Course Overview

Course Overview

Ah, JavaScript. It's so easy to get started. Wait, no it's not, it's overwhelming. Now sure, today, more of us are writing JavaScript than ever before, and if you've been coding for long, you may recognize the value of testing, linting, bundling, transpiling, automated builds, and more, but all too often, we ignore many of these items. Why? Because setting up a solid JavaScript development environment is so much work. The list of choices is so long and complex that it's hard to know even where to get started. Hi, I'm Cory House, and I'm here to help. I've been building rich web applications in JavaScript for over 15 years now, and frankly I'm tired of starting from scratch. So in this course, I propose a solution. What if your team had a powerful, rapid feedback development environment? What if every time you started a new project, you had a solid foundation that supported automated testing, bundling, linting, transpiling, error logging, and so much more out of the box? That's the goal of this course. Let's stop starting from scratch. Let's build a JavaScript starter kit so your team can rapidly get started and enjoy all the benefits of a modern, powerful, and automated development experience. A great JavaScript development environment makes doing the right thing the easy thing. In this course, we'll make it happen.

You Need a Starter Kit

You Need a Starter Kit

Hi, I'm Cory House, and welcome to Building a JavaScript Development Environment. I want to begin by selling you on the premise of this course. I believe your team needs a starter kit. No matter where you're running JavaScript, you need the same basic things. Do you like to fail fast? When you make a typo in your JS, do you want to know as soon as possible? When a test fails, would you like to know immediately? And are you overwhelmed by the number of options that are available in the JavaScript space today? Those are just a few of the reasons that you need a starter kit. Here are over 40 decisions that you might consider when creating a robust JavaScript development environment. And the wild thing is, this list isn't even comprehensive. Decision overload shouldn't affect your entire team on every new project. If you can make all of these decisions just once, then every future project can benefit from a solid foundation. And by the way, these are just some of the decisions that we're going to consider throughout the course. I couldn't even fit them all on one slide! So, are you feeling overwhelmed yet? Wouldn't it be nice if all these best practices were baked in your team's starter kit for future projects? The fact is, today your team needs its own development environment, or what I like to call a starter kit. Starting from scratch really isn't practical anymore. If your team is starting from scratch on each new project, they're going to spend weeks tripping over all of these decisions and boilerplate to get to the point that they have a solid development experience. So our goal in this course is to speed development and increase quality by reducing the list of things that you need to set up, do manually, or try to remember on each new project. So what's the big benefit of gathering your team together, and either selecting a starter kit or building your own? Well first, it codifies a long list of decisions, best practices, and lessons learned. It also encourages consistency. It protects us from forgetting important details. It

increases quality because doing the right thing becomes automatic. If you want your team to consistently do the right things, it's really valuable to make sure that doing the right thing is the easy thing. And finally, it avoids repeating work. You shouldn't have to think about how to handle minification, automation, transpiling, bundling, cache-busting and more every time you start a new project. A starter kit encapsulates all these decisions for you. See, today's list of best practices is just too long for any individual to track. So in this course, let's create a starter kit that does it for you.

## A Starter Kit Is an Automated Checklist

I like to think of a starter kit as an automated checklist. To help drive home the importance of checklists, let's consider an example from the medical field. Imagine that you have a sick patient and a doctor needs to run a line into their body. When a doctor puts a line into a patient, they have a specific set of steps that they have to perform. They have to wash their hands with soap, clean the patient's skin, put drapes over the entire patient, wear sterile equipment, and put a sterile dressing over the catheter site once the line is in. But here's what's interesting, although doctors have memorized this simple list, at one ICU doctors who are observed for a month accidentally skipped at least one of these steps one third of the time. That's a big deal, because skipped steps lead to infections which can lead to further sickness or even death. I realized the power of starter kits while reading "The Checklist Manifesto". The big idea is simple: as professionals, we think we can remember all the steps involved, but we can't. As we just saw, checklists aren't just for amateurs, they're for professionals too. Doctors placing lines in patients must follow a short list of tasks to help avoid infections, yet each year, thousands die from line-infections because of skipped steps. So one hospital instituted mandatory checklists with these amazing results: the 10 day line-infection rate fell from 11% to zero. So they followed patients for 15 more months, and only two line-infections occurred during that entire period. So they calculated that in this one hospital, the checklists have prevented 43 infections, eight deaths, and saved over two million dollars in costs. See, doctors know what to do, but there are so many steps involved that it's easy in a high-pressure situation to overlook an important step in their job. And it's the same story today for JavaScript developers, we know what to do, but there are so many steps involved in doing things right, that it's easy to overlook a step. A JavaScript starter kit is a living, automated, and interactive checklist. It removes the mental burden of trying to remember all the best practices, because it makes them automatic. It makes sure the right things get done even when you're in the heat of the moment. Thus, a well-designed starter kit becomes a pit of success. Doing it right becomes easier than doing it wrong, that's a big win. Now hopefully I've sold you on why a JavaScript starter kit is so critical for your team. So now let's discuss who this course is for.

