdd-react-es6-labs>
- This is the (very new) homepage for the course. I'm starting to post articles there that are good for going deeper into some topics covered by the course.
- Please bookmark and check it out as you have time, and if you find an article that should be there, post it to the course issues
  - <https://github.com/watzthisco/tdd-react-es6-labs/issues>

# 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
- Three possible approaches:
  - Manage manually
  - Manage with npm / webpack
  - Bower

# 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 or browserify



# Chapter 6:

## Building and Refactoring

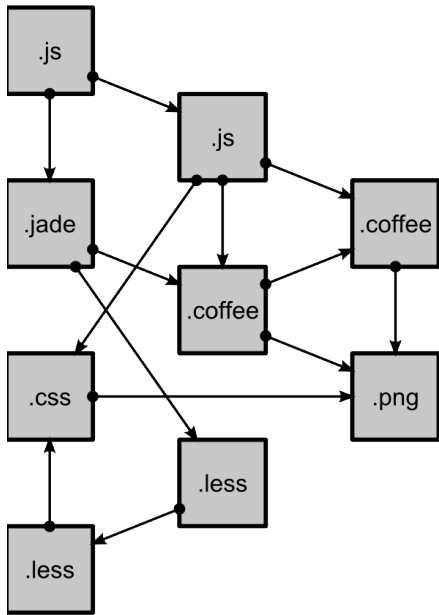
### Objectives

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

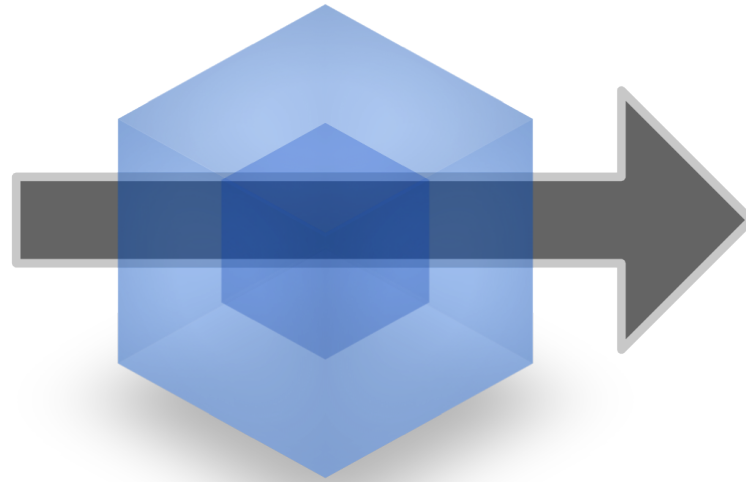
# Building the `dist` directory

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

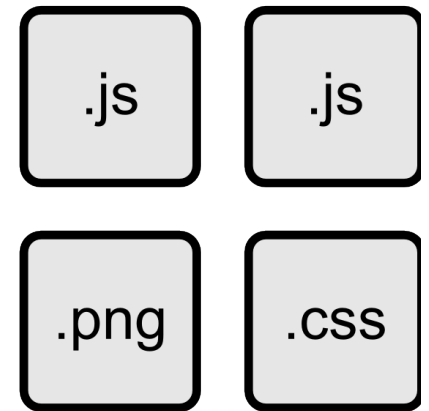
# webpack



modules  
with dependencies



**webpack**  
MODULE BUNDLER



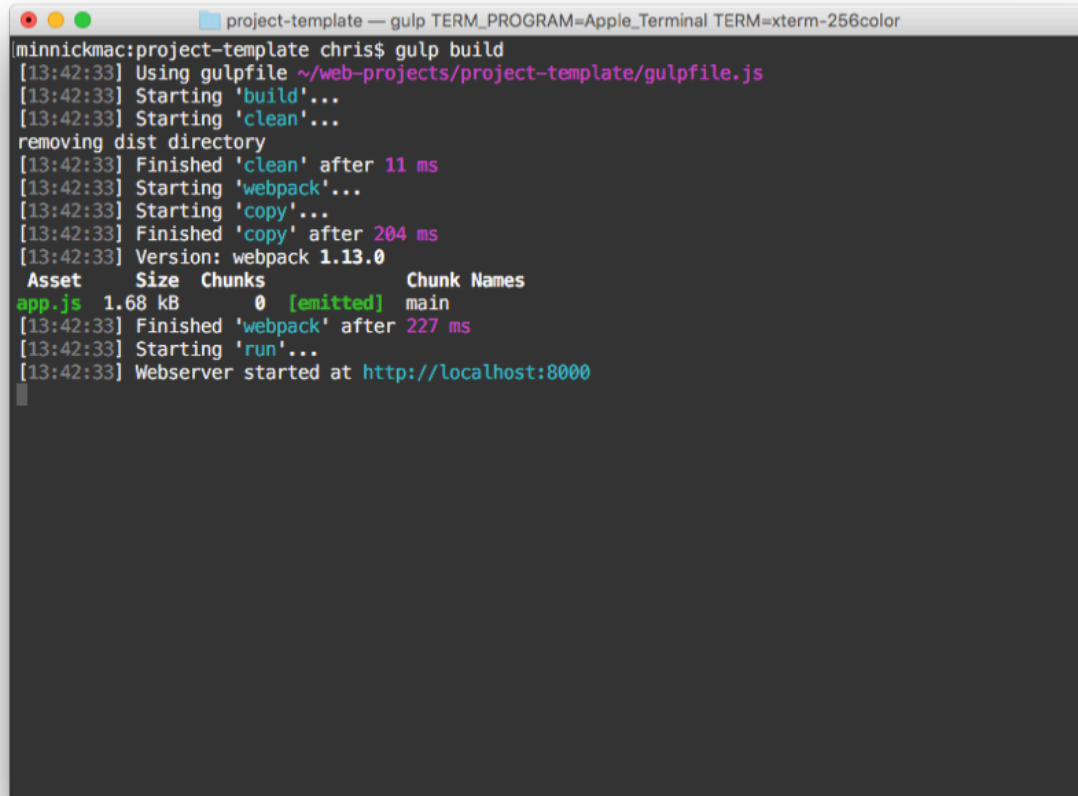
static  
assets

# 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



```
project-template — gulp TERM_PROGRAM=Apple_Terminal TERM=xterm-256color
minnickmac:project-template chris$ gulp build
[13:42:33] Using gulpfile ~/web-projects/project-template/gulpfile.js
[13:42:33] Starting 'build'...
[13:42:33] Starting 'clean'...
removing dist directory
[13:42:33] Finished 'clean' after 11 ms
[13:42:33] Starting 'webpack'...
[13:42:33] Starting 'copy'...
[13:42:33] Finished 'copy' after 204 ms
[13:42:33] Version: webpack 1.13.0
   Asset      Size  Chunks             Chunk Names
app.js  1.68 kB          0  [emitted]  main
[13:42:33] Finished 'webpack' after 227 ms
[13:42:33] Starting 'run'...
[13:42:33] Webserver started at http://localhost:8000
```

# Lab 13: README Update and Refactoring

# Chapter 7:

## ES2015 (ES6)

### 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

# "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';  
    ...  
