# Pluralsight Course: Building a JavaScript Development Environment

Cory House (published Nov 2016)

Contents

[Pluralsight Course: Building a JavaScript Development Environment 1](#_Toc475901108)

[You Need a Starter Kit 4](#_Toc475901109)

[Starter Kit is an Automated Checklist 4](#_Toc475901110)

[What Belongs In a JS Starter Kit? 4](#_Toc475901111)

[Set Up Github 4](#_Toc475901112)

[Course Agenda 5](#_Toc475901113)

[Editors and Configuration 6](#_Toc475901114)

[Intro 6](#_Toc475901115)

[What to look for an JS editor 6](#_Toc475901116)

[JS Editor Recommendations 6](#_Toc475901117)

[Editorconfig 6](#_Toc475901118)

[Package Management 7](#_Toc475901119)

[Package Managers 7](#_Toc475901120)

[Install Node and NPM 7](#_Toc475901121)

[Package Security 7](#_Toc475901122)

[Demo: Node Security Platform 7](#_Toc475901123)

[Development Web Server 8](#_Toc475901124)

[Web Server Options 8](#_Toc475901125)

[Demo: Set Up Express Web Server 8](#_Toc475901126)

[Sharing Work in Progress 8](#_Toc475901127)

[Demo: Localtunnel 8](#_Toc475901128)

[Task Automation 10](#_Toc475901129)

[Automation Options 10](#_Toc475901130)

[Demo: NPM Start Script 10](#_Toc475901131)

[Demo:Pre/Post Hooks 10](#_Toc475901132)

[Demo: Create Security Check and Share Scripts 10](#_Toc475901133)

[Demo: Concurrent Tasks 10](#_Toc475901134)

[Transpiling 11](#_Toc475901135)

[Javascript Versions 11](#_Toc475901136)

[Transpilers 11](#_Toc475901137)

[Babel Configuration 11](#_Toc475901138)

[Build Script JS Style – ES5 or Transpiled? 11](#_Toc475901139)

[Demo: Set up Babel 12](#_Toc475901140)

[Bundling 13](#_Toc475901141)

[Module Formats 13](#_Toc475901142)

[Why ES6/ES2015 Modules? 13](#_Toc475901143)

[Choosing a Bundler 13](#_Toc475901144)

[Demo: Configuring Webpack 13](#_Toc475901145)

[Demo: Configure Webpack with Express 14](#_Toc475901146)

[Demo: Create App Entry Point 14](#_Toc475901147)

[Demo: Handling CSS with Webpack 14](#_Toc475901148)

[Source Maps 14](#_Toc475901149)

[Linting 15](#_Toc475901150)

[Why Lint? 15](#_Toc475901151)

[Linters 15](#_Toc475901152)

[Configuring ESLint 15](#_Toc475901153)

[Watching Files with ESLint 15](#_Toc475901154)

[Why Lint via an Automated Build? 15](#_Toc475901155)

[Demo: ESLint Set Up 16](#_Toc475901156)

[Demo: Watching Files 16](#_Toc475901157)

[Testing and Continuous Integration 17](#_Toc475901158)

[Test Decisions Overview 17](#_Toc475901159)

[Demo: Testing Setup 18](#_Toc475901160)

[Demo: DOM Testing 18](#_Toc475901161)

[Demo: Watching Tests 18](#_Toc475901162)

[Why Continuous Integration? 18](#_Toc475901163)

[What Does Continuous Integration Do? 18](#_Toc475901164)

[Choosing a CI Server 18](#_Toc475901165)

[Demo: Appveyor 19](#_Toc475901166)

[HTTP Calls 20](#_Toc475901167)

[HTTP Call Approaches 20](#_Toc475901168)

[Centralizing HTTP Requests 20](#_Toc475901169)

[Demo: Fetch 20](#_Toc475901170)

[Selective Polyfilling 21](#_Toc475901171)

[Why Mock HTTP? 21](#_Toc475901172)

[How to mock HTTP? 21](#_Toc475901173)

[Our Plan for Mocking 21](#_Toc475901174)

[Mocking Libraries 21](#_Toc475901175)

[Demo: Creating a Mock API Data Schema 21](#_Toc475901176)

[Demo: Creating Mock Data 21](#_Toc475901177)

[Demo: Serving Mock Data via JSON Server 22](#_Toc475901178)

[Demo: Manipulating Data via JSON Server 22](#_Toc475901179)

[Project Structure 23](#_Toc475901180)

## You Need a Starter Kit

You want good things like immediate error detection and clean processes.

Too many decisions to make about too many things... decision overload.

A starter kits acts like a set of best practices baked into a dev environment.

### Starter Kit is an Automated Checklist

…

### What Belongs In a JS Starter Kit?

* Package Management
* Bundling
* Minification
* Sourcemaps
* Transpiling
* Dynamic HTML generation
* Centralised HTTP
* Mock API framework
* Component libraries
* Dev web server
* Linting
* Automated testing
* Continuous integration
* Automated Build
* Automated deployment
* Working example app

I’d also like to add Selenium tests.

### Set Up Github

Using Git for source control. This supports CI and automated deployment.