## Who Is This Course For?

These days, it seems JavaScript is everywhere. So you're likely wondering, Which JavaScript developers is this course for? Around a decade ago, Jeff Atwood, blogger on codinghorrror.com and co-founder of stack overflow, said something truly prothetic in a blog post called "The Principle of Least Power". Any

application that can be written in JavaScript will eventually be written in JavaScript. At the time, this statement sounded pretty outlandish, yet here we are a decade later, living in a world where npm, effectively JavaScript's package manager, is easily the largest package manager in the world with over 300 thousand npm packages, and it's growing fast. JavaScript is swallowing the world, today, you can use JavaScript to deploy an app nearly anywhere. Now, of course, JavaScript got its start on the web, so if you're building web apps this course is for you. But today, you can also use your JavaScript skills in all sorts of interesting areas. You can run your JavaScript on the server using node JS. You can use JavaScript to build mobile apps using NativeScript, PhoneGap, or React Native. And with great tools like NWJS and most recently, Electron, you can even create installable desktop apps that run on Mac and Windows. Well, in short if you're working in JavaScript, then this course is for you, whether on the Web, the server, mobile, or desktop. Many of the patterns, practices, and libraries that we'll explore are useful in all of these contexts. That said, the demo we're going to build is a traditional web-oriented development environment. And we'll use node for our build tooling and API, along with GitHub for source control. But I'm focusing on primarily universal principles, so most of what we cover will apply to all of these JavaScript based technologies. So the question is who isn't this course for? Well maybe you're already working in a highly opinionated and comprehensive framework that provides a rich development environment baked-in. Ember is an example of this. When you choose Ember, you're expected to use the Ember CLI, Ember's built-in tooling takes care of most of what I discuss in this course for you. See, Ember is comprehensive and notoriously opinionated. It already has opinions on file structure, testing, CLI, and much more. Ember's like a religion, you accept it all or you won't fit in. Ember is effectively a comprehensive starter kit because the full development environment comes baked-in. In comparison, Angular, React, node JS, Electron, Backbone, Vue, and many other libraries and frameworks are less opinionated. They solve specific problems so you can pose them with other libraries and patterns to build a real app. Many JavaScript technologies leave testing, builds, file-structures, deployments, validation, HTDP calls, and more up to you. This can be intimidating at first, but there's an obvious advantage: you're free to select the best of read technologies and patterns that suit you best. And your final solution isn't weighed down by things that you don't need. Of course, there's no right answer on this continuum. But the less opinions that your current JavaScript framework or library offers, the more that this course will be useful for you. Of course, depending on the library or framework you're working with, there may already be a variety of starter kits or boilerplates available for you to use. So you might be asking, What are the advantages of selecting an existing boilerplate project instead of building your own? Well the benefits of using someone else's project are quite obvious. You can, of course, save time, at least initially. Of course, the more that you have to spend time learning how it works so that you can tweak it to your needs, the less benefit that this applies. And if many people are already using the boilerplate, you can also trust that it's a proven setup that's likely to be reliable. This doesn't mean that it does exactly what you want, but at least it means that you can likely trust what it claims to do. And finally, many boilerplates are full-featured, they offer a wide variety of potential use cases that would be very time-consuming to handle when starting from scratch. These are all compelling reasons to consider what's already out there. However this course exists because there are many benefits to building your own development environment. First, freedom. You're free to select the best and most relevant technologies for your needs. You can build a development that's perfect for your team and technology stack. This helps avoid unnecessary complexity. Any existing boilerplate projects

are likely to contain technologies that you don't want or need, or it can be time-consuming and error-prone removing features or tweaking how they work when it's a project that you're not familiar with. When you build your own environment, you understand it in detail. This makes it easy to fix bugs and tweak the setup as time passes.

## What Belongs in Your Starter Kit?

You might be wondering what belongs in your team's JavaScript starter kit. Here's just a few items I believe belong in your starter kit: package management, bundling, minification, sourcemaps, transpiling, dynamic HTML generation, centralized HTTP requests, a mock API framework, your component libraries, of course, a development webserver, linting, automated testing, continuous integration, an automated build and automated deployment. And finally it should include a working example app to help convey how all these parts come together, including directory structure and file-naming practices. These are just a few of the concerns that we'll cover in this course as we explore the long list of options in each area, and build a robust JavaScript development environment from the ground up. You can think of this course like a playbook of key decisions that you need to make when setting up a development environment. Since the list of decisions and options is so long, without a structured playbook like this, it's easy to overlook important considerations. So in this course, I'll walk you through the long list of decisions to consider, so you can create a development environment that suits your needs best.

