React

Speed Coding

Code Along Edition

ES6 Redux Webpack Firebase Enzyme Flexbox

reactspeed.com

Manav Sehgal

React Speed Coding

From concept to coding real world React apps, speedily!

Manav Sehgal

This book is for sale at http://leanpub.com/reactspeedcoding

This version was published on 2016-08-29

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Manav Sehgal by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought React Speed Coding. On my way to develop my new app soon!

The suggested hashtag for this book is #reactspeedcoding.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#reactspeedcoding

Also By Manav Sehgal

React Eshop

Shalini, Rama

Contents

Awesome React Ecosystem

Welcome reader. The aim of this book is to get you from concept to coding real world React apps, as fast as possible.

React ecosystem is constantly evolving and changing at a fast pace. This book equips you to take the right decisions matching your project requirements with best practices, optimized workflows, and powerful tooling.

Code Along Edition

We are significantly revising this edition of React Speed Coding book.

- Adding new Code Along GitHub repository[1] containing branches for code you complete in each chapter.
- Chapter by chapter demos are available at new demos[2] website.
- Making each chapter stand on its own so you can complete a significant learning step at the end of each chapter.
- Ensuring that your learning path is as linear as possible, without too many cross-references due to refactoring of code we write in each chapter.
- The new edition also features better typography, color coding, and more screenshots to aid your learning.
- Upgrading to the latest development environment and dependencies as of this writing.
- Adding new sections to make your React journey faster, easier, and better.
- Reducing the code you write to achieve the same goals.
- Redesigning the UI CSS using Block, Element, Modifier method for more scalable, yet less verbose design.
- Several new custom components for Buttons, Forms, Layout, and other features.

[1] https://github.com/manavsehgal/react-speed-book

[2] https://manavsehgal.github.io/react-speed-demos/

Easy start React

If you want to dig into React coding right away, you can fast forward first four chapters in this book.

Getting started with React is now easier than ever. Get your first React app up and running in three easy steps. You may want to read the sample chapter titled Easy Start React[3] from our new book React Eshop. We use Facebook's **Create React App** scaffold generator to fast start into our first React app.

Go ahead give it a go. You will appreciate the magic behind the scenes as we learn to build a powerful development toolchain on similar lines as Facebook's Create React App in this book. This will result in two benefits for our readers. First, you will learn about important React ecosystem technologies including Webpack, PostCSS, ESLint, and Babel in a step-by-step linear manner. Second, you will be able to easily extend apps generated using Create React App as you mature beyond the scaffolded code and the opinionated defaults used by Create React App.

Who this book is for

The React Speed Coding book assumes basic knowledge of programming in JavaScript, HTML, and CSS.

If you are a complete beginner, there is enough guidance available for you to make this your first programming primer with suggested additional reading.

Experienced web developers will master React component design workflow using latest ES6 language features. If you already program in React, you can use this book to optimize your development, testing, and production workflow.

Development environment

This book assumes you have access to a Mac, Linux, or a Cloud based editor offering virtual machine hosted development environment in your Web browser.

We will walk you through the entire React development environment setup from scratch.

While it is possible to run samples from this book using Windows, there are known issues[4] in setting up Node and certain NPM packages.

[3] https://leanpub.com/reacteshop/read#leanpub-auto-easy-start-react

[4] https://github.com/nodejs/node-gyp/issues/629

As a safe alternative you can use any of the Cloud based code editors which offer Linux Development environment within the convenience of your web browser. You do not need to know how to use Linux to operate these web based editors. You can start with a generous free account with basic stack including Node.js already setup for you.

Cloud9[5] is our favorite web based code editor. Other options include Nitrous[6].

On Mac or Linux you can use your favorite code editor. This book is written using the open source Atom[7] editor from the Github team. Atom gets you started coding by just dragging and dropping a folder onto the editor. You can then add power user features as you grow with Atom using custom packages, code snippets, among others.

[5] https://c9.io/
[6] https://www.nitrous.io/
[7] https://atom.io/

Why React is awesome

Writing the React Speed Coding book, companion code, and the ReactSpeed.com[8] demo website has been fun and fulfilling at the same time. Thanks to the amazing ecosystem that React and open source community have created in a relatively short span of time.

What we love about React and companion libraries like Redux is how they introduce constraints and flexibility at the same time. A very difficult goal to achieve when writing generic libraries and frameworks. React and Redux seem to have done so elegantly. Growing GitHub stars and cross-industry adoption is proof of this achievement.

To us React is about thinking in design and architectural patterns. It is more than making choices about which framework or library to use, or how to use these. We rapidly raise our thinking to design, requirements, solving real-world problems, that our apps are expected to address.

Learning React is about future proofing our investment more than any other framework or library with similar goals. Thanks to flexibility of integrating with React, even some of the competing frameworks offer integration paths with the React ecosystem. These include Meteor-React integration, Redux use cases with Angular, and TypeScript-React playing well together, just to name a few.

Most awesome aspect of learning React is that it is an ecosystem. It has a life of its own above and beyond Facebook, the original authors of React core. No wonder you see companies like Netflix, Airbnb, Kadira, Khan Academy, and Flipboard contributing their React libraries and tools to the open source.

Successfully navigating this growing ecosystem, making the right technology stack decisions along the way, will make the difference between an average programmer and a world-class designer-developer of the future. We sincerely hope React Speed Coding can contribute to your journey in mastering the React ecosystem of technologies.

Here’s to moving from Concept to Code to Cash, speedily!

[8] https://reactspeed.com

Why read React Speed Coding

React Speed Coding enables you to optimize your React development workflow and speed up the app design lifecycle.

Setup React Webpack development environment complete with Node and Babel including development, testing, and production workflows. Production optimize Webpack development toolchain for CSS, JS, HTML pre-processing, faster builds, more performant code.

Learn ES6 React features including arrow functions, template literals, variable scoping, immutability, pure functions, among others.

Create complete single page app using Redux store, actions, and reducers.

Create custom React Speed UI library using Flexbox and PostCSS, with goals including responsive design, single page app components, ease of customization, reusable code, and high performance.

Apply Behavior-Driven Development techniques to create a comprehensive testing strategy for your apps. This includes ESLint and StyleLint to provide in-editor coding guidance on industry best practices for JavaScript and CSS. Use Mocha to describe specs. Chai for writing assertions. Sinon to spy on methods and events. We also learn about Enzyme for simple yet powerful React component level testing.

Adopt a comprehensive component design workflow including five strategies for s tarting component design by creating React components from embeds, REST APIs, samples, and wireframes.

Integrate your apps with serverless architecture using Firebase hosting. Create REST API for component design workflow using Firebase visual tools. Connect custom React components you create in this book with Firebase realtime database.

Run demo app and components live at ReactSpeed.com[9] website.

Visit our popular GitHub repository[10] to download and reuse source code from this book.

[9] https://reactspeed.com

[10] https://github.com/manavsehgal/reactspeedcoding

Prior art

The author would love to take the credit for coining the term “Speed Coding”. However, Speed Coding is based on very strong foundations and popular prior art.

Speed of Developer Workflow. Speed Coding follows some of the methods and tools as prescribed by the Lean Startup principles. See infographic[11] for code faster, measure faster, learn faster.

Code faster principles we cover in this book include writing unit tests, continuos integration, incremental deployment, leveraging open source, cloud computing, refactoring, just-in-time scalability, and developer sandboxing.

Measure faster principles include usability tests, real-time monitoring, and search engine marketing.