}
```

- Necessary in ES6, except in modules, where it's implied

# 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

- More expressive closure syntax

- ES6

- ```
odds = evens.map (v => v+1);
```

- ES5

- ```
odds = evens.map (function (v) { return v+1; });
```



# 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.)

- Custom Interpolation
- Expression interpolation for arbitrary methods

```
get`http://example.com/cart?order=${orderId}`;
```

# 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

```
var Square = 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;  
    }  
}
```

# 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"
```

# reduce ()

- **Array.reduce()**
  - Applies a function against an accumulator and each value of the array to reduce it to a single value.
  - Takes two parameters
    - A callback function (reducer)
    - An initial value (array)

```
var total = [0, 1, 2, 3].reduce(function(a, b) {  
    return a + b;  
});  
// total == 6
```

- <http://jsbin.com/regosow/edit?js,console>

# Promises

- First class representation of a value that may be made asynchronously and be available in the future

```
function msgAfterTimeout (msg, who, timeout) {  
  return new Promise((resolve, reject) => {  
    setTimeout(() =>  
      resolve(`${msg} Hello ${who}!`), timeout)  
    })  
}  
  
msgAfterTimeout("", "Foo", 100).then((msg) =>  
  msgAfterTimeout(msg, "Bar", 200)  
) .then((msg) => {  
  console.log(`done after 300ms:${msg}`)  
})
```

- <http://jsbin.com/dozimot/edit?js,console>

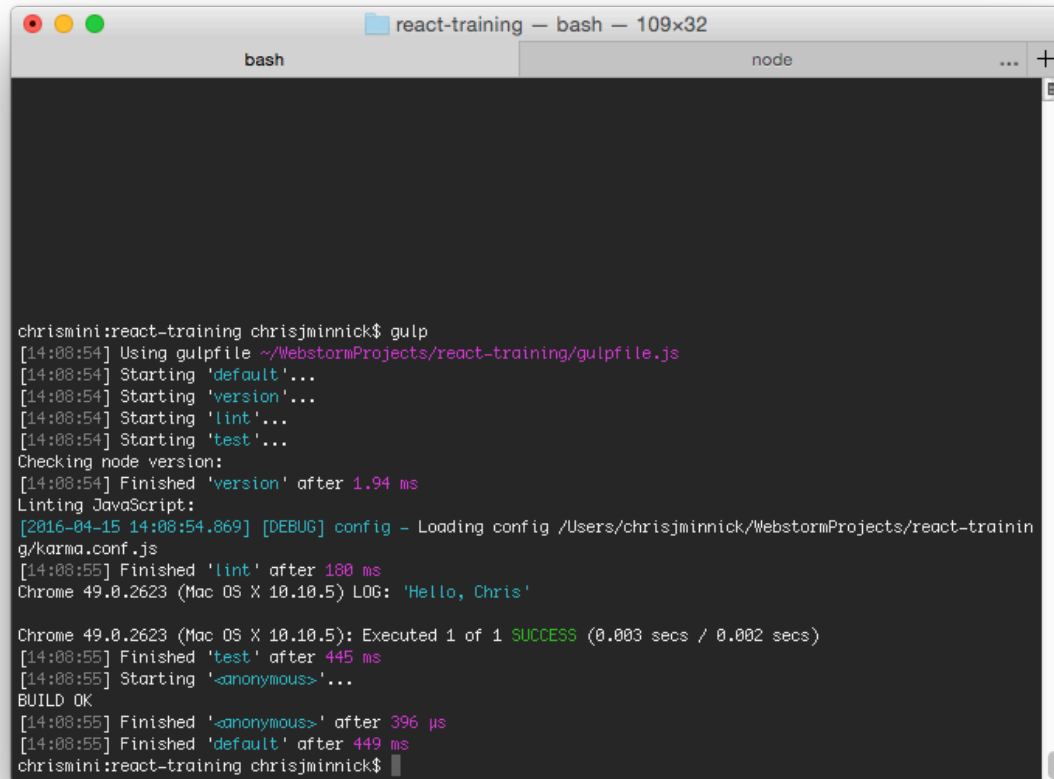


# Babel

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

The word "BABEL" is written in a large, bold, black, stylized font that resembles a thick brushstroke or calligraphy. The letters are slanted to the right. A thin vertical line is positioned to the right of the word.