* Install Git from git-scm.com (I installed v2.11, accepted all defaults)
* Sign for a Github account at Github.com. (I already have account MikeAtWest (GoWestBaby69) )
  + Create new repository:
    - Repo name: pluralsight-js-dev-env
    - Initialise this repo with e README: Tick.
    - Add .gitignore: Node
  + Click ‘Clone or Download’ button, copy the URL
    - <https://github.com/MikeAtWest/pluralsight-js-dev-env.git>
* Create directory for the project:
  + C:\Users\michael.mundy\Documents\VS Code\PluralsightJSDevEnv
  + Open the folder with VS Code
    - CTRL+` to open the terminal in VS Code
    - git clone <https://github.com/MikeAtWest/pluralsight-js-dev-env.git>
    - This will clone the repo into a directory with the same name as the repo.
    - Re-open VS Code with that folder
* To commit changes:
  + 1) To stage all the files you changed:  
    git add .
  + 2) Commit changes locally  
    git commit –m “added new feature”
  + 3) Push work up to GitHub  
    git push
* Should commit work after each module of the course.

Windows will store the logged in Git user. To delete this: You may have to check windows credential manager and delete the github entry under control panel > user accounts > credential manager > Windows credentials > Generic credentials

### Course Agenda

For each topic: List Options -> Makes Recommendation -> How to Implement Recommendation

## Editors and Configuration

### Intro

Choosing and Editor, and editor configuration.

### What to look for an JS editor

* Need strong ES2015+ support: Autocompletion; Parse ES6 imports; Report unused imports; Automated refactoring.
* Framework intelligence (for Node etc.)
* Built-In terminal.

### JS Editor Recommendations

Atom; WebStorm; Brackets; VS Code.

Used **VS Code** in this course.

You could use VS, but it’s not as good for JS. Use VS for the back end code.

### Editorconfig

Automate style consistency via settings in an .editorconfig file in root of project.

In VS Code:

* Go to [editorconfig.org](http://www.editorconfig.org) and download the plugin for VS Code, copy command, hit Window+P, run command and install.
* Add a ‘.editorconfig’ file and add settings. Restart VS Code.
* In any code file, hit SHIFT+ALT+F to apply formatting to code.

[Pushed to GitHub]

## Package Management

For a long time JS had no package management – now have many choices. We’ll review the choices; install NPM; and set up security scanning.

### Package Managers

Package managers let you share code. Bower used to be popular (requiring no build process), but now everyone needs a build process.

NPM is now clearly the standard, which is what we’ll use.

JSPM is also pretty cool, worth considering. But NPM plus a good bundler does the same job. There’s also Jam and Volo.

### Install Node and NPM

Go to nodejs.org and install latest version of node (I installed v7.6.0, accepted all defaults)

Node comes with NPM (Node Package Manager) built in.

(Can use nvm-windows to run multiple versions of Node.)

NPM uses a package.json file as its manifest (list of packages that the project is using).

In VS Code, add package.json and populate with code from course.

Then in terminal: npm install

### Package Security

NPM loads packages that might be published by anyone. Two approaches to checking for known bad packages are retire.js and the Node Security Platform. Recommends the later, which is already included as a dev dependency as ‘nsp’ in the package.json.

Probably best to run this check as part of the start script. (Slows build a little, but guarantees you’ll know about any vulnerabilities before you’ve used the bad package anywhere.)

### Demo: Node Security Platform

In VS Code terminal:

npm install –g nsp

nsp check

(Later we’ll call this from a NPM script when we run the start script.)

[Pushed to GitHub]

## Development Web Server

You’re going to want to run your code in a browser, which will require a web server. We’ll configure a dev web server so that it shows the latest results every time we hit Save. Also add services to share work.

### Web Server Options

* NPM package http-server:
  + Dead simple; http-server directory… done. No config.
  + Only serves static files.
* NPM package live-server:
  + Similar to http-server, but supports live-reloading the page when you save a file.
* Express:
  + More comprehensive and configurable. Can live non-static files, like APIs.
  + Can be used a production server.
* Koa and Hapi: Competitors to Express.
* Budo: Works great with Browserify, supports hot reloading.
* Webpack: has a built-in web server. Serves from memory. Supports hot reloading.
* BrowserSync: Server, including dedicated IP address on local network, and all interactions remain in synch on the IP… so you can ‘robot’ half a dozen browsers and devices at the same time. Very handy for testing across different screens and devices. Integrates with Webpack and Express. See github.com/BrowserSync/recipes for using BrowserSync with other servers.

### Demo: Set Up Express Web Server

We’re going to use Express.

In VS Code:

* We’ve already included express as a dependency in package.json.
* Add /buildScripts
* Add srcServer.js in new folder.
  + Add code from course.
* Add /src folder and put an index.html page in it.
* In terminal: node buildScripts/srcServer.js

Should have browser open and display the index HTML.

### Sharing Work in Progress

How can we quickly share work in progress with other on the public web?