Learn faster principles we apply in this book include smoke tests, split testing, and rapid prototyping.

Speed of Design. Speed Coding embraces the **designer-developer** evolution and also bases certain principles on the Design Thinking[12] methodology and Visual Design[13] principles.

We use these techniques in designing React Speed UI library using custom React components, PostCSS, Flexbox, and SVG.

Speed of Technology Decisions. Speed Coding technology stack is compared with industry best practice guidance including the awesome ThoughtWorks Technology Radar[14].

Technology Stack section, part of Introduction chapter, highlights React ecosystem technologies we cover in this book, in line with recommendations from the Radar.

The cover image for our book depicts NASA space shuttle lift-off and is representative of our central theme. The science of speed. We thank Pixabay[15] for providing this NASA imagery in the Creative Commons.

[11] http://visual.ly/lean-startup

[12] http://www.fastcompany.com/919258/design-thinking-what

[13] https://www.academia.edu/11637848/Visual_Design_Principles_An_Empirical_Study_of_Design_Lore

[14] https://www.thoughtworks.com/radar

[15] https://pixabay.com/en/space-shuttle-lift-off-liftoff-nasa-992/

Stakeholder perspectives on speed

In order to fulfill the promise of Speed Coding, we need to start by establishing some baselines. What are we speeding up? How are we measuring this speed? Why does it matter?

Let us start with the Why. Speed Coding is essential for three stakeholders. The user. The developer. The sponsor.

As app users we define *speed* mostly as performance and reactivity of the app. We even define speed as frequency of timely and desired updates to the apps we are using. Most importantly we define speed by time it takes to get things done.

As developers we define speed in terms of our development workflow. How long does it take to code, build, test, deploy, debug, and reactor. We also define speed of decision making relating to our development and technology stack.

As sponsors for an app project we define speed in terms of time to market. How long does it take to move from **Concept to Code to Cash**. Believe us, you first heard that phrase here, and we truly mean it!

Who is using React

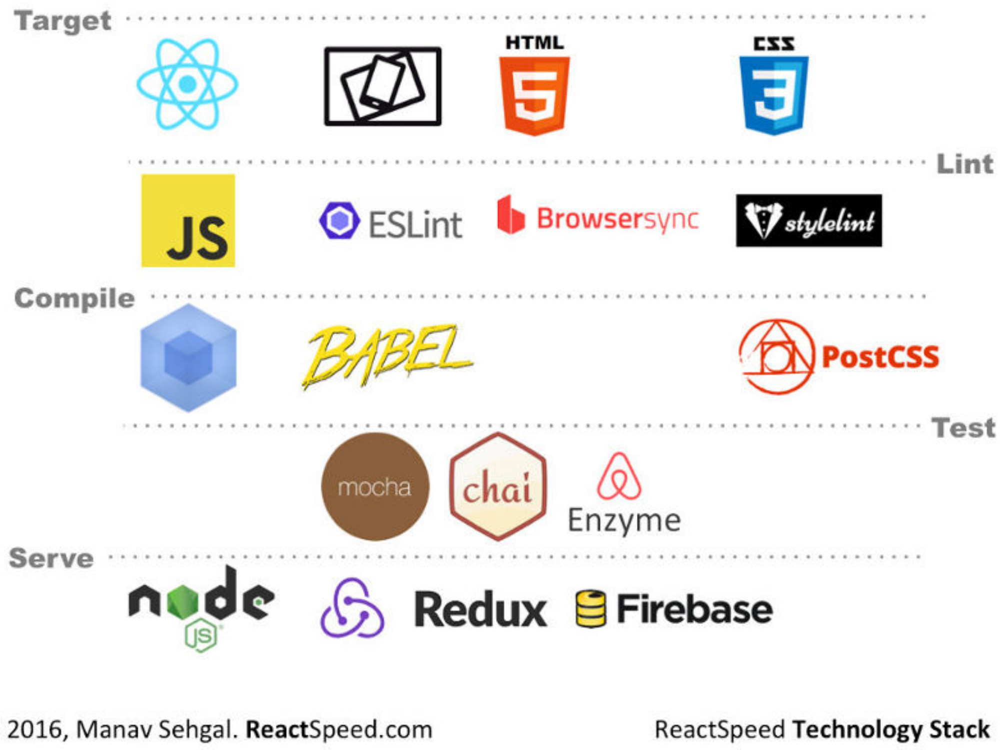
Open sourced, developed, and used by Facebook and Instagram teams. React has also found wide adoption among leading technology, app, and digital companies.

- Airbnb are contributors of the popular React style guide.
- Atlassian redesigned their popular messaging app HipChat in React and Flux. They chose React because it is component based, declarative minus the bloat, uses Virtual DOM, relatively small library as opposed to full framework, simple, offers unidirectional data flow, and easily testable.
- BBC mobile homepage uses React.
- CloudFlare has active React projects on GitHub.
- Flipboard are authors of popular React Canvas.
- Khan Academy has several GitHub projects using React.
- Mapbox React Native module for creating custom maps.
- Netflix chooses React for startup speed of UI, runtime performance, and modularity it offers.
- Uber has several React GitHub projects including react-vis, a charting component library for React.

Technology stack

ThoughtWorks Technology Radar ranks technologies based on Adopt > Trial > Assess > Hold relative ranking. This is based on their own usage of these technologies across projects for leading enterprises globally. In terms of speed of decision making about your own technology stack, this is one tool that proves very helpful.

React Speed Coding will be addressing following technologies, platforms, techniques, frameworks, and tools.



ReactSpeed Technology Stack

ES6 (Adopt). JavaScript ECMAScript 6 is right at the top of the Radar list of languages and frameworks. We cover important concepts relevant for coding React in ES6.

React (Adopt). React is a close second on the Radar. Of course this book is all about React so we are well covered.

Redux and Flux (Trial). Redux is a new entrant on the Radar. We are dedicating an entire chapter and a relatively complex app for decent coverage of this important technology in React ecosystem. Flux is an architectural style recommended for React. Redux evolves ideas of Flux, avoiding its complexity, according to the author of Redux.

React Native (Trial). Another entry high on the Radar from React ecosystem. We are covering Flexbox which is one of the key technologies in React Native stack. Of course React and Redux make up the mix.

GraphQL (Assess). Another up and coming technology in React stack. GraphQL is an alternative to REST protocol. Goes hand in hand with Relay, another technology from the Facebook camp.

Immutable.js (Assess). Yet another Facebook open source project. Goes well with Redux.

Recharts (Assess). Integrates D3 charts and React. We will implement samples using Rumble Charts, a popular alternative, in this book.

Browsersync (Trial). Browsersync is a great time-saver for multi-devices testing of mobile-web hybrid apps. React Speed Coding implements Browsersync + Webpack + Hot Reloading. So if you make any changes in your JSX, these should update on all devices on saving the changes. While maintaining your current UI state. Isn't this awesome!

GitUp (Trial). Graphical tool complementing Git workflow. We are find this tool useful for going back in time and revising commit logs for instance.

Webpack (Trial). We are implementing your React developer workflow using Webpack. Two chapters are dedicated to get you started with Webpack and help you production optimize the workflow.

Serverless Architecture (Assess). We are implementing serverless architecture using Firebase. Another technology worth evaluating is AWS Lambda, though it may not be in scope for this book.

Measuring speed

So it will be nice to define some baseline measurements of speed and see if we can improve these as we go through the book.