## Set Up Github

Assuming I've sold you on why building your own JavaScript development environment is so important, let's talk about the plan for the rest of the course. In this course, I'm going to use git for source control. If you're already familiar with git and GitHub, I suggest creating a repository for this course since it will come in handy later. So if you're already familiar with git and GitHub, go create a repository and you can jump to the next clip. Now, don't worry if you're not familiar with git or GitHub. It's certainly not required to understand the course, however I'll use git and GitHub in two specific parts of the course. In module nine, we'll configure continuous integration. I'll show how to integrate continuous integration servers with a GitHub repository. And in module 13, we'll create an automated deployment to Heroku that integrates with GitHub. So if you want to follow along with the exercises in these modules, follow these next few steps to get set up with git and GitHub. First, install git from "git-scm.com". Just accept the defaults during installation, they'll be just fine for our purposes. Next, go to "github.com" and sign up for an account. Don't worry, it's free. Once you've signed up and logged in, you can click the plus sign in the top right-hand part of the screen to create your first repository. I recommend using the settings that I've shown here, the most important point is to be sure to initialize the repository with a gitignore for node. This will assure that git ignores the node module's folder. It's huge, so you don't want to check it in. And the fourth and final step is to clone the repository to your machine. To do that, click the green "Clone or Download" button and then copy the URL that it provides. Then, open up your command line and run the git clone command in the directory where you'd like to place this project. Of course, be sure

that you use your URL that you just copied instead of the one you see here. This command will clone the repository into a directory with the same name as your repository. And that's it. If you'd like to commit your work to git as you follow along with the course, there are three commands that you need to know. First, you can stage your work by saying "git add dot". The dot says, stage everything. Then you can commit your stages locally and provide a message for the commit. Finally, you can push your work up to GitHub by saying, git push" I won't mention git again until much later in the course, but I suggest committing your work after each module, so that you have a handy working checkpoint that you can go back to if necessary. And with this out of the way, now we can talk about the course agenda.

## Agenda

Let's walk through the plan for the rest of the course. Before we move on, I want to clarify: this course is about building a JavaScript development environment from the ground up. But you'll commonly hear people use the term Boilerplate, Starter Kit, Seed, or Starter Project as well. Throughout this course, I'll use all these terms, they all mean the same thing. Here's the rhythm that we're going to follow throughout the course. We'll walk through a three-step process for each topic. I'll begin by discussing the list of options for solving a given problem like linting, testing, bundling, and so on. After reviewing the options, I'll make my recommendation and justify my suggestion. Then, I'll show how to implement my recommendation. By following this rhythm one topic at a time, we'll create a robust and scalable JavaScript development environment from the ground up. And you'll have a good understanding of the many potential ways to build your own development environment. So now that you understand the general rhythm of the course, let's discuss the high-level course structure. We just got the intro out of the way here in module one. In module two, we'll set the foundation for a development environment by considering how to select a JavaScript editor. And we'll look at enforcing key configuration via editorconfig. In module three, we'll make an important, but ultimately easy decision, we'll pick a package manager and set it up. And then, in module four we'll focus on development of web servers, so you can automatically run your app without manually configuring web server on your machine. And we'll investigate interesting ways to rapidly share your work-in-progress using powerful, free services. In module five, we'll explore options for automation so we can script our entire development and build processes to happen with a single command. And these days you'll likely want to transpile your code so you can enjoy the latest features of JavaScript, or alternatives that compile down to JavaScript. So we'll explore this idea in module six. In module seven, we'll move on to bundling. You need to bundle your code in a way that's suitable to your environment. So we'll talk about our options there and see specifically how to configure my recommendation. In module eight, we'll consider options for linting our code to help assure consistency and help catch issues the moment that we hit save. Of course, you'll also want to test your code so you can confirm that it works as expected in an automated fashion. You'll see a long list of decisions that you need to make to set up a productive workflow for doing test-driven development in JavaScript. We'll also integrate with free continuous integration servers on both Linux and Windows so that we're notified immediately if someone has broken the build. And these days, nearly every JavaScript application needs to make HTTP requests. We'll look at various ways to look at HTTP requests, including popular libraries, and we'll see how to create mock APIs so that you can do

rapid and realistic development without an actual API present. In module 11, I'll explain why including a demo app in your starter kit is so important, and we'll explore keys for structuring scalable JavaScript projects. In module 12, we'll address a key topic: production builds. We shouldn't rely on humans to ensure we get this right, so in this module we'll create an automated build that prepares the application for production. With a single command, we'll minify the code, split our code strategically into multiple bundles, bust cache, set up error logging and more. And in the final capstone module, we'll create a deployment process that automatically deploys our API and user interface to separate cloud hosts. And I'll close out the course by reviewing strategies for keeping your production applications updated with the latest versions of your starter kit. We have a lot of ground to cover. Let's wrap up this intro with a quick summary.