* Localtunnel: Super-simple way to expose localhost to others
* Ngrok: Slightly more setup, but can be secured with password
* Now: Actually deploys files to web, supports Node.js projects
* Surge: Actually deploys files to web, only static

### Demo: Localtunnel

In VS Code terminal:

* npm install localtunnel –g
* node buildScripts/srcServer.js
  + Site opens in localhost
* Open a second terminal session using the + button at the right side of the terminal
  + lt –port 3000
    - This will give you a URL where you can see the localhost on the public web 0as long as the local server is running.)
  + Lt –port 3000 –subdomain mike
    - Will give url [mike@localtunnel.me](mailto:mike@localtunnel.me)

For bonus points, use BrowserSync as server, and use localtunnel to expose to others while you robot both sites.

Also: I should add a one-time script that runs all these ‘install localtunnel ‘ and similar NPM setup scripts.

[Pushed to GitHub]

## Task Automation

We want our tools and builds to be run automatically and consistently. Review options; set up NPM scripts.

### Automation Options

* Grunt: Oldest option.
* Gulp: More modern; runs in memory; code rather than convention.
* NPM script: Skip the middleman plugins (like Gulp) and just call NPM modules.

### Demo: NPM Start Script

In package.json, in ‘scripts’, add “start”: “node buildScripts/srcServer.js”

Then in terminal: npm start

By convention, that will call the “start” script, which starts the Express server and opens the borwser to our localhost.

### Demo:Pre/Post Hooks

Lets add a start message that will appear in the terminal when we start the server.

By convention, any script starting with ‘pre[x]’ will run before any script called [x].

In package.json, in scripts, add "prestart": "node buildScripts/startmessage.js",

Add startmessage.js file.

npm start

### Demo: Create Security Check and Share Scripts

In package.json, in scripts, add "security-check": "nsp check"

Note: This script does not need the have the ‘npm install –g nsp’ manually run first, because when you install a NPM package, it’s automatically added to the paths known by NPM (see the node\_modules/bin folder).

In package.json, in scripts, add "share": "lt –port 3000"

### Demo: Concurrent Tasks

You can use the npm-run-all –parallel command to run several tasks at once.

Can also use ‘npm start –s’ to reduce noise in console.

[Pushed to GitHub]

## Transpiling

ES6/2015 adds a lot of goodness to ES5 (which languished for a decade). But that’s made transpilers a big deal, so we can use ES6 today.

### Javascript Versions

* ES1: 1997
* ES2: 1998
* ES3: 1999
* ES4: Never happened
* ES5: 2009 – a decade gap
* ES6/ES2015: 2015 – tons of new features, classes, interfaces, arrow functions…
* ES7/ES2016: 2016 – a few features, but commitment to yearly releases
* ES7/ES2017: 2017 = Hopefully asyn, await, class props, object spread…
* . . .

### Transpilers

A JS transpiler converts a language to JS. There are over 100 languages that currently have JS transpilers.

But there’s only a few mature options: Babel, TypeScript (and maybe Elm).

* Babel: Transpiles the latest version of JS down to ES5, so it will run on browsers that don’t yet support ES6 or higher.
* TypeScript: Superset of ES6… ES6+, with type safety. Supports autocomplete and refactoring.

Some libraries can’t handle TS, because it’s not standard JS (early React and ESLint didn’t work with TS, but did with Babel). Babel also doesn’t need type definition files.

Cory argues Test, Lint, Babel, Great libs + IDE gives much the same benefits as TS.

Elm isn’t JS… it’s clean and immutable and friendly errors, no runtime errors. But you have to learn a new language in afunctional paradigm.

Cory opts for Babel. (I’ll replace this with TypeScript later, from <https://www.typescriptlang.org/docs/handbook/react-&-webpack.html> )

## Babel Configuration

You can use a separate .babelrc config file, or put it in package.json.

Babel can also transpile experimental JS features, which requires installing a plugin for the feature.

If you’re running an app on Node or Electron, Babel also has plugins to determine the support for various ES6 features, and then not transpile them.

### Build Script JS Style – ES5 or Transpiled?

* ES5: No transpiling required (so faster), no transpiler dependency.
* Transpiled: All the new features, same coding style and linting everywhere.

Cory will transpile build scripts as well.

### Demo: Set up Babel

In package.json, you’ll see several babel dev-dependencies; babel-cli, babel-core, babel-loader, babel-prepset-register, and babel-register.

Add a ‘.babelrc’ file to root, and set {“presets”: [‘latest”] }, which tells Babel to use all the latest standardised features (but not experimental features).

Existing code is in ES5, so let’s change some code to ES6 so we’ll have something to transpile. In /buildScripts/startMessage.js, change ‘let chalk = require('chalk');’ to ‘import chalk from ‘chalk’;’

If we then run npm start, it’ll fail with an error because Node doesn’t understand ‘import’.

So, in package.json, change "node buildScripts/startmessage.js" to "babel-node buildScripts/startmessage.js".

Now, babel-node transpiles our JS from ES6 to ES5 before passing it to Node.

Now make same changes to srcServer.js, and to the open-src script.

Now, in the next module we use babel for the app code as well as our start-up scripts.