Website Performance. Google PageSpeed defines 25+ criteria for website performance as relative measures or percentile scores compared with rest of the Web. For example *Enable Gzip Compression* is 88% as recommended baseline.

As on May 9, 2016 the ReactSpeed.com website is evaluating grade A (93% average) on PageSpeed score, grade B (82% average) on YSlow score, 2.1s page load time, 834KB total page size, with 26 requests. View GTMetrix ReactSpeed.com report here[16].

As on Aug 9, 2016 while the React Speed app has more than 3,000 lines of code, we improve our page load time to 1.3s, 438KB page size, and only 13 requests back to the servers. Our page speed score improves to 95%, and YSlow score is 86%.

Load Impact (Radar Trial). Online load testing tool. We are using this tool to perform concurrent user load tests on ReactSpeed.com website.

As on May 9, 2016 with 20+ custom React components live on ReactSpeed website, we are recording faster than 200ms load time for our website for 25 concurrent users. That translates to handling approximately 2,50,000 monthly visitors. Excellent! View results snapshot here[17].

Build and Deploy Time. How long does it take to run the developer workflow.

As on May 9, 2016 our development server continuously builds and updates our app as we save our working code. Production build takes 5.3s with around 250 hidden modules. We deploy 44 files to Firebase several times during a day.

Time to Release. How long does it take to ship new features to production.

Since start of ReactSpeed project we have closed 150 production commits to GitHub over a 30 day period. Our peak is 40 commits during week of April 10.

Production Payload. How optimized are our production assets.

As on May 9, 2016 our CSS library is 4.7KB Gzip, 21KB minified with 25+ style modules. App JS bundle is 42KB minified. Vendor JS bundle is 192KB minified. HTML is 3KB.

[16] https://gtmetrix.com/reports/reactspeed.com/Mn36KHic

[17] https://app.loadimpact.com/load-test/39d3b00c-d9fa-4056-8606-7ebe9026e161?charts=type%3D1%3Bsid%3D___li_clients_active%3A1%3B%3Btype%3D1%3Bsid%3D___li_user_load_time%3A1&large-charts=type%3D1%3Bsid%3D___li_clients_active%3A1%3B%3Btype%3D1%3Bsid%3D___li_user_load_time%3A1

Time to fix issues. How long does it take to fix issues in code. Code issues can be of several types including compliance with coding guidelines and best practices, logical bugs, usability issues, performance issues, among others.

As on May 10, 2016 it took us 6 hours to resolve 300+ issues down to 3 open issues using ESLint integration with Atom editor and Webpack. The issues ranged from coding best practices to refactoring requirements as per React coding patterns.

Continuos production build workflow. This can be measured by number of commands or developer actions required to complete one production ready build. Alternatively how automated is this lifecycle.

NPM for all the things. Can be measured based on number of project dependencies that are updated from NPM or popular managed repositories and CDNs. This is Trial stage at ThoughtWorks Technology Radar.

Static code analytics. This can be measured for number of lint warnings or errors. Complexity analysis of JavaScript code can be included apart from other static code analytics.

Why learn React comparing with Angular

One of the most important decisions modern app developers make is choosing the right front-end framework for their technology stack. Most popular question relates to choosing React over Angular 2.

We recognize that React is at its core a simpler library representing the View pattern, while Angular 2 is a complete framework offering Model, View, and Controller architectural patterns.

React is actually an ecosystem of well designed and highly popular libraries, open sourced mostly by Facebook (React, React Native, Immutable, Flow, Relay, GraphQL), other leading developers (Redux, React Material UI, React Router), and industry leaders (Flipboard React Canvas, Airbnb Enzyme).

React + Redux for instance offer the Model (Store), View (React, Actions), and Controller (Reducers) pattern to compare on equal grounds with the Angular 2 stack.

Why learn Angular 2? It is like learning Yoga, from one Guru, in a large group.

If you are in a **large** team, Angular will be your choice to get everyone on the same page, faster, at scale.

Contrary to popular opinion on the subject, we think Angular 2 is **faster** to learn when compared to React for the same goals, simply because you are making fewer "first-time-learner or developer" decisions along your journey.

- Angular2 and TypeScript are opinionated,
- Most documentation is "single version of truth" from one source (Google, the authors of Angular and Microsoft, the authors of TypeScript),
- There is mostly "one Angular/TypeScript way" of doing things, so fewer decisions to make along your learning and development journey,
- Angular 2 API and TypeScript language are well documented.
- Official samples are up to date with latest changes in the API, well mostly.
- Development boilerplates or starters are fairly mature, some like Angular Universal are backed by Google/Angular core team.
- The development and build tool-chain is mostly addressed by Angular/-TypeScript, and few popular starter projects.

Why learn React? It is like doing cross-training, with multiple experts, at multiple locations.

If you are in a **lean** team or a single developer-designer-architect, "the React way" may be more fun. Learning React is more **rewarding** in the long run.

As you are making “hopefully informed” decisions all along your learning and development journey, you become a more thorough developer, designer, architect in the long haul. If you make your decisions by evaluating pros-cons of architectural and design patterns, you are becoming a better developer.

React is opinionated for fewer core concepts like one-way-data-binding and offers sensible workarounds even for that.

- There are many ways to develop in React starting from how to define React components, how to create React build pipeline, which frameworks to integrate with, how to wire up a backend, the list goes on.
- You will learn from multiple sources and authors, not just Facebook/Instagram, the authors of React. Having these multiple perspectives will give you stronger real-world decision making muscles!
- Facebook and the React community is very driven by app performance patterns. Most core React concepts are centered around creating high performance code.
- The React community is more component driven. You will most likely find more reusable code.
- The community is also increasingly driven by “Developer Experience”. Writing beautiful code, writing more manageable code, writing readable code, and tools that make developer’s experience more fun and visual. See kadirahq/react-storybook as an example.

In our experience learning both and going back and forth helps. Programming design patterns remain the same. Syntactical sugar changes. Learning one, reinforces the other.

Shared learning path between Angular 2 and React

JavaScript. JavaScript (ES5 and ES6) is fundamental. TypeScript transpiles to JavaScript. React-JSX-Babel tooling transpiles to JavaScript.

CSS3. You cannot do serious front-end coding without it.

HTML5. It is obvious, but extend your knowledge on concepts like Offline Storage and Device Access, best practice starters like HTML5 boilerplate.

Webpack. Modern day packaging, module bundling, build pipeline automation tooling.

Design Patterns and Object Oriented principles. Composition, Inheritance, Singletons, Pure Functions, Immutability, and many others are core concepts helping you in doing good development in general.

Algorithms and data structures. Serious development cannot be done without using these in a good measure.

Backend as a Service. Firebase, AWS Lamba, among others.

Microservices and REST/APIs. No modern app is built in isolation these days.

So, here is a learning path if you want to go beyond React. Learn React first, build some reusable components, learn the component design workflow. Learn Angular 2 next, try reusing your component design and above mentioned shared learning here. Maybe round off your knowledge by learning Meteor (more opinionated with best practice patterns for speed coding and performance) and integrating React and Angular, replacing Meteor's Blaze.

Setup React Webpack

You will learn in this chapter how to setup React development environment starting from scratch. By the end of this chapter we will have a starter boilerplate to develop React apps.

We will cover following topics in this chapter.

- How to install Node.js and use Node Version Manager
- Setup package.json to manage your NPM dependencies
- Quick access companion code for this book using Github
- Install starter dependencies for React, Webpack, and Babel
- Create Webpack configuration for development pipeline automation
- Write a simple React app to run your Webpack setup

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

Preview complete demo website[18] hosted on Firebase, as you code this app by the end of this book.