### Summary

In this module we explored why starter kits are so useful in this fast-paced and complicated era. We saw that starter kits reduce decision fatigue by standardizing a long list of agreed tools, patterns, and practices. They increase quality because doing the right thing becomes the easy thing, it's automatic. The list of best practices in JavaScript has become too long for most people to hold in their head. A good JavaScript development environment creates a pit of success by codifying a long list of lessons learned. When you find something new that everyone should start doing you now have a place that you can codify the practice so that it happens automatically from now on. A well-structured development environment provides rapid feedback on your work, quickly reporting on any linting, testing, or compilation errors the moment that you hit save. And we saw how checklists are critical in other fields like health care, to ensure that patients get the best possible care. Even though doctors know what to do, it's easy to overlook a step or two in the heat of the moment. A well-designed JavaScript development environment operates like an automated checklist. It assures the developer doesn't overlook important concerns or make common mistakes along the way. And I wrapped up by quickly discussing who will benefit most from this course. In short, if your team is writing JavaScript without a starter kit, this course is for you. Okay, with this intro out of the way, let's get rolling. In the next short module, let's choose an editor and use editorconfig to enforce consistency.

## Editors and Configuration

### Intro

To get things started, we need to make a few key baseline decisions about our tooling. So in this short module, let's begin with an obvious point: we need a JavaScript editor. So let's quickly look at some options, and I'll help outline what you need to be looking for in a solid JavaScript editor today. Then we'll wrap up this short module by assuring everyone on our team uses the same critical editor settings, like spaces versus tabs, line endings, and more, by using a standard called editorconfig. Alright, let's get started.

## What To Look For in a JavaScript Editor

Let's begin with the difficult job of selecting a JavaScript editor. So what should you look for? Well, first it should offer strong support for the latest JavaScript features. Most editors do so today, but keep in mind that the larger your editor is, the slower it will typically be to add support for new JavaScript features, and it's also likely to be slower to add support for alternative languages that transpile to JavaScript in the future. Now when we talk about strong ES2015+ support, you should look for things like this: Does it index your JavaScript file so that it can intelligently provide autocompletion? Does it parse ES6 imports? Since ES6 imports are statically analyzable, strong JS editors can report unused imports as well as typos and import statements. Does it offer automated refactoring tools like rename and extract function? Again, since ES6 imports are now statically analyzable, such refactoring can finally be handled in a safe and automated way. This is a great win that you get for free if you use a modern JS oriented editor. And many editors now offer built-in support for popular frameworks and libraries like Node, React, and Angular. For instance, WebStorm and VSCode both offer built-in node debugging, and WebStorm even recently added support for React proptype autocompletion. Many more examples like this exist. Finally, this is arguably the most important to me: a built-in terminal. All the tools we're going to use throughout this course are node based, and they rely on the command line. Having a terminal built-in gives you a single window with everything you need to know about the current status of your application. It's handy having one spot to check for any linting errors, broken tests, build status, code coverage, and more.

## JavaScript Editor Recommendations

There's a wonderful selection of powerful editors these days, but these are the four that I recommend considering. These editors offer many, if not all the benefits that I listed on the previous slide. WebStorm is currently my favorite editor, however it's also the only option on this list that isn't free, but it does offer a free 30-day trial. Now, in this course I'm going to use VSCode because it's free, it's fast, it offers a built-in terminal, it has excellent Git integration and node debugging, and it boasts a rich plug-in ecosystem. Now, when writing JavaScript it's easy to default to the IDE that you already know. If you're a .Net developer, Visual Studio will feel like a natural choice. And if you're a Java developer, perhaps continuing to do all your work in Eclipse or Netbeans seems the most logical. Of course, these full-blown IDEs can certainly handle JavaScript and much more. These are an okay option too, but it's important to recognize that these IDEs aren't as focused on supporting the latest features in JavaScript. So don't feel obligated to use a single editor for everything. If you're currently writing code in a large IDE like one of these, you owe it to yourself to check out the lighter weight JavaScript editors that I showed on the previous slide. See, I like to think about it like this: if you're writing both front and back-end code, there's nothing wrong with using a different editor on each side. This is becoming increasingly common in a service-oriented world, where the technologies that we use on the back-end for services are

completely different than those that we use on the front-end. Using different editors for front and back-end development means that you're choosing the best tool for the job for each scenario.

## Editorconfig

Another common war on any dev-team is editor configurations and stylistic choices like tabs versus spaces. I have great news, you don't need to fight this war any longer. There's now a handy way to maintain consistency. If you create a file called ".editorconfig" and place it in the root of your project, you can specify how your editor should handle common settings like tabs versus spaces, indent size, line feeds, char sets, and trailing white space. And your team doesn't even have to use the same editors to enjoy all of this consistency. The editors at the top have support built-in, and those listed here at the bottom require a plug-in. So here's how this works: you create a file called ".editorconfig" and place it in the root of your project. Here's the editor config that I typically use. Yes, I'm choosing spaces, but hey, if you prefer tabs, we can still be friends, I guess. Unless you start preaching to me. Anyway, that's the beauty of this file, although I standardize my code to use spaces via editorconfig, I honestly always hit Tab to indent my code, but the nice thing is with editorconfig in place, any tab that I hit is automatically converted into two spaces based on the indent style and indent size settings that you see here. How cool is that? This way, those that prefer hitting tab can continue to do so, and the editor will quietly do the right thing, so handy.