# Lab 14: Transpiling with Babel

A terminal window titled "react-training -- bash -- 109x32" with tabs for "bash" and "node". The terminal shows the output of running "gulp" in a directory named "react-training". The output includes timestamps, task names, and durations for tasks like "default", "version", "lint", "test", and "anonymous". It also shows the execution of "karma.conf.js" and the final "BUILD OK" message.

```
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-training/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:

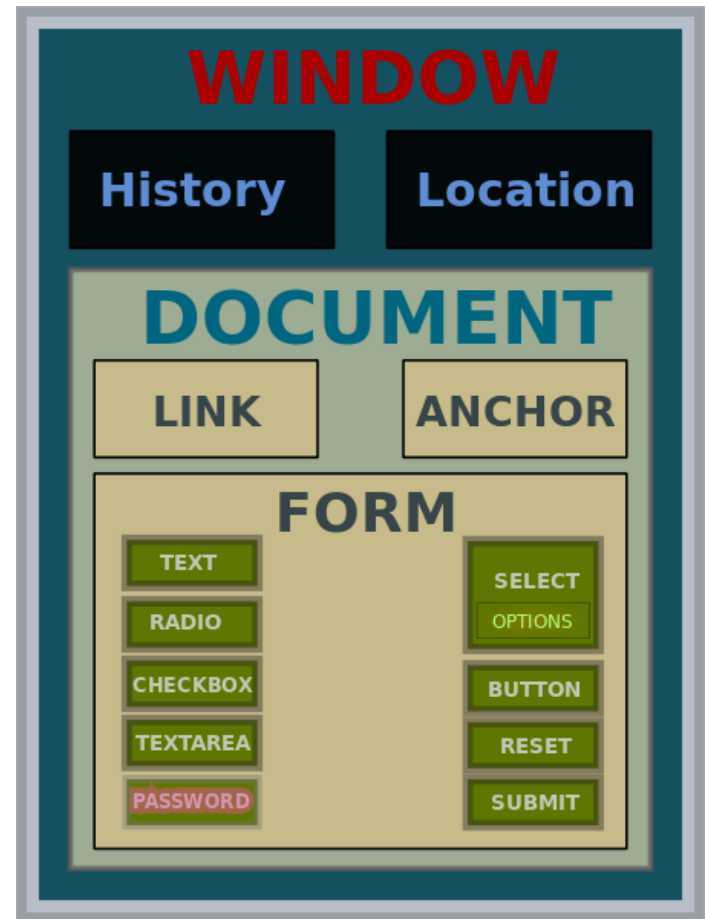
# The Document Object Model

## 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>
```

```
var FavoriteSongs = React.createClass({
```

```
  render: function() {
```

```
    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'));
```

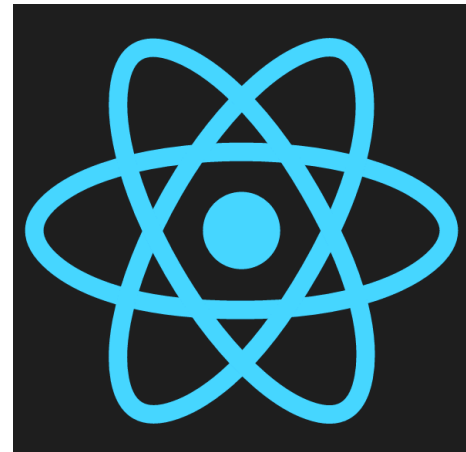
```
</script>
```

# 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 comparable 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,