View current chapter demo[19] of the app we build in this chapter.

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c01 origin/c01-setup-react-webpack
npm install
npm start
```

[18] https://reactspeed-c3e1c.firebaseapp.com

[19] https://manavsehgal.github.io/react-speed-demos/c01/

Installing Node.js

You will need Node.js to get started with React. Your Mac OS comes with Node pre-installed. However you may want to use the latest stable release.

Check you node release using `node -v` command.

We recommend installing or upgrading Node using Node Version Manager (NVM). Their Github repo[20] documents install and usage.

To install NVM:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.0/inst\
all.sh | bash
```

Now you can install a Node release by choosing one from Node releases[21] page.

The command `nvm install 5.10.1` installs a stable release for us.

One of the advantages of using NVM is you can switch between multiple node releases you may have on your system.

Here's what our terminal looks like when using `nvm ls` to list installed node releases.

```
v4.2.3
v5.3.0
v5.10.0
->      v5.10.1
system
default -> 5.3.0 (-> v5.3.0)
node -> stable (-> v5.10.1) (default)
stable -> 5.10 (-> v5.10.1) (default)
iojs -> iojs- (-> system) (default)
```

Using `nvm use x.y.z` command we can switch to `x.y.z` installed node release.

[20] https://github.com/creationix/nvm

[21] https://nodejs.org/en/download/releases/

Setting up package.json

You will require `package.json` to manage your NPM dependencies and scripts.

Create a new one using `npm init` command, selecting defaults where uncertain.

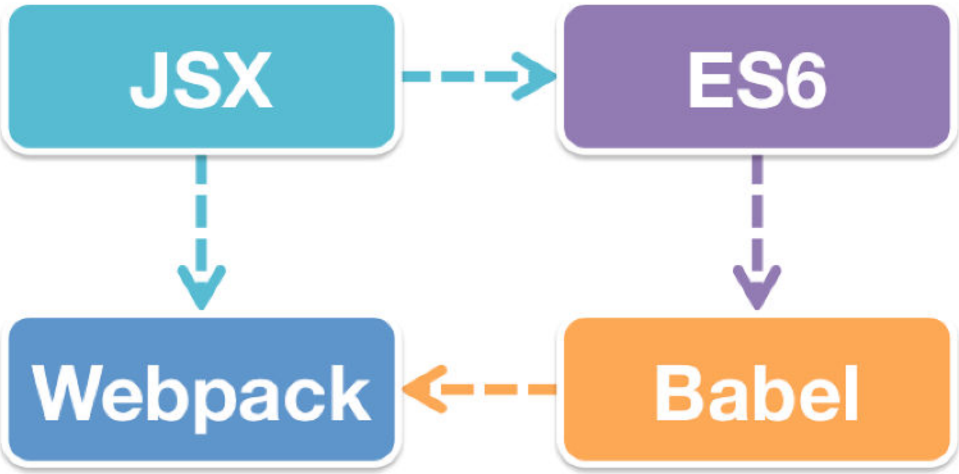
This is what our `package.json` looks like as we start off. Note that we added the `private` flag to avoid accidental publishing of the project to NPM repo, and also to stop any warnings for missing flags like project repo.

```json
{
  "name": "react-speed-book",
  "version": "1.0.0",
  "description": "Companion code for React Speed Coding book",
  "main": "index.js",
  "private": true,
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/manavsehgal/react-speed-book.git"
  },
  "author": "Manav Sehgal",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/manavsehgal/react-speed-book/issues"
  },
  "homepage": "https://github.com/manavsehgal/react-speed-book#readme"
}
```

The dependencies section will start showing up as we add npm dependencies.

Installing starter dependencies

Before we start writing our React app we need to install starter dependencies for our development environment. React uses JSX as the XML-like syntax extension over JavaScript to specify component tree structure, data flow, and event handlers. JSX is processed by Webpack module bundler using specific loaders or convertors.



React Dependency Stack

React recommends JavaScript ES6 for latest features and best practices. ES6 needs Babel for compiling to ES5 and maintain browser compatibility. Babel integrates with Webpack to stitch it all together for our app.

React and React DOM

React is available via NPM and this is the recommended way of using React in a project. React core is available in the `react` package. The `react-dom` package targets browser DOM for rendering React. React enables several targets including iOS, Android for rendering React apps.

```
npm install --save react
npm install --save react-dom
```

Webpack

Webpack is used for module packaging, development, and production pipeline automation. We will use `webpack-dev-server` during development and `webpack` to create production builds.

```
npm install --save-dev webpack
npm install --save-dev webpack-dev-server
```

HTML generation

You can add functionality to Webpack using plugins. We will use automatic HTML generation plugins for creating `index.html` for our app.

The `html-webpack-plugin` webpack plugin will refer to template configured within webpack configuration to generate our `index.html` file.

```
npm install --save-dev html-webpack-plugin
```

Loaders

Webpack requires loaders to process specific file types. CSS loader resolves `@import` interpreting these as `require()` statements. Style loader turns CSS into JS modules that inject <style> tags. JSON loader enables us to import local JSON files when using data fixtures during prototyping our app.

```
npm install --save-dev css-loader
npm install --save-dev style-loader
npm install --save-dev json-loader
```

PostCSS and Normalize CSS

PostCSS loader adds CSS post-processing capabilities to our app. This includes Sass like capabilities including CSS variables and mixins using `precss` and automatic vendor prefixes using the `autoprefixer` plugin for PostCSS.

Normalize CSS offers sensible resets for most use cases. Most CSS frameworks include it.

The `postcss-easy-import` plugin enables processing `@import` statements with rules defined in webpack configuration.

```
npm install --save-dev postcss-loader
npm install --save-dev precss
npm install --save-dev autoprefixer
npm install --save-dev normalize.css
npm install --save-dev postcss-easy-import
```

Babel

Babel compiles React JSX and ES6 to ES5 JavaScript. We need `babel-loader` as Webpack Babel loader for JSX file types.

Hot loading using `babel-preset-react-hmre` makes your browser update automatically when there are changes to code, without losing current state of your app.

ES6 support requires `babel-preset-es2015` Babel preset.

```
npm install --save-dev babel-core
npm install --save-dev babel-loader
npm install --save-dev babel-preset-es2015
npm install --save-dev babel-preset-react
npm install --save-dev babel-preset-react-hmre
```

Presets in Babel

Presets in Babel are a collection of plugins as npm dependencies which help transform source to target code using Babel. For full list of such plugins[a] visit Babel's website.

[a] http://babeljs.io/docs/plugins/

Configure Babel .babelrc

Babel configuration is specified in `.babelrc` file. React Hot Loading is required only during development.

```
{
  "presets": ["react", "es2015"],
  "env": {
    "development": {
      "presets": ["react-hmre"]
    }
  }
}
```

Dependencies in package.json

The dependencies within package.json now lists all the installed dependencies for our app so far.

```
"dependencies": {
  "react": "^15.3.0",
  "react-dom": "^15.3.0"
},
"devDependencies": {
  "autoprefixer": "^6.4.0",
  "babel-core": "^6.13.2",
  "babel-loader": "^6.2.4",
  "babel-preset-es2015": "^6.13.2",
  "babel-preset-react": "^6.11.1",
  "babel-preset-react-hmre": "^1.1.1",
  "copy-webpack-plugin": "^3.0.1",
  "css-loader": "^0.23.1",
  "html-webpack-plugin": "^2.22.0",
  "json-loader": "^0.5.4",
  "normalize.css": "^4.2.0",
  "postcss-easy-import": "^1.0.1",
  "postcss-loader": "^0.9.1",
  "precss": "^1.4.0",
  "style-loader": "^0.13.1",
  "webpack": "^1.13.1",
  "webpack-dev-server": "^1.14.1"
}
```

Configure Webpack in webpack.config.js

Webpack configuration drives your development pipeline, so this is a really important file to understand. We will split various sections of the config file to aid step-by-step learning.