[Pushed to GitHub]

## Bundling

Why bundle? NPM packages use the CommonJS module pattern – but browsers don’t understand this pattern, so you have to transpile them down to ES5 to use them.

You can also use bundling to combine the JS files needed for a certain page into one download – so the user has to download less to use the page.

### Module Formats

* Global variable (i.e. no modules)
* IIFE (Immediately Invoked Function Expressions)
* Asynchronous Module Definition (AMD): define
* CommonJS: required
* Universal Module Definition (UMD): Combo of AMD and CommonJs
* ES6/ES2015 Modules: import

ES6 is the standard; it’s what we should be using from now on.

### Why ES6/ES2015 Modules?

Standardized; eventually they won’t need tranpiling. Statically analyzable – so IDEs can provide autocompletion, refactoring, unused functions… fails fast. Tree shaking (in Webpack 2). Easier to read, plus named imports (bit of imported files) and default exports.

### Choosing a Bundler

Now that we have a module format, we need to pick a bundler to package our modules intelligently for the browser.

* RequireJs: Used the AMD module pattern.
* Browserify: First popular NPM package bundler. Large plugin system.
* Webpack: Bundles things other than just CSS – CSS, fonts, images, etc. Hot reloading webserver that serves from memory.
* Rollup: First bundler with tree shaking. Faster loading code than Webpack and Browserify. Quite new. No hotloading or code splitting.
* JSPM: Uses SystemJS behind the scenes to handle all sorts of module formats. Supports loading modules at runtime. Uses Rollup in its builder.

Cory recommends Webpack because it is comprehensive and mature. Intelligent multi-format bundling, hot reloading, intelligent module splitting are all good features. Webpack 2 will offer tree shaking. (Poll indicated 75% wanted Webpack.)

### Demo: Configuring Webpack

Add webpack.config.dev.js to root.

Because we’re already using Babel to compile our code, we can use ES6 code in this config.

Get code from bit.ly/2dSZwea

Note that in dev configuration Webpack will not actually create any files… it serves from memory.

See React with Webpack and Redux course for more detailed approach, or the Webpack Fundamentals course.

But we need to update our dev server to handle webpack’s bundle…

### Demo: Configure Webpack with Express

In srcServer.js, we import webpack and the webpack config, then we call webpack and pass the new object the compiler settings in the constructor. The we call ‘app.use’ to tell Express to use the Webpack as middleware.

But our apop doesn’t actually use any JS yet… so we need to add that.

### Demo: Create App Entry Point

When we set up Webpack, we set the entry point to the application as src/index (which refers to index.js)) So add index.js in /src

In index.js, we add an import of Numeral, which is a number formatting module, plus some ES6 code to log a string with a formatted number.

Then add a script tag pointing to bundle.js in index.html (In webpack.config, we set the output path and filename to src and bundle.js).

Npm start, and you see the message in the log, and if you check the ntwork tab, you see the bundle was received. The bundle contains our JS, transpiled down to ES5, and the bundle also contains the numeral module JS… all in a single file.

Next, lets bundle some CSS too.

### Demo: Handling CSS with Webpack

We already have a setup in our webpack.config that says how to load and bundle CSS.

Add a index.css file, and in index.js as import ‘./index.css’; (just like we were importing a .JS file).

How’d that work? Webpoack parsed the stylesheet, then used JS to inject the styles into the page. (In production, you probably just use a traditional style sheet link.)

How do you debug this stuff? It doesn’t look anything like our original code. The answer is source maps.

### Source Maps

How do I debug transpoiled and bundled code? Source maps map the code back to the original source, so in dev tools you can see original code. Source maps are only loaded if the dev tools are open, so they don’t bulk up the load (which would defeat the purpose of minification and bundling).

In webpack.config, we tell Webpack to geneate source maps via devtool: 'inline-source-map'.

There are a bunch of alternative source map generation options youi can use; the webpack docs for devtool.

Add ‘debugger’ to a line in code where you want the dev tolls debugger to pause. F12, reload, and you’ll see the code pause at debugger – and it’s the original code, not the bundled and transpiled code.

[Pushed to GitHub]

## Linting

### Why Lint?

Because you can catch errors at save time, instead of having to figure them out in production.

Enfoce consistency: Curly brace positions, disallow confirm/alert, trailing commas, globals, eval, etc.

Avoid mistakes: Extra parentheses, overwriting a function, making an assignment in a conditional, missing default case in switch, using debugger or console.log, etc.

### Linters

* JSLint: Original and opinionated.
* JSHint: Was popular.
* ESLint: Most popular. Really powerful and configurable. De facto standard.

ESLint does not work on TypoeScript. May need to use TSLint instead.

### Configuring ESLint

Lots of decisions to make: config format, whioch built-in rules, warnings or errors, plugins, use presets?

#### Decision 1: Configuration File Fomat

Can use a separate file, or add a ‘eslintConfig’ section to package.json.

#### Decision 2: Which Rules to Enable?

ESLint has a big set of default rules that you can use or not. Have a meeting and decide once and for all which to use.

#### Decision 3: Warning or Errors?

Which rules should just show a warning, and which should throw an error and break the build?