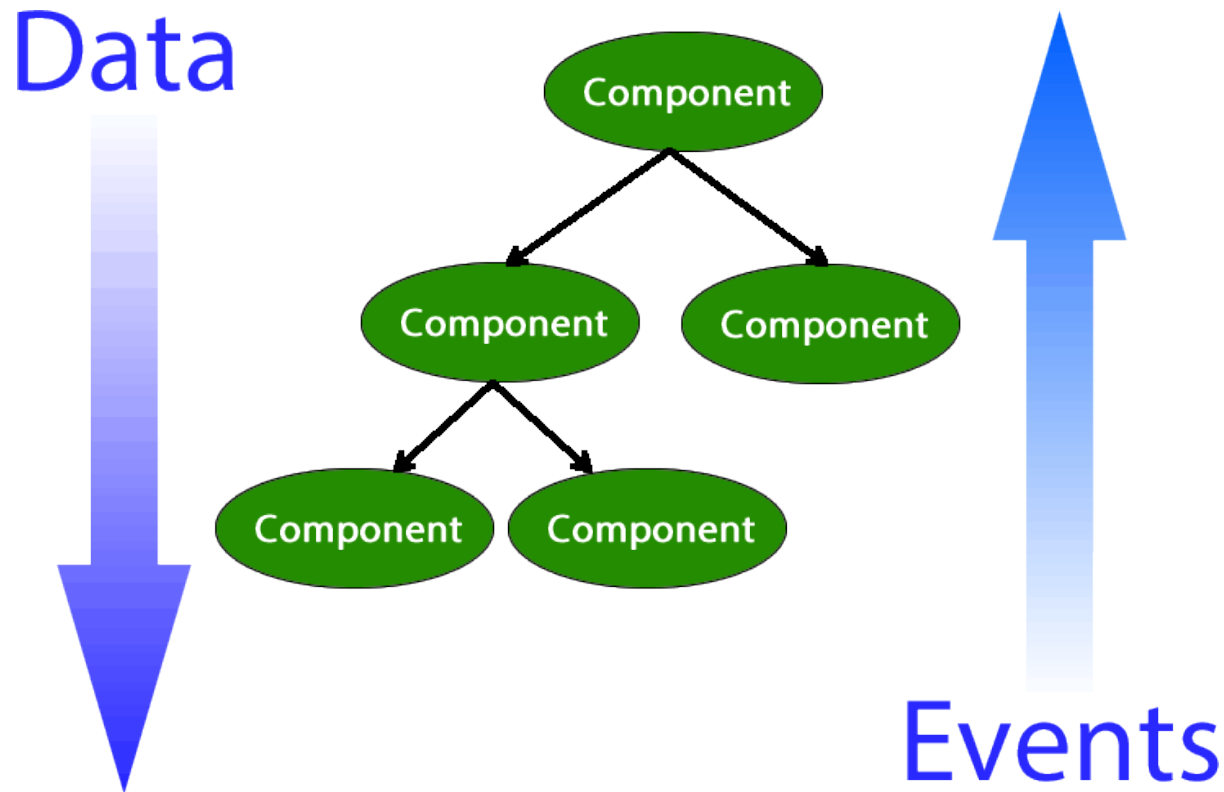
The Management

# 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



# 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

# Changing Props and State

## Changing *props* and *state*

-	<i>props</i>	<i>state</i>
Can get initial value from parent Component?	Yes	Yes
Can be changed by parent Component?	Yes	No
Can set default values inside Component?	Yes	Yes
Can change inside Component?	No	Yes
Can set initial value for child Components?	Yes	Yes
Can change in child Components?	Yes	No

# Setting Initial State

```
// using createClass method
var MyComponent = React.createClass({
  getInitialState() {
    return { /* some initial state */ }
  },
```

```
// using ES6
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    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.



# 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

# Understanding Components

- Created using `createClass()` or by extending `React.Component`
  - Pass in an object with a render method
  - Returns a single element
    - 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
- `<div><p><em></em></p></div>` = correct
- `<div></div><p></p>` = wrong

# 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'},  
  {category:  
    'Sporting Goods',  
    price: '$29.99',  
    stocked: true, name:  
    'Basketball'},  
  {category:  
    'Electronics',  
    price: '$99.99',  
    stocked: true, name:  
    'iPod Touch'},  
  {category:  
    'Electronics',  
    price: '$399.99',  
    stocked: true, name:  
    'iPhone 5'},  
  {category:  
    'Electronics',  
    price: '$199.99',  
    stocked: true, name:  
    'Nexus 7'}  
];
```

☐ Only show products in stock

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

```
var ProductCategoryRow =  
  
React.createClass({  
  render: function() {  
    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

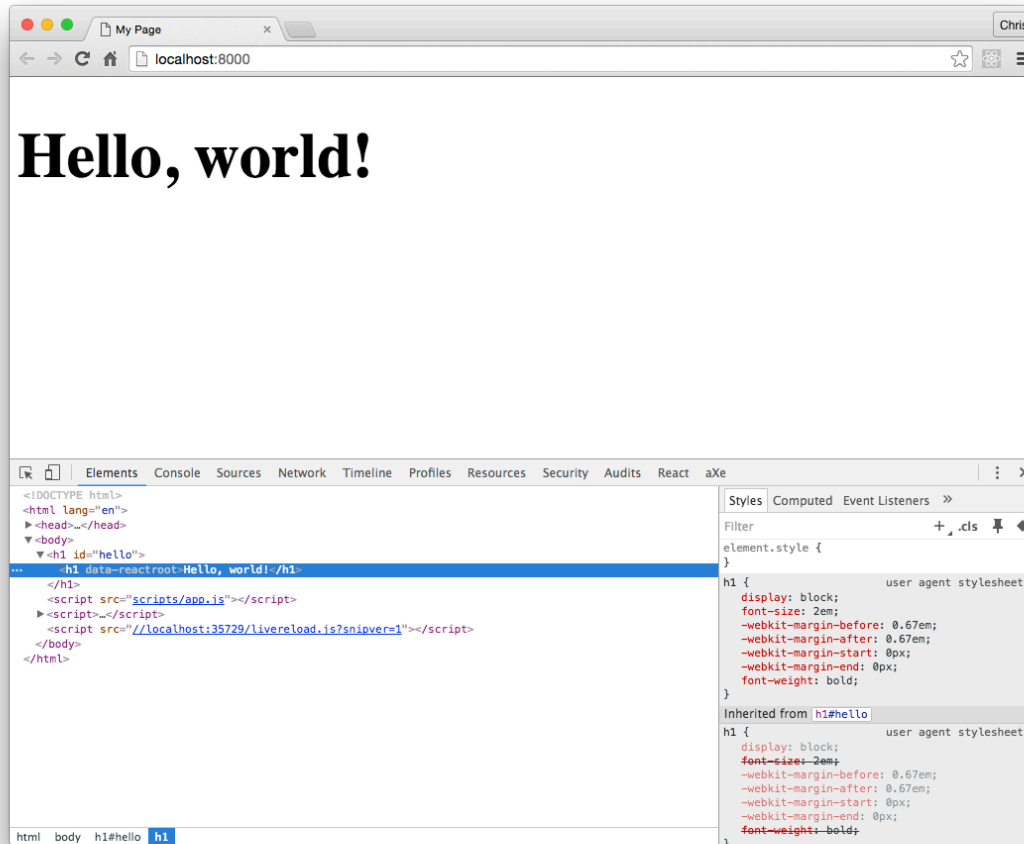
# Step 4 – Where Should Your State Live?

- 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.

# Lab 16: Hello React!





# 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.

# Using JSX

## Objectives

- Write JSX
- Use React with JSX
- Use React without JSX

# 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