Initialization

To start off, you need to initialize the config file with dependencies. These include webpack itself, an HTML generation plugin, a webpack plugin to copy folders from development to build target, and Node path library for initializing default paths.

Next we initialize the default paths. We also configure defaults for our HOST and PORT configuration.

```
// Initialization
const webpack = require('webpack');

// File ops
const HtmlWebpackPlugin = require('html-webpack-plugin');

// Folder ops
const CopyWebpackPlugin = require('copy-webpack-plugin');
const path = require('path');

// PostCSS support
const postcssImport = require('postcss-easy-import');
const precss = require('precss');
const autoprefixer = require('autoprefixer');

// Constants
const APP = path.join(__dirname, 'app');
const BUILD = path.join(__dirname, 'build');
const STYLE = path.join(__dirname, 'app/style.css');
const PUBLIC = path.join(__dirname, 'app/public');
const TEMPLATE = path.join(__dirname, 'app/templates/index.html');
const NODE_MODULES = path.join(__dirname, 'node_modules');
const HOST = process.env.HOST || 'localhost';
const PORT = process.env.PORT || 8080;
```

- APP path is used to specify the app entry point, location of the root component
- BUILD path specifies the target folder where production compiled app files will be pushed
- STYLES path indicates the root CSS file that imports other styles
- PUBLIC path is a folder in development that is copied as-is to root of BUILD, used for storing web server root files like robots.txt and favicon.ico, among others
- TEMPLATE specifies the template used by html-webpack-plugin to generate the index.html file
- NODE_MODULES path is required when including assets directly from installed NPM packages

Entry points and extensions

Next section defines your app entry, build output, and the extensions which will resolve automatically.

```
module.exports = {
  // Paths and extensions
  entry: {
    app: APP,
    style: STYLE
  },
  output: {
    path: BUILD,
    filename: '[name].js'
  },
  resolve: {
    extensions: ['', '.js', '.jsx', '.css']
  },
```

Loaders

We follow this by defining the loaders for processing various file types used within our app.

React components will use `.jsx` extension. The React JSX and ES6 is compiled by Babel to ES5 using the babel webpack loader. We use the `cacheDirectory` parameter to improve repeat development compilation time.

The `include` key-value pair indicates where webpack will look for these files.

CSS processing is piped among `style` for injecting CSS style tag, `css` for processing import paths, and `postcss` for using various plugins on the CSS itself.

We need JSON processing much later in the book to enable loading of local JSON files storing data fixtures (sample data) for our app.

```
module: {
  loaders: [
    {
      test: /\.jsx?$/,
      loaders: ['babel?cacheDirectory'],
      include: APP
    },
    {
      test: /\.css$/,
      loaders: ['style', 'css', 'postcss'],
      include: [APP, NODE_MODULES]
    },
    {
      test: /\.json$/,
      loader: 'json',
      include: [APP, NODE_MODULES]
    }
  ]
},
```

PostCSS plugins configuration

PostCSS requires further configuration to handle how PostCSS plugins behave.

Now we add `postcssImport` which enables Webpack build to process `@import` even from `node_modules` directory directly.

Sequence matters as this is the order of execution for the PostCSS plugins. Process imports > compile Sass like features to CSS > add vendor prefixes.

```
// Configure PostCSS plugins
postcss: function processPostcss(webpack) {  // eslint-disable-line no\
-shadow
  return [
    postcssImport({
      addDependencyTo: webpack
    }),
    precss,
    autoprefixer({ browsers: ['last 2 versions'] })
  ];
},
```

Configure webpack-dev-server

Now that we have loaders configured, let us add settings for our development server.

Source maps are used for debugging information.

The `devServer` settings are picked up by `webpack-dev-server` as it runs.

The `historyApiFallback` enables local browser to be able to handle direct access to route URLs in single page apps.

The `hot` flag enables hot reloading. Using the `inline` flag a small webpack-dev-server client entry is added to the bundle which refresh the page on change.

The `progress` and `stats` flags indicate how webpack reports compilation progress and errors.

```
// Source maps used for debugging information
devtool: 'eval-source-map',
// webpack-dev-server configuration
devServer: {
  historyApiFallback: true,
  hot: true,
  progress: true,

  stats: 'errors-only',

  host: HOST,
  port: PORT,

  // CopyWebpackPlugin: This is required for webpack-dev-server.
  // The path should be an absolute path to your build destination.
  outputPath: BUILD
},
```

Plugins

We now wrap up by adding plugins needed during our development.

This section specifies the compilation workflow. First we use the `DefinePlugin` to define the NODE_ENV variable.

We then activate the Hot Reloading plugin to refresh browser with any app changes.

Then we generate any HTML as configured in the `HtmlWebpackPlugin` plugin.

```
plugins: [
  // Required to inject NODE_ENV within React app.
  new webpack.DefinePlugin({
    'process.env': {
      'NODE_ENV': JSON.stringify('development') // eslint-disable-line\
 quote-props
    }
  }),
  new webpack.HotModuleReplacementPlugin(),
  new CopyWebpackPlugin([
    { from: PUBLIC, to: BUILD }
  ],
    {
      ignore: [
        // Doesn't copy Mac storage system files
        '.DS_Store'
      ]
    }
  ),
  new HtmlWebpackPlugin({
    template: TEMPLATE,
    // JS placed at the bottom of the body element
    inject: 'body'
  })
]};
```

HTML webpack template for index.html

We can now add a custom template to generate `index.html` using the `HtmlWebpackPlugin` plugin.

This enables us to add `viewport` tag to support mobile responsive scaling of our app. We also add icons for mobile devices. Following best practices from HTML5 Boilerplate, we add `html5shiv` to handle IE9 and upgrade warning for IE8 users.

```
<!DOCTYPE html>
<html class="no-js" lang="">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <title>React Speed Coding</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width, initial-scale=1\
">

    <link rel="icon" href="/favicon.ico" />
    <!--[if lt IE 9]>
        <script src="//html5shiv.googlecode.com/svn/trunk/html5.js"></\
script>
        <script>window.html5 || document.write('<script src="js/html5s\
hiv.js"><\/script>')</script>
    <![endif]-->
  </head>
  <body>
    <!--[if lt IE 8]>
        <p class="browserupgrade">You are using an <strong>outdated</s\
trong> browser. Please <a href="http://browsehappy.com/">upgrade your \
browser</a> to improve your experience.</p>
    <![endif]-->

    <div id="app"></div>
    <!-- Google Analytics: change UA-XXXXX-X to be your site's ID. -->
  </body>
</html>
```

Configuring startup scripts in package.json

We can configure startup scripts in `package.json` to speed up our development even further.

Before we do that, let us create a copy of `webpack.config.js` as `webpack.prod.config.js` intended for production build version of our webpack configuration. We will only make one change in the production configuration. Changing `NODE_ENV` in `plugins` section from `development` to `production` value.