Warnings assist rapid work… errors stop bugs. Dev team should agree that commiting warnings is not acceptable.

#### Decision 4: Which plugins?

You can add plugins for libraries, like linting specific to React or Angular.

#### Decision 5: Use a preset?

Avoid the four decisions by using someone else’s set. ESLint comes with a bunch of recommended presets, which you can then tweak. Or use someone elses, like airbnbs, XO or StandardJS.

### Watching Files with ESLint

Issue: ESLint doesn’t have a watch command to make it run automatically. You can use eslint-wacther with Webpack to relint all files on save. Cory recommends eslint-watch (which does not require Webpack) to wrap ESLint – also adds better warnings, cleans messages, and can lint non-bundled files (like scripts).

Issue: Does not support many experimental JS features. If you need them, use babel-eslint.

### Why Lint via an Automated Build?

Many editors lint as you go, so why lint in the build? One place to check all issues; single config acrooss all editors; should be part of continuous integration.

### Demo: ESLint Set Up

Using recommended rules, with eslint-watch.

Add .eslintrc.json to root.

Grab rules from bit.ly/jsdeveslint

Notes:

Root: true lets ESLint know this is the one and only root lint rule file.

‘env’ lets lint know about environment, so it’ll expect and permit certain global variables (which are usually a no-no.)

And add a “lint” script to the package.json to call the ES Lint Watch (esw) package to watch our JS source code.

npm run lint

If you get any Errors, you’ll also get a scary looking build error after the results.

If you only have Warnings, no build error.

One warning is for a console.log in srcServer.js – but that’s ok because that’s a build script, not an app file.

Add /\* eslint-disable no-console \*/ to the file to disable that rule for that file.

Add //eslint-disable-line no-console to disable that rule just for the current line.

### Demo: Watching Files

Oddly, eslint-watch doesn’t watch files all by itself. To enable that:

Add a script: “lint:watch”: “npm run lint -- --watch’. That runs lint, and passes the watch flag to it.

Then add lint:watch to the default scripts in the start script.

Now, when you npm start, it’ll lint the code and tell you if it’s clean as it starts the site. It’ll also re-lint the code each time you change it while it’s still running.

[Pushed to GitHub]

## Testing and Continuous Integration

### Test Decisions Overview

Concentrates on **unit testing** – checking a function returns the expected results. (There’s also integration testing, and UI testing (see Selenium).)

Might be worth looking into options for those two types of testing.

#### Decision 1: Testing Framework

* Mocha: Very configurable.
* Jasmine: Popular.
* Tape: Lean and simple
* Quint: Oldest (was used for tetsing JQuery)
* Ava: Runs tests in parallel.
* Jest: From FB, used mostly by React devs, is a wrap over Jasmine.

Might be worth having another look at Jest.

Cory picks Mocha.

#### Decision 2: Assertion Libraries.

An assertion is a statement of what you expect to happen. An assertion library is a method to specify assertions. Popular libraries are Chai or Expect.

Mocha doesn’t come with a default assertion library – we’ll use Chai.

#### Decision 3: Helper Library

JSDOM: Provides a virtual DOM, so you can test withoiut actually opening a browser.

Cheerio: ‘JQuery for the server’. Lets you check the virtual DOM using JQuery selectors.

#### Decision 4: Where to Run the Tests

In the browser? Headless browser like Phantom JS (no visible interface)? In-memory DOM (Like JSDOM)?

We’ll use JSDOM.

#### Decision 5: Where do Test Files Belong?

Put them in a centralised folder, or alongside the source code?

Cory prefers having them front-and centre in a file alongside the source code, so he can see them when he sees the code. Also means import paths are simple. And it’s easy to get the tests for a file – it’s right next to it. And easy to move files around.

#### Decision 6: When Should Tests Run?

You shoud run them every time you hit Save. This increases visibility, and otherwise you’ll forget to bother to run them… then they are useless.

This is practical for unit tests, because they run really quick. (Maybe not practical for integration testing or UI testing.)

### Demo: Testing Setup

Mocha for testing framework; Chai for the assertion library; JSDOM as the helper library; we’ll run tests in Node with JSDOM; put test files alongside code files; and we’ll run tests every time we hit save.

In /buildScripts, add testSetup.js for testing config.

Add “test” script to run Mocha, use the ‘progress’ report style (which is compact, use the testSetup config file, and give the path to the test files (anything ending in test.js).

Npm test; it’ll complain there’s no tests yet.

Add /src/index.test.js as our test file for index.js. Add the example test, and try out with setting expected results to match or not.

So that tested a bit of code… next lets test something in the DOM.

### Demo: DOM Testing

Add imports for jsdom and fs (file system).

Now add test to check if HTML includes ‘Hello’. See code. This loads a copy of the index file, then pass the index to jdsom, with a callback that gets passed an err and a window. (JSDOM provides us with a virtual Window object.) Then we can just use JS to check the DOM in the window.

### Demo: Watching Tests

Add a test:watch watch script to run test with the watch parameter, then add it to the default start scripts.

### Why Continuous Integration?

When team commits code, it’s useful to ensure the code builds on another machine – that’s the continuous integration server.