```
var LoginBox = React.createClass({
  render: function() {
    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: function 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

```
var Signout = React.createClass({  
  render: function() {  
    return (<div>logout</div>);  
  }  
});
```

```
var Login = React.createClass({  
  render: function() {  
    return(<div>login</div>);  
  }  
});
```

```
var Container = React.createClass({  
  render: function() {  
    return(  
      <div>  
        {this.props.isLoggedIn ? <Signout /> : <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



**Welcome!**

What is this question?

☐ Answer 1

☐ Answer 2

☐ Answer 3

# 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`
- Either way is fine, but there are some differences
- `React.Component` is the ES6 way.
- `React.createClass` may be removed in the future.

# 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

# 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 ReactClass (created via React.createClass).
- JSX gets compiled to React.createElement



# Namespaced Components

- You can create one component that has other components as attributes in order to simplify your code.
- To create sub-components using `React.createClass`:

```
var MyTableComponent = React.createClass({...});  
MyTableComponent.Row = React.createClass({...});  
MyTableComponent.Cell = React.createClass({...});
```

# Using Namespaced Components

- Use dot syntax to access sub-components.

```
var Table = myTableComponent;  
var App = (  
  <Table>  
    <Table.Row>  
      <Table.Cell />  
      <Table.Cell />  
    </Table.Row>  
  </Table> );
```

# 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).

# Benefits of Pure Functions

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

# F.I.R.S.T.

- React Components should be:
  - **F**ocused
  - **I**ndependant
  - **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*

# 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!**

# `this.props.children`

- **Accesses contents of an element.**

```
<p>This is my paragraph.</p>
```

```
this.props.children = "This is my  
paragraph."
```

```
<ToDoItem>
```

```
  <ToDoCheckbox />
```

```
  <ToDoDescription />
```

```
</ToDoItem>
```

- **in `ToDoItem`, `this.props.children` is an array of components containing `ToDoCheckbox` and `ToDoDescription`.**



## Lab 18: Passing Props

**Welcome to the poll!**

**What is the best?**

☐ **Tacos!**

☐ **Pizza!**

☐ **Cheese!**

**Go!**

# 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>
```

- <http://www.webpackbin.com/V1VNoINA->

# 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

**Welcome to the poll!**

**What is the best?**

- ☐ Tacos!
- ☐ Pizza!
- ☐ Cheese!