Now `npm start` will run webpack-dev-server using `inline` mode which works well with hot reloading browser when app changes without manually refreshing the browser.

Running `npm run build` will use webpack to create our production compiled build.

```
"scripts": {
  "start": "NODE_ENV=development webpack-dev-server --inline",
  "build": "NODE_ENV=production webpack --config webpack.prod.config.j\
s"
},
```

The webpack-dev-server will pick up the `webpack.config.js` file configuration by default.

Run webpack setup

index.js

Add `index.js` in the root of our development folder.

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  <h1>App running in {process.env.NODE_ENV} mode</h1>,
  document.getElementById('app')
);
```

We will learn each aspect of a React component in the next chapter. For right now we are writing a minimal React app to test our Webpack configuration.

style.css

We also import the Normalize CSS library in our main style entry file.

```
@import 'normalize.css';
```

public/ folder

To test how are public folder is copied over to build, let us add `favicon.ico` file to public folder in root of our development folder.

Running webpack

We can now run the development server using `npm start` command in the Terminal. Now open your browser to localhost URL that your webpack dev server suggests.

The browser app displays message that you are running in development mode. Try changing the message text and hit save. Your browser will refresh automatically.

You can also build your app to serve it using any web server.

First add a file server like so.

```
npm install -g serve
```

Now build and serve.

```
npm run build
serve build
```

As you open your localhost URL with port suggested by the file server, you will note that the message now displays that you are running in production mode.

The `build` folder lists following files created in our webpack build.

```
app.js         <== App entry point
favicon.ico    <== Copied as-is from public/ folder
index.html     <== Generated by HTML webpack plugin
style.js       <== Contains our styles
```

Congratulations… You just built one of the most modern development environments on the planet!

ES6 React Guide

This chapter guides you through important React concepts and ES6 features for speeding up your React learning and development journey.

You will learn following concepts in this chapter.

- How to define a React component using ES6 syntax
- What are modules and how to import and export React components
- Why we need constructors
- How components talk to each other and the UI using events, props, and state
- Importance of stateless components
- Using React Chrome Extension to inspect your component hierarchy at runtime

For Speed Coding in React, ES6 is essential. Not only does it reduce the amount of code you end up writing, ES6 also introduces language patterns for making your app better designed, more stable, and performant.

Let us create a Hello World app to understand the React and ES6 features together. Subsequent chapters introduce more React and ES6 features based on the context of samples written for the chapter. This spreads your learning journey as you apply these concepts.

Code Along. You can clone the source for this entire book, change to app directory, checkout just the code for this chapter, install and start the app to launch the local version in your default browser.

Preview complete demo website[22] hosted on Firebase, as you code this app by the end of this book.

View current chapter demo[23] of the app we build in this chapter.

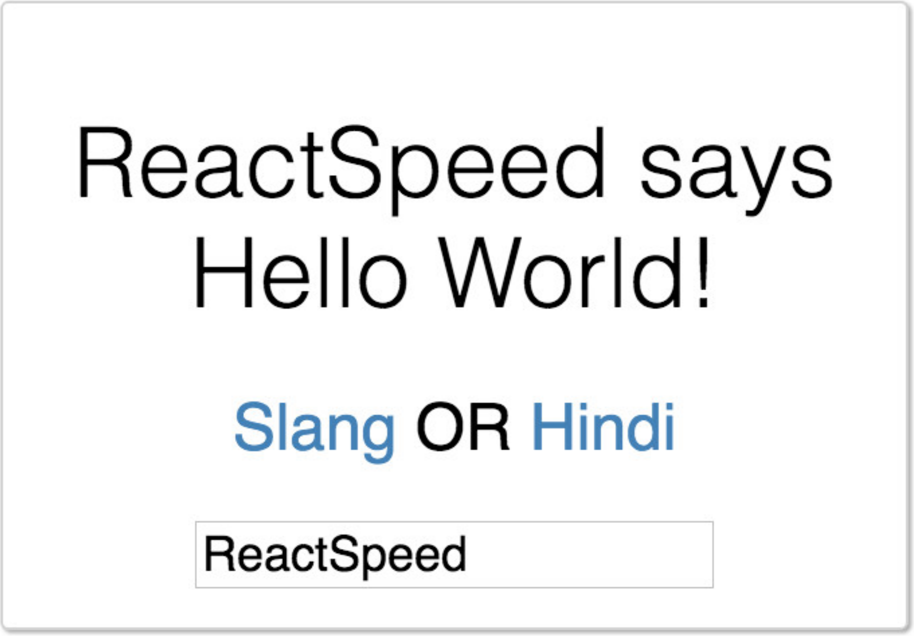
[22] https://reactspeed-c3e1c.firebaseapp.com

[23] https://manavsehgal.github.io/react-speed-demos/c02/

```
git clone https://github.com/manavsehgal/react-speed-book.git
cd react-speed-book
git checkout -b c02 origin/c02-es6-react-guide
npm install
npm start
```

Hello World React

We will write a Hello World app with three custom components. A component to contain our app, a Hello component to render the greeting message, and a World component to handle user events like changing greeting or adding a name.



ReactSpeed says Hello World!

ReactSpeed

Hello World app in your browser

Component composition and naming

React apps follow component based development. So understanding how components are composed, how they relate to each other, simplifies several aspects of your React learning path. This includes understanding how React app folders and files are organized.

Ownership vs. Parent-Child

According to Facebook React docs, "It's important to draw a distinction between the owner-ownee relationship and the parent-child relationship. The owner-ownee relationship is specific to React, while the parent-child relationship is simply the one you know and love from the DOM."

Component Hierarchy. React has owner components which render or set properties on other components. This ownership continues across all components building a tree-like hierarchy.

Root Component. In case of our Hello World ap, `index.jsx` represents the root component (one that does not have an owner). The root component owns and renders `World` component. The `World` component is owner of `Hello` component.

Component File Naming. Root component inside a folder is named `index.jsx` and the component name takes on the name of the folder, `app` in this case. Components other than root are named same as the component class names, including PascalCase. Refer naming conventions[24] in Airbnb style guide.

[24] http://webpack.github.io/docs/code-splitting.html#es6-modules

Files and folder structure

This is the files and folder hierarchy you will create by the end of this chapter.

```
.babelrc                  <== Babel configuration
package.json              <== NPM configuration
webpack.config.js         <== Webpack development configuration
webpack.prod.config.js    <== Webpack production configuration
app/                      <== React app root
  index.jsx               <== Root app component, app entry point
  style.css               <== Style entry point @import from styles/
  styles/                 <== Style partials
    base/
      elements.css        <== Base HTML element styles
    components/
      world.css           <== World component specific styles
  public/                 <== Copied as-is to root of build folder
    favicon.ico
  components/             <== Components folder
    Hello.jsx             <== Custom component
    World.jsx             <== Custom component
  templates/
    index.html            <== Template used by HTML webpack plugin
build/                    <== Webpack generated build folder
node_modules/             <== NPM dependencies installed here
```

Root component index.jsx

We start by writing the entry point to our HelloWorld React app. We replace the `index.js` created in the last chapter with a new one we start from scratch.

Module import

In React each component is typically defined in a single file, also known as a module. Any dependencies are imported using `import` statement. The statement specifies name of the exported component, constant, or function from the dependencies.

Dependencies are then used within a component. Dependencies are also used as rendered components within an owner component. Like in our case `World` component is imported by `app` root component before it is rendered.