Things that can break other people’s builds: Forgot to commit a new file. Forgot to update package.json. Commit doesn’t run cross-platform. Node version conflict. Bad merge. Didn’t run the tests.

### What Does Continuous Integration Do?

What’s a CI server do?

* Runs an automated build, and lets you know if anyone broke it.
* Runs your tests.
* Checks code coverage levels.
* Automate deployment (if checks are deployed).

### Choosing a CI Server

* Travis: Mac
* Appveyor: Windows
* Jenkins
* Plus others.

We’ll look at Appveyor (coz we only use windows)

### Demo: Appveyor

* Go to ci.appveyor.com and register/sign up using your GitHub account
* New Project; pick GitHub; lists repos; pick yours.
  + Settings: Use the defaults.
* Back to the code:
  + Add appveyor.yml to root
  + Paste in config from example; this wilkl install NPOM packages and run test
  + git add .
  + git commit -m “Add test scripts and continuous integration; Add appveyor.yml”
  + git push
* Back to Appveyor site, and click Latest Build.
  + Will install dependencies, run build, tests.

So every time anyone checks anything into the repo, this will verify that it works (and not just on their machine).

[Pushed to GitHub]

## HTTP Calls

Pretty much every JS app will need to make HTTP calls, so we need to look at the libraries used, and at mocking HTTP calls, as well as building a mock API.

### HTTP Call Approaches

Library options depend on where you ruin your app:

* Node only:
  + http
  + request
* Browser only:
  + XMLHTTPRequest; old and creaky and verbose
  + JQuery
  + Framework-based: Angular includes own HTTP service
  + Fetch: New standard from Web Hypertext App Tech working group (WhatWG), will need polyfills for regular or isomorphic/universal version, see github.com/github/fetch, returns promises
* Node or Browser:
  + Isomorphic-fetch
  + xhr (package on NPM), similar to request library
  + SuperAgent
  + Axios

Fetch is the way to go for the browser.

### Centralizing HTTP Requests

You should have one spot where you do all your API calls because:

* Configure all calls in one place, like URLs, credentials, etc.
* Handle preloader logic, like a spinner
* Handle errors in one spot in a standard way
* Single seam for mocking the API

### Demo: Fetch

Set up Fetch in centralised spot.

To start, add a single API call to the existing Express server. In buildScripts/srcServer.js, add app.get(/users’…). Just returns some hardcoded JSON that lists some user data. Npm start, and can see data at localhost:3000/users

Now update app to call our API. Add .src/api, and create userApi.js, paste in api module code. Includes the whatwg-fetch modue, which polyfills fetch. Note how success and error handling are abstracted away behind a very simple get() function. This is a bit like the Repository pattern, abstracting the API away from the app.

Go to index.html, add HTML for table headings. In index.,js, add getUsers() to call our API, then populate table of users.

### Selective Polyfilling

Some browsers (like Chrome) already support Fetch, so why send our polyfill to everyone?

You an use scripts with a src pointing to <http://cdn.ployfill.io> to sniff whether the browser can handle a certain feature, then provide a poly fill if necessary.

### Why Mock HTTP?

It’s useful to mock HTTP requestes because:

* Unit testing
* Instant response
* Keep working if services down
* Avoid inter-team bottlenecks
* Work offline

### How to mock HTTP?

* You can use the Nock site to return a specific response for a call.
* You can point to a static JSON file.
* You can create a dev webserver with JSON server, which returns editable data from JSON files. You can use JSON Schema Faker to generate random JSON data.
* You can user Express or BrowserSynch as a real server with a real API.

### Our Plan for Mocking

Declare schema using JSON Schema Faker; Use faker.js to geneate random data; use JSON Server to generate a mock API based on the schema, and supports adds/updates/deletes on the JSON files.

### Mocking Libraries

JSON-schema.org

Github.com/json-schema-faker/json-schema-faker; includes faker.js and chance.js

Faker.js: Check out the interactive fake data front end, and docs for faking values via a schema definition

Github typicode/json-server: The JSON Server repo

### Demo: Creating a Mock API Data Schema

Add /buildScripts/mockDataSchema.js, paste in the schema. (Note if you leave off the ‘required’ definition, the API will occasionally leave out unrequired properties.)

### Demo: Creating Mock Data

We have a schema, now use JSON Schama Faker to generate random data and pop it in a file.

Add buildScripts/generateMockData.js, paste in code. This passes our schema to the JSON Schema Faker, then drops the result into a data file.

Then add script in package.json called “generate-mock-data” to call that. Npm run generate-mock-data to test.

### Demo: Serving Mock Data via JSON Server

JSON Server will look at our JSON data, then create a fake API that exposes the data.

Add “start-mockapi” script, which calls json-server and tells it to watch the db.json file that we just created and filled with random user data, then create a API at port 3001. Npm run start-mockapi, tehn go to the URL that results, <http://localhost:3001/users>. It’ll return our user data, served up via our mock API via HTTP.

Generating new mock data each time we start the app is good, because it’ll expose errors that occur in corner cases. Add “prestart-mockapi” script that runs our generate mock data script before we start the api, and add start-mock-api to our start script.