**Go!**

# 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.
- <http://codepen.io/chrisminnick/pen/LNXVMY>

# Uncontrolled Components

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

```
render: function() {  
  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
- ☐ Pizza
- ☒ Cheese

Current Choice: Cheese

**Go!**

# Stateless Functional Components

- If your component only has a render method and optional props, you can just create a normal JavaScript function, without the `createClass` abstraction.

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

# 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



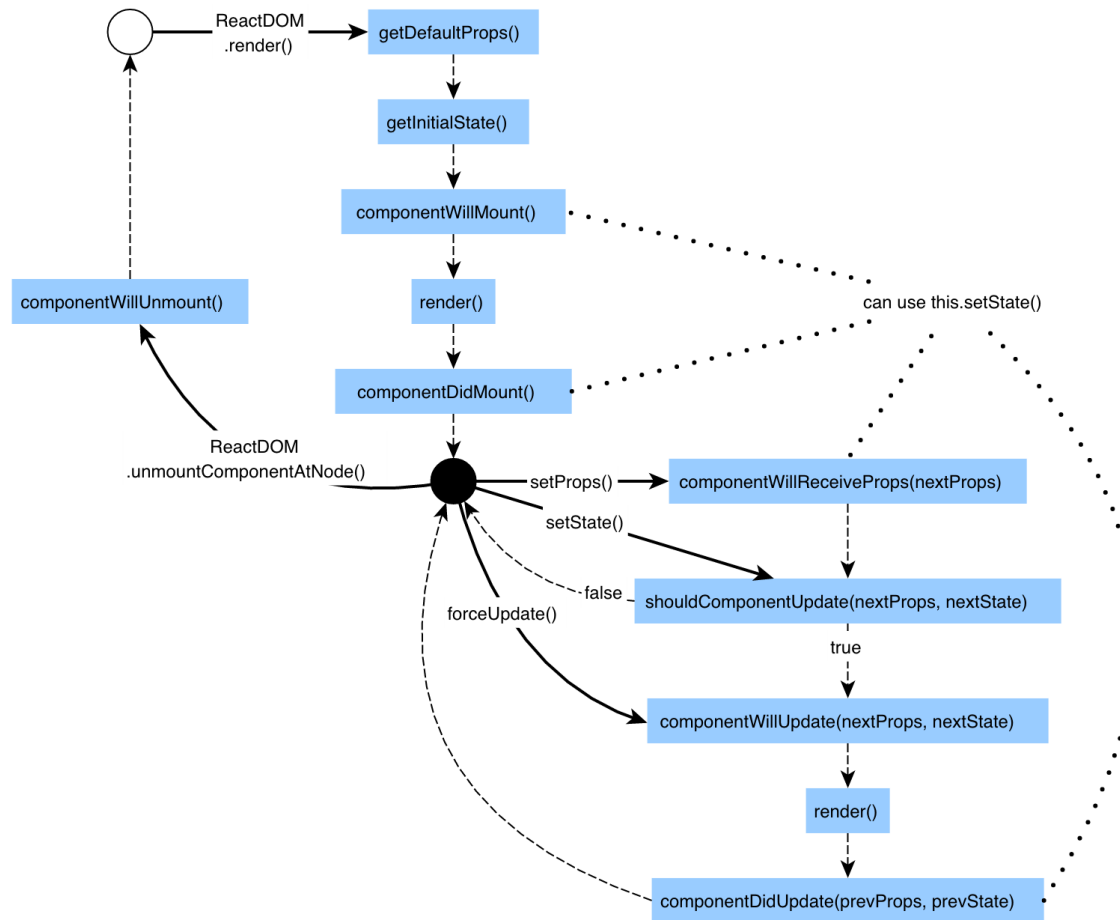
# Mount/Unmount Life-Cycle Methods

- `getDefaultProps`
- `getInitialState`
- `componentDidMount`
- `componentWillUnmount`

# Data Life-Cycle Methods

- `componentWillReceiveProps`
- `shouldComponentUpdate`
- `componentWillUpdate`
- `componentDidUpdate`

# Component Life Cycle



# Events

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

```
var ShareButton = React.createClass({
  onClick: function (evt) {
    alert("wow!");
  },
  render: function () {
    return (
      <div onClick={this.onClick}>Share!</div>
    );
  }
});

ReactDOM.render(<ShareButton />,
  document.getElementById("share"));
```

# Lab 22: Life Cycle and Events

Elements Console Sources Network Timeline Profiles Resources Security >> ⋮ ✕		
⊘ 🔍 top ▼ <input type="checkbox"/> Preserve log		
componentWillMount		<u>app.js:20275</u>
componentDidMount		<u>app.js:20280</u>
componentWillUpdate		<u>app.js:20304</u>
componentDidUpdate		<u>app.js:20309</u>
componentWillUpdate		<u>app.js:20304</u>
componentDidUpdate		<u>app.js:20309</u>
componentWillUpdate		<u>app.js:20304</u>
componentDidUpdate		<u>app.js:20309</u>
correct		<u>app.js:20269</u>
>		

# Presentational Components

- Components that just output presentation
- Contain no logic

# Container Components

- Wrap presentational components
- Contain the logic and state

# Composition

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



# 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

# PropTypes (cont)

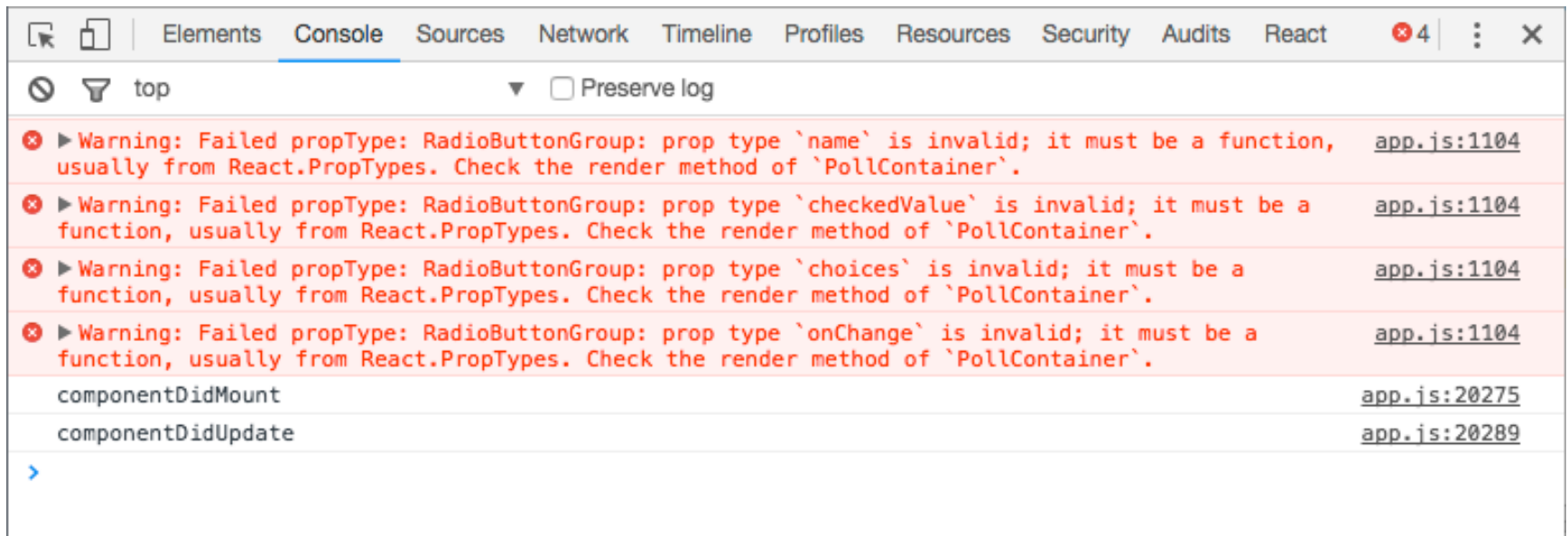
```
var React = require('react')
var PropTypes = React.PropTypes
var Icon = React.createClass({
  propTypes: {
    name: PropTypes.string.isRequired,
    size: PropTypes.number.isRequired,
    color: PropTypes.string.isRequired,
    style: PropTypes.object
  },
  render: ...
});
```

# PropTypes.shape()

- Used for validating nested properties