## Demo: Editorconfig

Alright, enough talk, let's now set up an editorconfig for our project. Alright, here I am in VSCode, the editor that I'll be using throughout the course. To assure that everyone uses the same basic editor setting, let's add an editorconfig file here to the root of the project. Make sure that you call it ".editorconfig", you need that leading dot, it is significant. And I will just paste in the contents here. As you can see, I am enforcing a two-space indentation style, the end-of-line should use a line feed, and we will be trimming white space and inserting a final newline automatically with this editorconfig. And now that I've added this file, all the future files that I create will be formatted based on these editorconfig settings. However, as we just saw in the slides, some editors like VSCode require a plug-in to provide editorconfig support. To install the editorconfig plug-in for your editor, go over to editorconfig.org and click on download a plug-in. Then just scroll down to find your editor, and I'll click on Visual Studio Code in blue, since that's what I'm using, I will copy the command that I need to paste in to the command palette. I'll hit command-P, and paste this in, and when I hit Enter, you can see the top result is what we're looking for. I'll click Install, and after it's installed, I will enable it. At this point, VSCode will need to be restarted, I'll hit OK, and when it fires back up now editorconfig will enforce any future files that I create. So when your team uses editorconfig, you just need to be sure that everyone installs the appropriate plug-in for the editor that they're using so that the settings are enforced.

## Summary

Let's wrap up. We started off by briefly considering a few popular and popular JavaScript editors, including Atom, WebStorm, Brackets, and VSCode. In this course, we'll use VSCode. We saw how editorconfig is a handy way to assure that our code contains consistent spacing, character sets, and more, regardless of which editors our teammates are using. So great, now that our editor is set up, in the next module we'll make another critical decision: it's time to choose a package manager so that we can easily pull in the dependencies that we need to build our applications.

# Package Management

## Intro

These days virtually every popular language has it's own package manager, yet JavaScript soldiered on for many years with no compelling package management story. Thankfully these days we have a variety of JavaScript Package Managers to choose from. In this module, let's begin by exploring the list of Package Managers we have to choose from. We'll ultimately go with npm for this course which probably didn't surprise you much and we're going to set it up for our demo project. We'll wrap up by looking at a couple interesting services that will scan your packages for security vulnerabilities. This way we can help protect ourselves from known vulnerabilities in the packages that we choose to install. Okay, let's get started with Package Management.

## Package Managers

Okay, of course you need a Package Manager because every language benefits from a standardized method for sharing code. JavaScript, not surprisingly has multiple options. Not too long ago, Bower was the most popular option. Bower branded itself as the Package Manager for the web, but today people have largely moved away from Bower and toward npm. Why? Well, Bower became popular by supporting the entire web platform and packaging libraries in a format that didn't require a build step. The thing is, these days nearly everyone has a build step because they're transpiling, minifying, linting and so on. The bottom line is, Bower has become mostly irrelevant. Npm has matured by fixing friction points for front end developers, node has grown wildly in popularity, and bundlers have continued to become more powerful. All of this has come together to make npm the defacto standard for JavaScript Package Managers. It's an extremely low friction way to handle packages. Today npm is clearly the most popular Package Manager for JavaScript. I haven't touched Bower in a long while because npm offers everything that I need so as far as Package Managers go, npm is my recommendation for you without hesitation, but there are certainly other interesting players out there. One is JSPM. JSPM stands for JavaScript Package Manager. JSPM allows you to install packages from its own list of repositories as well as from other locations, including npm, GitHub, and Bower. JSPM also bundles your code so we'll talk about JSPM more when we discuss bundlers. It's an interesting option that's worth considering and

JSPM is certainly useful, but npm combined with a powerful bundler is certainly the most popular option today, so that's what I'll use within this course. Finally, there are other less popular Package Managers out there like Jam and volo, but chances are you can find everything you need in npm. In fact, in many cases, you'll have a hard time of choosing between multiple good options for solving a given problem. Now that's a great problem to have.