Next to need to update app to hit the :3001 API endpoint instead of our :3000 hardcoded API. Add /src/api/buildURL.js, and paste in code. This checks if we’re at localhost, and if so swaps to our mock api endpoint at 3001. Then import and use baseURL in userApi.js to alter our API url. Npm start (and probably refresh) and see our new random data in the app.

But the Delete function doesn’t work yet.

### Demo: Manipulating Data via JSON Server

In userApi.js, ad the deleteUser() function. Note out private del() function centralises anything to do with deletions, while keeping deleteUser() clean. Also add code to index.js to call the deleteUser() function when clicking Delete links.

When you click Delete, the record is actually deleted from the JSON file.

[Pushed to GitHub]

## Project Structure

### Why a Demo App?

* Shows clear expectations of directory structure and file names
* Framework usage
* Testing
* Mock API in use
* Demos working automated deployment
* Codifies decisions
* Interactive example of working in the stack

### Tip 1: JS belongs in a .js File

Don’t slap JS code in a script tag. No testing… no linting… no reuse… no transpilation… no imports.

Don’t generate js on the server-side… same sort of issues. Instead inject JSON from the server describing the situation, and let the JS handle behaving differently based on that. Generate data, not code.

### Tip 2: Consider Organising by Feature (instead of by File Type)

Results in a lot of bouncing around the folder structure looking for files.

### Tip 3: Extract Logic into Plain Old JavaScript Objects (POJOs)

A POJO contains logic in plain JS, without any framework code.

Makes it easier to test code, and easy to change the framework.

**CHECK react-slingshot as example framework.**

[Pushed to GitHub]

## Production Build

Creating an automated production build.

### Minification and Source Maps

A JS minifier makes JS files smaller. It shortens variable and function names, removes comments and whitespace and new lines. Newer versions will also do tree shaking to eliminate unused code. Basically, a minifier gets rid of code that’s only needed by humans so it can make the file smaller.

### Demo: Production with Webpack Configuration with Minification

Copy webpack.config.dev as webpack.config.prod

In prod config:

* Change devtool from ‘inline-source-amp’ to ‘source-map’. This runs a bit slower, but it will create source maps that deal with minification, transpilation and bundling.
* Change output path to ‘dist’; we’ll create physical files in the ‘dist’ directory (unlike the in-memory only files we serve in dev).
* Add UglifyPlugin to plugins section, and import webpack (as we’re now using a webpack plugin)
* Add Dedupe plugin (to remove any duplicate packages from our minimised code)

Next need a script to run the prod build script. Add buildScripts/build.js, and paste in code.

### Demo: Configure Local/dist Server

It’s handy to be able to run the production code locally to see how it works.

* Copy srcServer.js as distServer.js.
* Remove all the webpack stuff – we’ve already used webpack to create the physical files in the /dist folder, so we don’t need it any more.
* Add app.use(express.static('dist')); to tell Express to serve up static files.
* In app.get, change to serving files from the /dist folder.
* Import compression, and add app.use(compression()); so that files will be served using GZip compression.

*Note: Need to move app.use(compression()) above app.use(express.static('dist')); for compression to work.*

So that’s our local dist server set up, but we’ll still need a mock API for local dist as well.

### Demo: Toggle Mock API Between Src and Dist

In baseUrl.js update code to instead check the querystring for ‘useMockApi’ to flip between real API (in srcServer.js) and mock API (which is the JSON Server stuff).

So now <http://localhost:3000> will hit the real API (which will return our hardcoded data), and <http://localhost:3000/?useMockApi=true> will hit the mock API (which will return our random data from JSON server).

### Demo: Production Build NPM Scripts

Add four scripts to do the production build:

* clean-dist: This deletes and recreates our /dist folder, to guarantee that it’s clean.
* prebuild: This runs clean-dist, followed by the test script and the lint script, so we can be confident that our code is valid before we build it.
* build: This uses babel-node to transpile our code.
* postbuild: This starts the dist server.

npm run build -s

That will build successfully, putting the files into /dist and opening the site… but it’ll tehn error out because we have not put copies of our static files (index.html) into the /dist folder. So…

### Dynamic HTML Generation

You may want to generate HTML dynamically on Production. Why?

* If your system is dynamically renaming HTML files, it’s good to automatically reference those files in your bundle.
* Handle dynamic bundle names, and set cache-expires dates.
* Inject prod-only resources (like error loading, see below).
* Minify HTML.

How to include the reference to the bundle in the index.html page?

* Hardcode it (like we’ve done so far)
* Manipulate via Node (use Regexp to update the files)
* Use html-webpack-plugin to use Webpack to manipulate files.

### Demo: Dynamic HTML Generation

Use html-wepack-plugin to handle our index file.

We could do this crudely by just copying the index.html into the /dist, but we can do some other tricks if we use Webpack to handle our HTML.

In both .config files, import HtmlWebpackPlugin and add it to the plugins. This will tell Webpack to copy the index.html file into /dist, as well as to inject the bundle file into it.