```
propTypes: {  
  data: React.PropTypes.shape({  
    id: React.PropTypes.number.isRequired,  
    title: React.PropTypes.string  
  })  
}
```

# Lab 23: PropTypes



# 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>`

# 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 `componentWillMount`
    - 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.

# 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'
  }
}
```

# 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.

# Testing React Components

## Objectives

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

# Shallow Rendering

- Renders just a single component, regardless of where it is in the hierarchy.
- Allows you to isolate components for testing.

# 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

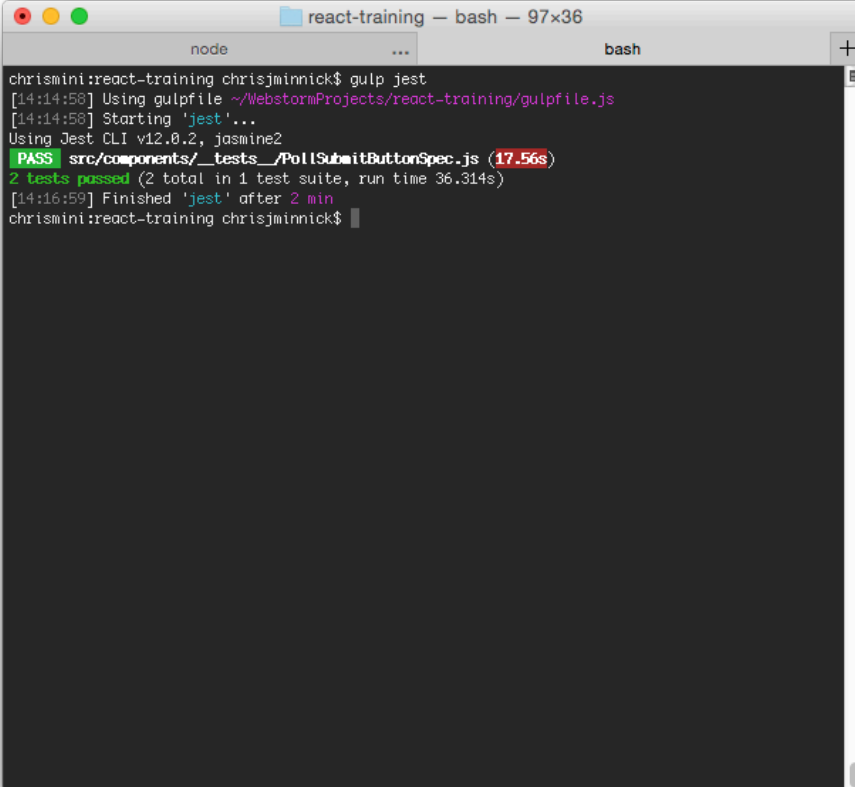
# TestUtils

- `renderIntoDocument()`
- `mockComponent()`
- `isElement()`
- `isElementOfType()`
- `isDomComponent()`
- `isCompositeComponent()`
- `isCompositeComponentWithType()`
- `findAllInRenderedTree()`
- `more...`

# Enzyme

- Testing utility for React, created by AirBnB
- Reduces boilerplate
- 3 render modes
  - shallow
  - mount
  - render
- To use, pass a React component into a mode method.
  - `const wrapper = shallow(<PollSubmitButton />);`

# Lab 24: Testing React Components

A terminal window titled "react-training — bash — 97x36" with tabs for "node" and "bash". The terminal shows the execution of "gulp jest" in a directory "chrismini:react-training chrisjminnick\$". The output includes timestamps, the use of a gulpfile, starting Jest, and the Jest CLI version. It shows a successful test run for "src/components/\_tests\_/PollSubmitButtonSpec.js" with a duration of 17.56s. The final result is "2 tests passed (2 total in 1 test suite, run time 36.314s)" and the completion of the Jest run after 2 minutes.

```
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 enzyme --save-dev
```

```
npm install jest-enzyme --save-dev
```

- Put this at the top of your test files

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

- Make sure your package.json includes the following:

```
"jest": {  
  "setupTestFrameworkScriptFile": "node_modules/  
  jest-enzyme/lib/index.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

**Go!**

# React Router

- Declarative way to do routing
- Maps components to URLs

...