Demo: Install Node and npm Packages

Let's jump back in the editor and get a few foundational pieces set up. We're going to install Node and npm and also create a package.json file that's going to store the references to all the packages we're going to use for our development environment. To get started on our environment set up, let's install the latest version of node. I prefer using the 6.x branch so that I can enjoy the latest features. Also, the 6.x branch loads modules four times faster than the 4.x version, but the 4.x version should work just fine for this course as well. Just make sure that you're running at least version four. Now if you already have node installed and you're concerned about upgrading and breaking existing projects, you can run multiple versions of node using nvm on Mac or nvm windows on Windows. When you install node, npm, node's package manager comes bundled along with it so in the next clip we'll add package.json to our project. Node's package manifest is package.json. Package.json stores the list of npm packages that we are using as well as the npm scripts that we'll end up setting up later. Now instead of installing all these packages one by one, let's just use the package.json in this jist. The real URL is long so, use this shortened URL to get here, this will save us time because we won't have to manually type the names of all the packages that we'll be using. These are the packages that we'll use throughout this course. You likely don't recognize some of these names and that's okay. I'll introduce them as we go along. I'm going to copy this and go back to the editor and create package.json here in the root. Just paste this into our editor, save it. Now we have our references to the packages that we'll use in this course. If you've never seen a package.json file before just understand we have the name of the project, the initial version, a description, some scripts which we'll be adding in later to help automate our processes. The author's name, the license for this code, and then a list of Production dependencies, and development dependencies since we're building a development environment, our dependencies are all listed down here and these are all npm packages with their version. Now that we have a packaged json file on our project, let's install the necessary npm packages. To do that, we'll open up our command line which in vs code is Integrated. I can open this Integrated Terminal. Of course if I preferred I could just use the terminal built into my operating system. On Mac it looks like this, on Windows you'll have a DOS command line instead or you could install git bash if you prefer a UNIX style command line on Windows. I'm just going to use the built in terminal for vs code. The nice thing about using a built in terminal on an editor is that it opens by default in your root directory for that project so you don't end up having to cd into here after opening a terminal. I really prefer using these built in terminals and to install our packages, I'll just type npm install. This should take a minute or two on your machine as it downloads all of these packages and places them within a folder called node modules in your project, but instead of waiting, I'll just speed this along through pure magic. As you do your install you may see a few warnings along the way and if you look at the node modules folder now we can see all of the different modules

that we'll potentially be using throughout this course. These are dependencies, but all of these dependencies have other dependencies each of those get installed in their own separate folder here in node modules. We just downloaded dozens of packages, wouldn't it be nice if we could scan these to help protect ourselves from known security vulnerabilities? Let's explore how we can do that in the next clip.

## Package Security

It's worth remembering that packages can be published npm by anyone, so that might make you a little bit paranoid. Now thankfully there are people working on this exact problem. Reire.js and the Node Security Platform are two ways that you can check your project's dependencies for known vulnerabilities. Node Security Platform is my preference at this time. It offers a simply command line interface that you can automate checking for security vulnerabilities automatically. All you do is call nsp check as part of your build and then it reports the number of vulnerabilities found if any. We're going to run node security project by hand, but you may want to consider adding the check as part of your start script. There are a number of options for when to run this and they all have their tradeoffs. The first and most obvious is to manually run node security check, but that's easy to forget. The second option is to run it as part of npm install but the packages you use may have security issues later so merely checking at install isn't sufficient. You can run it during production build or if you're using GitHub automatically as part of your pull request, but both of these options are a bit too late. By then you've already used the package so you have a lot of potentially expensive rework ahead if you need to select an alternative package. The final option is to check as part of npm start. This way each time you start development, the security status of your packages are checked. This does the downside of slowing start a bit and requiring a network connection to perform the check, but it has the advantage of notifying you quickly when a security issue exists. You'll see how to automate tasks like this in an upcoming module on automation.

## Demo: Node Security Platform

Let's install Node Security Platform so that we can automate security checks for our dependencies. To perform node security scanning, let's install the node security project globally so that we can run it directly on the command line. I'll say npm install globally nsp which stands for node security project. We'll wait a moment for that to install and now that we've installed it globally we can run it right here on the command line by just saying nsp check and what it will do is check all of our packages for any known vulnerabilities. You can see that it comes back right away saying that no known vulnerabilities are found. It looks like we're in good shape. Now you'll note that we installed this module globally here so that we could easily run it directly on the command line. In an upcoming module on automation, I'll show how we can call this from an npm script and thus avoid having to install it globally. Now that our Package Manager is set up, let's wrap up this module with a short summary.

## Summary

In this module, we saw that there are currently many Package Managers to choose from, but in reality, this choice is easy. Npm has clearly become the defacto standard and is now the largest Package Manager in the world. We created a package.json file which is the manifest that npm uses and in future modules we'll augment this file with scripts that will automate our development and production build processes. We wrapped up by looking at services for security scanning and ultimately installed Node Security Project and ran it on the command line. In the next module, let's review the long list of interesting Development Webservers to choose from and we'll set up a Development Webserver so that we can see our application shell run in a web browser.

## Development Web Server

### Intro

If you're working in JavaScript, chances are you'll want to run at least some of your code in the browser. So in this short module, let's quickly consider a few interesting options for development web servers. We'll configure our web server so it's automatically started as part of our development process and opens our application so we can view our results immediately every time that we hit save. And what if we want to quickly share our work and process with others? Well, we'll look at some interesting and useful services for making that quick and easy.