We can do even better, by minifying the HTML as well, by adding minification settings into the HtmlWebpackPlugin.

### Bundle Splitting

In a larger app, you’ll want to have multiple bundles, preferably one per page (or feature), so user only has to load the bundle for thing they just asked for. Also it’d be nice to have our vendor JS in a separate bundle, so they don’t have to reload that next time we update a small portion of our app code.

### Demo: Bundle Splitting

We’ll set up a separate bundle for our vendor code.

A few steps:

* We currently have only one entry point; we’ll change that from an array of entry points to an object with two properties: main (or app code from /src/index) and vendor (from src/vendor).
* Vendor doesn’t exist yet, so we add src/vendor.js. It has nothing in except an import of fetch (the only library that we’re using.) Webpack will use this to start tracking the vendor libraries that we want in a separate package. In a real app, you’d import all the libraries in the app; JQuery, Bootstrap, React, etc.
* We’ll need to use the CommonsChunkPlugin in our webpack.config.prod to do our bundling (Webpack calls bundles chunks), adding the name ‘vendor’, which matches our vendor entry point. The plugin will keep track of the packages that are imported in the ‘vendor’ bundle, and will exclude them from any other bundles (that is, our ‘main’ bundle). If you left this plugin out, it’d create two bunldes… but fetch would be included in both bundles.
* Finally, we need to change the output filename from ‘index.js’ to ‘[name].js’. HtmlWebpackPlugin will take care of changing the [name] to a script reference for our ‘vendor’ and ‘main’ script bundles.

Npm run build –s, and you’ll see two bundles – ‘vendor’ and ‘main’.

### Cache Busting

If we set a cache-expires header on a bundle, we can tell the browser nit to reload that file until the expiry date is reached (which might be up to a year in the future). This is great because it saves reloads. But then you need a way to ‘bust’ the cache, so the user can reload a new copy of the bundle if it’s updated.

Plan for cache busting:

* Hash bundle file name (so if bundle does not change, the file name stays the same, and will remain cached).
* Now that the bundle name is a hash, which will vary, we’ll need to generate the HTML dynamically so that we can inject a the bundle name.
* We’ll do all that using the with the HtmlWebpackPlugin.

### Demo: Cache Busting

Need to set production server to deliver .js files with a far-future cache expiry date.

In webpack.config.prod:

* import WebpackMd5Hash
* Add it as a plugin
* In the output filename, set to [name].[chunkhash].js. That will put the hash of the file contents into the filename.

npm run build –s

Check /dist, and see that filenames have hashes in the middle. And index.html has script references to the bundle files including the hashes.

### Demo: Extract and Minify CSS

Currently Webpack is dynamically generating our CSS and imbedding it in the bundle. You probably want to serve your CSS from a separate file (as well as support cache busting for that file).

To do this:

* Import ExtractTextPlugin
* Add the plugin to the list of plugins.
* Replace the current module loader for CSS with the ExtractTextPlugin

The css?sourcemap in the loader tells Webpack to generate CSS sourcemaps as well.

npm run build -s

You’ll now see a .css file and a .css.map file, which have hashes in the name. the index.html file now also has a link=’stylesheet’ in it.

### Production Error Logging

There are various services that support logging JS errors in production: TrackJS, Sentry, New Relic and Raygun.

When evaluating an error tracking service:

* Error metadata:
  + Does it give browser details?
  + Capture stack trace>?
  + Capture user’s previous actions?
  + Custom API for logging own data?
* Notifications and integrations
  + Send emails when errors occur, or intragrate with Slack?
* Analytics and filtering
* Cost?

This is not an easy task, it’s worth paying for the service.

### Demo: Error Logging using TrackJS

Sign up for a TrackJS test account. ([MMundy@west.com](mailto:MMundy@west.com), School69)

Add the script tags in the head at the top of the index.html page.

Npm run build –s

In browser, in console call trackJs.track('ahoy trackjs!');

Go back to TrackJS site, and the error will be logged under Your Data

But, with this setup, TrackJS will also be running in dev, and logging bugs from there, which will add noise to our error log. Next clip shows how to dynamically inject portions of HTML (like our TrackJS scripts) in different environments.

### Demo: HTML Templates via EmbeddedJS

We’ll use the templating engine support that’s built into HtmlWebpackPlugin.

HtmlWebpackPlugin supports a number of templating languages, including Jade, EJS, Underscore, Handlebars and HTMLLoader. It defaults to EJS 9EmbeddedJS), so we’ll use that. See EmbeddedJS.com.

We want to only inject the TrackJS script code into our production code. To do so:

* In the prod config, add the token from TrackJS (and that’s refered to in our script) as a property on the config object for HtmlWebpackPlugin called trackJSToken.
* In index.html surround TrackJS with EJS conditional that checks if trackJSToken exists as a property on the HtmlWebpack configuration.
  + This looks suspiciously like inline ASP, but it’s not. It’s EJS.
  + Technically, our .html file isn’t an HTML file any more, it’s an .ejs file. But it’s ok to leave it as .html so editors will color the html properly.

So pretty much done. Time to talk about shipping to production…

[Pushed to GitHub]