```
var routes = (  
  <Router history={browserHistory}>  
    <Route path="/" component={App}>  
      <Route path="about" component={About}/>  
      <Route path="users" component={Users}>  
        <Route path="/user/:userId" component={User}/>  
      </Route>  
      <Route path="*" component={NoMatch}/>  
    </Route>  
  </Router>  
>;  
module.exports = routes;
```

# Lab 26: React Router

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

## Lab 27: React Router, Part 2

- Going deeper into React Router to use `browserHistory` and `IndexRoute`.

# 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

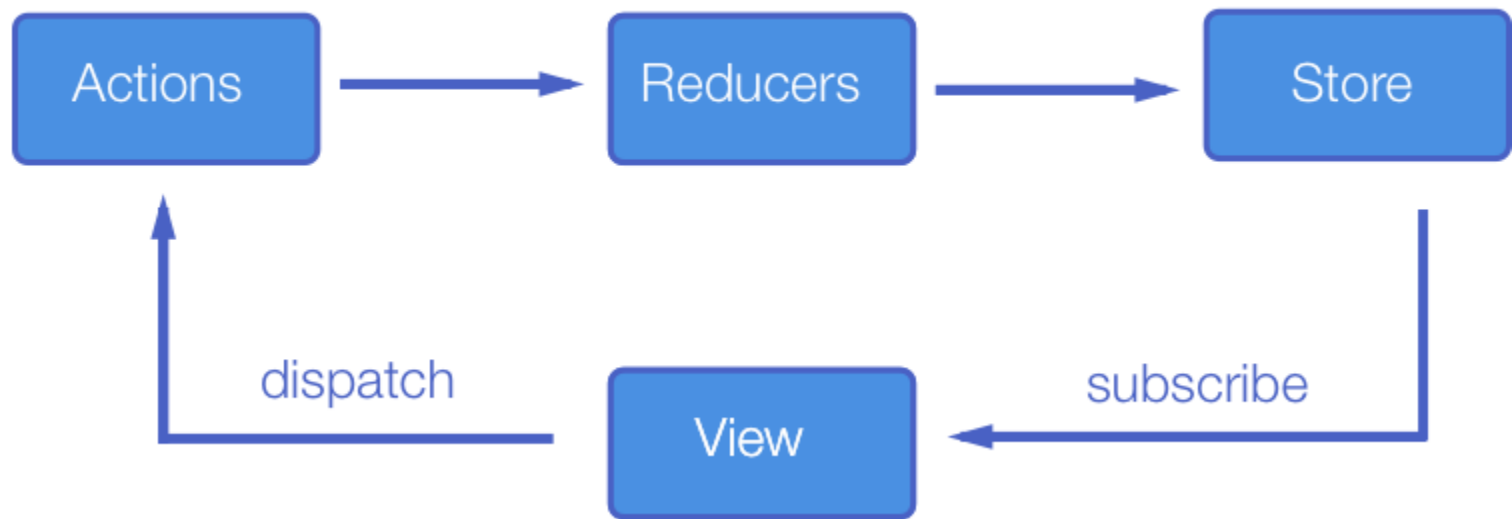
# Stores & Immutable State Tree

- 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'
}
```





# 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.slice(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)`

# 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 Async Actions
    - Thunk Middleware
  - Relay
    - Good for large applications
    - Declare data needs of components with GraphQL
    - Relay automatically downloads data and fills out the props

# Redux with Ajax

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



# Redux Pros and Cons

## Redux Pros

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

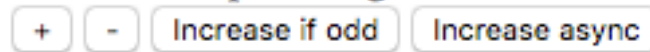
## Redux Cons

- 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.

# 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

# Lab 31: Redux Async Actions

- Retrieve poll data from the server using Redux Async Actions

# create-react-app

- New, no-config method for setting up and running React projects.

# Lab 32: create-react-app and enzyme

- Install create-react-app
- Create a project using create-react-app
- Write some new components
- Write tests for the new components
- Write a test first and then writing a component to increment a counter when a button is clicked.
- As time allows, write more complex tests and components.

# 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
- Single Responsibility

# 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()`



# 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.

# Using Pre-built Components

- Thousands of React components are shared via npm
  - Searchable through databases like [React-Components.com](https://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

## **Disclaimer**

- WatzThis? takes care to ensure the accuracy and quality of this courseware and related courseware files. We cannot guarantee the accuracy of these materials. The courseware and related files are provided without any warranty whatsoever, including but not limited to implied warranties of merchantability or fitness for a particular purpose. Use of screenshots, product names and icons in the courseware are for editorial purposes only. No such use should be construed to imply sponsorship or endorsement of the courseware, nor any affiliation of such entity with WatzThis?.

## **Third-Party Information**

- This courseware contains links to third-party web sites that are not under our control and we are not responsible for the content of any linked sites. If you access a third-party web site mentioned in this courseware, then you do so at your own risk. We provide these links only as a convenience, and the inclusion of the link does not imply that we endorse or accept responsibility for the content on those third-party web sites. Information in this courseware may change without notice and does not represent a commitment on the part of the authors and publishers.

## **Copyright**

- Copyright 2016, WatzThis?. All Rights reserved. Screenshots used for illustrative purposes are the property of the software proprietor. This publication, or any part thereof, may not be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, storage in an information retrieval system, or otherwise, without express written permission of WatzThis, PO BOX 161393, Sacramento, CA 95816. [www.watzthis.com](http://www.watzthis.com)

## **Help us improve our courseware**

- Please send your comments and suggestions via email to [info@watzthis.com](mailto:info@watzthis.com)