```
import React from 'react';
import ReactDOM from 'react-dom';
import World from './components/World.jsx';

ReactDOM.render(
  <World />,
  document.getElementById('app')
);
```

Thinking in modules is central to how Webpack bundles your code and traces dependencies while creating chunks. However, Webpack 1.x does not natively support ES6 modules, though this is on their 2.x roadmap. This is where Babel steps in. Read more about Webpack ES6 support[25] on the Webpack official docs.

The `ReactDOM.render` method uses React library targeting DOM output to identify a DOM element id as target for rendering our entire app component hierarchy, starting with the `World` component.

[25] http://webpack.github.io/docs/code-splitting.html#es6-modules

World.jsx component

Now we write the `World` component which renders the `Hello` component with a message and handles user inputs.

Class definition

We start by importing Hello component which we will render in World component.

A module exports a component using `export default` keywords. There can only be one such export. While importing such components we do not need to use the { } braces.

All React components extend React.Component. This enables our components to have access to React lifecycle methods and other features.

```
import React from 'react';
import Hello from './Hello.jsx';

export default class World extends React.Component {
```

Constructor

The constructor of our `World` component highlights three most important features of how components talk to each other and the user.

State. Changing UI or internal state of a component is maintained using `this.state` object. When state changes, rendered markup is updated by re-invoking the `render()` method.

Props. Properties are the mechanism to pass data from owner to rendered components.

Events. Methods or event handlers are bound to UI events (like onClick) to perform actions when the event takes place.

Constructor is called when component is created, so it is the right place for the following three objectives.

1. Using `super(props)` keyword to make the `props` object available to React.Component methods.
2. Setting initial component state. Using `this.state` object. One can set state to default values or properties passed from owner component accessing `props` object.
3. Binding event handlers to `this` context of the component class. Using `bind(this)` method.

```
constructor(props) {
  super(props);
  this.state = {
    currentGreeting: props.greet,
    value: 'ReactSpeed'
  };
  this.slangGreet = this.slangGreet.bind(this);
  this.hindiGreet = this.hindiGreet.bind(this);
  this.handleNameChange = this.handleNameChange.bind(this);
}
```

Event Handlers and setState

Components can exist with no events defined. In our HelloWorld app we define three events.

The slangGreet event handles user click on Slang link to change the greeting message accordingly. Likewise the hindiGreet does so with the Hindi greeting message.

The handleNameChange method handles user input in the input text box to change the name of greeting sender in the message. The `value` state is available to React controlled components as explained few sections later in this chapter.

We use `setState` method to change state within React components. This in turn re-renders the component UI calling the `render` method.

```
slangGreet() {
  this.setState({ currentGreeting: 'Yo!' });
}

hindiGreet() {
  this.setState({ currentGreeting: 'Namaste' });
}

handleNameChange(event) {
  this.setState({ value: event.target.value });
}
```

JSX and the render method

Let us now write the render method for World component. React component is rendered using JSX syntax. JSX is HTML-like syntax where the nodes are actually native React components providing same functionality as the HTML DOM equivalents. So `<div>`, `<h2>`, `<a>`, and others used in the JSX code are all native React components.

```
render() {
  const renderGreeting = this.state.value
    ? `${this.state.value} says ${this.state.currentGreeting}`
    : this.state.currentGreeting;
  return (
    <div className="World-card">
      <Hello greet={renderGreeting} message="World!" />
      <h2>
        <a onClick={this.slangGreet}>Slang</a>
        &nbsp;OR&nbsp;
        <a onClick={this.hindiGreet}>Hindi</a>
      </h2>
      <input
        type="text" value={this.state.value}
        placeholder="Enter a name"
        onChange={this.handleNameChange}
      />
    </div>
  );
}}
```

In the next four sections we will break down the render method JSX to understand more ES6 React concepts.

Template literals and ternary conditionals

We are using ES6 template literals[26] as strings with back-ticks. Template literals allow embedded expressions using the `${expression}` syntax.

[26] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Template_literals

We are also using the JavaScript ternary conditional expression to set the JSX value of `renderGreeting` based on `value` state.

```
const renderGreeting = this.state.value
  ? `${this.state.value} says ${this.state.currentGreeting}`
  : this.state.currentGreeting;
```

Properties

Properties are used for passing input data from Root/Owner component to Child. Owner defines a property=value which is used within rendered component. The "Owner-ownee" relationship can exist without property passing, as in owner simply rendering a component.

> Treat props as immutable. You can use props to set state within event handlers using `this.setState` method or state definition in constructor using this.state object. Props, state, and event bindings can be defined in the constructor.

```
<Hello greet={renderGreeting} message="World!" />
```

Event UI binding

Event UI binding is done by passing a prop named after the event `onClick` in this case and the bound event handler method.

```
<a onClick={this.slangGreet}>Slang</a>
&nbsp;OR&nbsp;
<a onClick={this.hindiGreet}>Hindi</a>
```

Controlled components

Current UI state changes as user clicks on greeting language links. We are also processing `input` data using `this.state.value` provided by React. The input control used in this example does not maintain its own state. It is known as *Controlled Component* as opposed to *Uncontrolled Component* if the value property is not used. Uncontrolled components manage their own state. Read more about handling forms[27] at Facebook React documentation.

```
<input
  type="text" value={this.state.value}
  placeholder="Enter a name"
  onChange={this.handleNameChange}
/>
```

[27] https://facebook.github.io/react/docs/forms.html

PropTypes and defaultProps

At the end of the class definition we define default properties and property types on the component constructor.

The `propTypes` are used in property validation during development to throw warnings in the Browser's JavaScript console, if your code is not meeting the validation criteria. Read more on Prop Validation[28] in Facebook post that lists various types, including custom validations.

```
World.propTypes = {
  greet: React.PropTypes.string.isRequired
};

World.defaultProps = {
  greet: 'Hello'
};
```

So if you change the value of greet to any number, the app will run, however you will see following warning in your browser console.

> Warning: Failed propType: Invalid prop `greet` of type `number` supplied to `World`, expected `string`.

[28] https://facebook.github.io/react/docs/reusable-components.html#prop-validation

ES7 Property Initializers

Property initializers[29] are an ES7 Stage 1 proposed feature.

In the prior section we were using class properties by defining the `World.propTypes` and `World.defaultProps` outside of the class definition on the component constructor.

Now using `babel-plugin-transform-class-properties` we can bring these within the class definition.

Install the Babel plugin supporting this ES7 transform.

```
npm install --save-dev babel-plugin-transform-class-properties
```

Update `.babelrc` with this new plugin.

```
{
  "presets": ["react", "es2015"],
  "env": {
    "development": {
      "presets": ["react-hmre"]
    }
  },
  "plugins": ["transform-class-properties"]
}
```

Next we update our class definition like so.

[29] https://github.com/jeffmo/es-class-fields-and-static-properties

```
export default class World extends React.Component {
  static propTypes = {
    greet: React.PropTypes.string.isRequired
  }

  static defaultProps = {
    greet: 'Hello'
  }

  constructor(props) {
```

Note the use of `static` statement before initializing `propTypes` and `defaultProps` in our class definition.

Complete World.jsx listing

We can further reduce few lines of code from our World component by importing `PropTypes` and `Component` from React core.

Here is the complete World component listing.