### Development Web Servers

Here are six interesting web servers to consider for JavaScript development. Let's discuss the merits of each. The absolute simplest way I've found to get this done is an MPM package called http-server. After installing it, you type http-server in the directory that you want to serve and it fires up a lightweight web server that serves the current directory as the web route. It really can't get simpler than this. And it's not just great because it's super lightweight and requires no configuration. As you can see, it also has an amazing logo. I mean, if the idea of strapping a rocket to a turtle doesn't move you, check your pulse. That turtle means business. Respect. Live-server is also lightweight. But it adds live-reloading capability so that every time you hit save your changes are immediately visible. So this is a nice improvement over http-server. Well assuming that you're willing to look past the fact that this project lacks a seriously awesome logo. You've likely heard of Express. Well it's still pretty lightweight. It's much more full-featured than the other options that we've discussed so far. And it's highly configurable. Unlike http-server and live-server, it's not just for static files. You can serve up complex APIs via node using Express as well. The advantage to choosing Express for your dev server is you can use it in production as well. So if you choose Express for your dev server, you have the advantage of running the same web server in all places. This allows you to create a common configuration and use it in all environments. If you're building APIs in node, then this is a big win that makes using Express as your development server an easy

choice. And while we're talking about Express, it's worth discussing its competitors, like koa and hapi. Express is the most popular and has the largest community, but koa is interesting because of its strong embrace of ES6 generators. And hapi was created by Wal-Mart labs, and boasts a compelling configuration model. If you're doing server side work in node, these three popular options are worth carefully considering. But if you choose any of these, you're likely to use these in production as well. So if you have an existing API layer using non JavaScript technologies like Ruby, Python, .NET or Java, then this is likely overkill for your development web server. There're also some interesting options that are specific to the bundler that you choose. We'll talk about bundlers more in an upcoming module. But if you're working with Browserify, then budo is a great option. Budo includes hot reloading, which means you can immediately see your changes reflected in the browser the moment you hit save. And if you choose a more comprehensive bundler like Webpack, you can enjoy its built-in web server. The advantage to this option is it avoids pulling in another dependency. And it's also very fast to load changes because it doesn't require generating any physical files. Webpack serves your app from memory. And just like budo, it also supports hot reloading. Again, we'll discuss bundlers more in an upcoming module, but it's worth noting that if you choose Browserify or Webpack as your bundler, there are popular dev server options for each. Finally, you should consider Browsersync. Browsersync is a free web server that has two really compelling features. First, Browsersync automatically sets up a dedicated IP address on your local network. So that you, or anyone that can hit the IP on your LAN can see your app. The second feature that it offers is really special. You can hit that same IP on multiple devices, and all the devices will remain in sync. And here I'm running the same application from two different web browsers, but they are being served from the same address via Browsersync. And you can see that as I navigate around and interact with the application, the two applications remain in sync. This is super useful for browser and device testing so you can assure that your work is rending properly on a variety of devices. I've seen people create a wall of devices and then use Browsersync to regularly get feedback that the design is working across a variety of screen sizes and platforms. You can even use Browsersync along with Webpack, Browserify, Gulp and more. There are detailed examples of using Browsersync with a variety of setups published here in the docs under recipes. So you can think of Browsersync as a powerful and free development web server, or as a way to augment your existing development web server with additional features. So these are six interesting development web servers to consider. One final note. To clarify, with the exception of Express, these web servers are not for production. They're merely for hosting your application for development on your local machine. We'll discuss production deployment in a later module.

Demo: Set up Express

For this course, let's use Express as our development web server because it's powerful, highly configurable, and extremely popular. We're going to need a web server for development, so let's use Express. Now, we already installed it when we ran MPM install earlier because Express is one of the mini packages listed in our package.json. So now we just need to configure Express. I like to keep all my build-related tools in a single folder. So let's create a folder called build scripts in the root of our project. Inside, let's create a new file and call it srcServer.js. So I'm going to follow the popular convention of

abbreviating source as S-R-C. Now this file will configure a web server that will serve up the files in our source directory. So to set this up we will call Express and we will need a reference to Path as well. And finally a reference to Open, which we'll use to open our site in the browser. And I'm going to set a variable here that stores the port that we are going to use in this course. We're going to use port 3000; of course feel free to choose a different port if that one isn't available on your machine. And then I'm going to create an instance of express. And set that to the variable app. Then what we want to do is tell Express which routes it should handle. For now, we're going to say any references to the root should be handled by this function. Which takes a request and response. And in this function, we will call res.sendfile, and I will use path to join... I'll use a special variable dirname that will get us the directory name that we're currently running in. I'm going to join that together with a path to our source directory which we haven't created yet. But within our source directory, that's where we will place our index.html file. And now we've declared our routing, we just need to tell Express that we'd like it to listen on the port redefined above. And then we'll add some err handling here, just say if there is an err, then go ahead and just log that to the console. Otherwise we will open up the website and I will just hard code in the address of the website. And the port that we will hit. And now that we're this far we should be able to open the terminal and say node buildScripts, 'cause we're going to reference the path where we put this. srcServer.js, so we're telling node to run this file. And when we do it's going to try to open it, but it will not be able to find that index file because we haven't created it yet. So let's go back over here and create a new folder called src. This is where we will place all of our source code. And I should've placed this at the root instead, so let me drag it up the root. And inside src, let's create a new file and we will call it index.html. I'll just paste a little boiler plate html in here to get us started. A little hello world. Hit save. Now I'll come back down here to the terminal, I'm going to hit ctrl+C to kill the previous process and re-run Express. And now we can see Express starting up on the port that we selected saying hello world, so we know that Express is handling our first call successfully. So it's coming in, it's seeing that we're requesting the root, and then it is sending the file that we specified here, and opening the application at this address on this port. I'll just add a semi colon that I'm missing here. So great, we now have Express configured. But in the next clip, let's talk about some easy ways to share our work-in-progess with other people.