```
import React, { PropTypes, Component} from 'react';
import Hello from './Hello.jsx';

export default class World extends Component {
  static propTypes = {
    greet: PropTypes.string.isRequired
  }

  static defaultProps = {
    greet: 'Hello'
  }

  constructor(props) {
    super(props);
    this.state = {
      currentGreeting: props.greet,
      value: 'ReactSpeed'
    };
    this.slangGreet = this.slangGreet.bind(this);
    this.hindiGreet = this.hindiGreet.bind(this);
    this.handleNameChange = this.handleNameChange.bind(this);
  }

  slangGreet() {
    this.setState({ currentGreeting: 'Yo!' });
  }

  hindiGreet() {
    this.setState({ currentGreeting: 'Namaste' });
  }

  handleNameChange(event) {
    this.setState({ value: event.target.value });
  }

  render() {
    const renderGreeting = this.state.value
      ? `${this.state.value} says ${this.state.currentGreeting}`
      : this.state.currentGreeting;
    return (
      <div className="World-card">
        <Hello greet={renderGreeting} message="World!" />
        <h2>
          <a onClick={this.slangGreet}>Slang</a>
          &nbsp;OR&nbsp;
          <a onClick={this.hindiGreet}>Hindi</a>
        </h2>
        <input
          type="text" value={this.state.value}
          placeholder="Enter a name"
          onChange={this.handleNameChange}
        />
      </div>
    );
  }
}
```

Hello.jsx stateless component

The `Hello` component renders Hello World message based on how `World` component calls it and current UI state.

Our `Hello` component is stateless. It does not define or change any state.

Why stateless components are important

According to Facebook, "In an ideal world, most of your components would be stateless functions because in the future we'll also be able to make performance optimizations specific to these components by avoiding unnecessary checks and memory allocations. This is the recommended pattern, when possible."

We are using arrow functions[30] which are ES6 shorthand for writing functions.

```
import React from 'react';

const Hello = ({ greet, message }) => (
  <h1>{greet} {message}</h1>
);

Hello.propTypes = {
  greet: React.PropTypes.string.isRequired,
  message: React.PropTypes.string.isRequired
};

export default Hello;
```

[30] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Component styles in world.css

We can add some styles to our app. The `/app/styles/components/world.css` path is used to create style for our World component.

```
.World-card {
  background: white;
  border: 1px solid lightgrey;
  border-radius: 3px;
  box-shadow: 1px 1px 1px 0 darkgrey;
  padding: 0.8em 1em;
  width: 300px;
  margin: 10px;
  text-align: center;
}
```

Base styles in element.css

We also add some shared styles in `/app/styles/base/element.css` file.

```
html {
  height: 100%;
  color: black;
  font-family: 'Open Sans', sans-serif;
  font-size: 18px;
  font-weight: 400;
}

h1 {
  font-weight: 300;
  font-size: 2em;
}

h2 {
  font-size: 1.333em;
  font-weight: 400;
}

a {
  color: steelblue;
  text-decoration: none;
}

a:focus,
a:hover {
  border-bottom: 1px solid steelblue;
  cursor: pointer;
}
```

Entry CSS using style.css

Finally we include our base and component styles, along with Normalize resets from the NPM module we installed earlier in this chapter.

```
@import 'normalize.css';
@import 'styles/base/elements';
@import 'styles/components/world';
```

The CSS entry file is stored at the `/app/style.css` path.

Run development server

This completes our first React app. Now run the app using the development server.

```
npm start
```

Once the app runs you should see following message from webpack-dev-server in your terminal window.

```
> react-speed-book@1.0.0 start ...
> NODE_ENV=development webpack-dev-server

http://localhost:8080/
webpack result is served from /
content is served from ...
404s will fallback to /index.html
Child html-webpack-plugin for "index.html":

webpack: bundle is now VALID.
```

Browse to your app on the url mentioned in webpack output. Now try changing some code like the style background or the Hello World message and hit save. Your browser should update the app without refreshing state.

When this hot loading update happens you will see following output in the browser console.

```
[HMR] App is up to date.
[React Transform HMR] Patching Hello
[HMR] Updated modules:
[HMR]  - 379
[HMR] App is up to date.
```

Now we build and serve our HelloWorld app.

```
npm run build
serve build
```

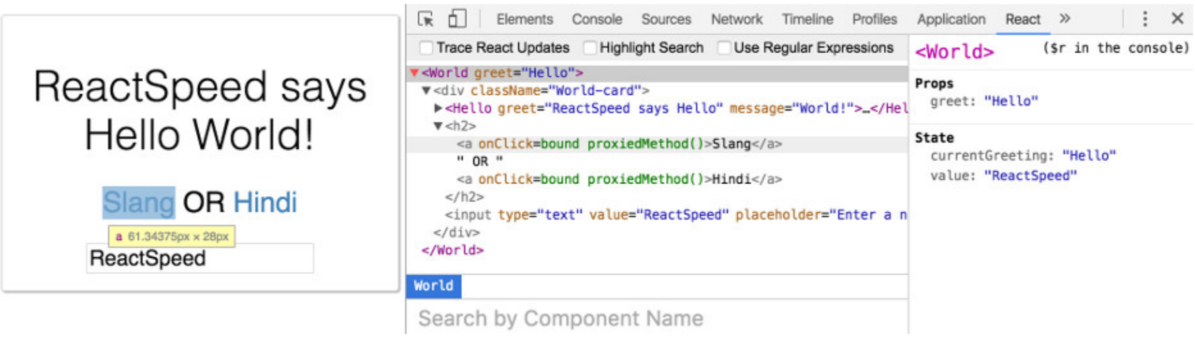
You will see a different webpack output on your terminal this time.

```
[ReactSpeed] react-speed-book: $ npm run build

> react-speed-book@1.0.0 build .../react-speed-book
> NODE_ENV=production webpack --config webpack.prod.config.js

Hash: 98bcd5d896e89e0c987a
Version: webpack 1.13.1
Time: 2534ms
      Asset     Size  Chunks             Chunk Names
     app.js  1.85 MB       0  [emitted]  app
   style.js  74.1 kB       1  [emitted]  style
favicon.ico  1.15 kB          [emitted]
 index.html  1.03 kB          [emitted]
    + 171 hidden modules
Child html-webpack-plugin for "index.html":
        + 3 hidden modules

[ReactSpeed] react-speed-book: $ serve build
serving .../react-speed-book/build on port 3000
GET / 200 8ms - 1.01kb
GET / 304 2ms
GET /style.js 200 4ms - 72.35kb
GET /app.js 200 69ms - 1.85mb
GET / 304 1ms
GET /style.js 304 0ms
GET /app.js 304 1ms
```

As you may have noticed the build is not highly optimized. The `app.js` file is a huge ∼2MB and css turned into JavaScript! In the chapter **Production Optimize Webpack** we will discuss various techniques to optimize for a production environment.

React Chrome Extension

In case you want to inspect how your components pass properties and how they are organized at runtime, you can install React Chrome Extension[31].



```
▼<World greet="Hello">
  ▼<div className="World-card">
    ▶<Hello greet="ReactSpeed says Hello" message="World!">…</Hel
    ▼<h2>
        <a onClick=bound proxiedMethod()>Slang</a>
        " OR "
        <a onClick=bound proxiedMethod()>Hindi</a>
      </h2>
      <input type="text" value="ReactSpeed" placeholder="Enter a n
    </div>
  </World>
```

```
<World>        ($r in the console)
Props
  greet: "Hello"
State
  currentGreeting: "Hello"
  value: "ReactSpeed"
```

React Chrome Extension in your browser

You can then select the code responsible for component UI and see the rendered UI highlighted. The extension will also update properties as they are passed along if you turn on the trace feature.

[31] https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en