## Sharing Work-in-progress

And while we're on the topic of development web servers there's a closely related topic which involves sharing your work-in-progess. As I mention, Browsersync is a handy way to share your work with coworkers. However, Browsersync doesn't expose a public IP for testing outside of your local network. So, what if you want to quickly share your work on the public web so that your client can review your work-in-progress? This used to be a hassle, but these days it's shockingly easy. Now I'm about to show you some novel services that you might not have heard of, and you might be thinking, why wouldn't I just put my work on a traditional cloud provider like AWS or Azure? The answer is, the services I'm about to show you make it hilariously easy to share your work without having to configure a cloud provider or pay for hosting of any kind. So if money and time are important to you, keep listening. I thought that might get your attention. Here are some ultra-low friction options for easily showing your customers

your work along the way. Localtunnel, ngrok, surge, and now. Let's talk about each of these in more detail. The lowest friction way that I've found is localtunnel. Localtunnel offers an elegant way to expose your local host via a public url. It punches a hole in your firewall so that your local machine can operate as a web server. The nice thing about localtunnel is how easy it is to get started. You install the localtunnel npm package globally, and then any time you want to expose your work to the public web, you start up your app and then type something like this: lt--port 3000. So I'm exposing my app, which is running on port 3000 with this command. And it's as simple as that. Quite slick. Ngrok is another very popular option to consider. Like localtunnel, ngrok punches a hole in your firewall so that your local machine can operate as a web server. It offers some additional features over localtunnel but requires some additional setup to support the extra power. With localtunnel, you simply install an npm package and you're rolling. With ngrok, you have to sign up, download and install the app, install an authtoken, and then you can finally start your app and start ngrok and specify the port that you'd like to use. None of this is a big deal, but it is a little extra setup friction. Now ngrok's advantage over local host is security. You can password protect access. With localtunnel, anyone with the url can access your app while your localtunnel is open. But the advantage of localtunnel of course is it's lower friction to get set up. Now takes a different approach than localtunnel and ngrok because it doesn't punch a hole in your firewall. Instead, it's an easy way to deploy your application up to the cloud so that others can review it. Any directory that contains a package.json can be uploaded to the cloud using one command: now. Each time than that you deploy to now, you're assigned a new unique url. And you can also use now for your real production deployment as well if desired. To use now, you install it globally and create a start script that opens your preferred web server such as Express. Then, any time you want to deploy the app, you just type 'now'. So now is a useful service if you're building a node.js application and want a low-friction way to host it in the cloud. But since now is publishing actual files to a public web server, it's a more permanent solution than ngrok or localtunnel. You don't need to keep your machine on for people to see your work-in-progess. And if your application has a server side component that's node-based, then now is a great option. Another similar service is Surge. Surge assumes that your app is just static html, JavaScript, and CSS files. So the downside of Surge is it only supports static files. But the upside is extreme simplicity. You don't have to punch a hole in your firewall to expose your work. Instead you can quickly move your static files up to their public web server. Surge publishes your static files to their service and then serves them at a public url. The process for getting started really couldn't be simpler. First you install Surge globally and then you type surge in your project directory. Now, the first time you run Surge you will be prompted to provide an email and password on the command line. Otherwise it will list your existing login and confirm the path that you'd like to serve. When you hit enter to accept a randomly generated url, it will upload your app and expose it on the url listed. Easy as that. And if you want to use Surge as a more permanent host, you can even use your own domain name. So with that approach, Surge becomes an easy way to do automated deployments via the command line. So bottom line, all these tools are handy and offer different advantages and constraints. If you're looking for the lowest-friction way to share your work and have no big security concerns, localtunnel is the easiest approach. It also works nicely with Browsersync and supports exposing any work on your local machine, regardless of platform. Ngrok offers a similar approach that's still straightforward but requires a little more initial setup work. In exchange, it offers more security by providing the ability to lockdown access behind a login. Surge and now take a fundamentally different approach by not punching a hole in your

firewall at all. Instead they push your static files up to their server. Both are quite easy to set up, but since your files are hosted on a public server, your local dev machine doesn't have to remain on. There's no dependency. This means that your hosting persists like traditional hosted solutions. Now the downside for Surge is it only hosts static files, whereas now also supports node.js projects.

Demo: Sharing Work-in-progress

Let's try localtunnel to see how easy it is to share our work-in-progess. To quickly share my work-in-progess I'm going to use localtunnel. So to get started let's intall localtunnel globally. So we'll say npm install localtunnel -g. And that g flag signifies that it should be installed globally. Now in the next module on automation I'll show how you can run this and other tools without needing to install the package globally. And now that it's installed we're ready to share whatever is running on our web server. So let's start our development web server using the command that we ran in the previous clip, which was node buildScripts srcServer.js. I'm missing an S. And now that we have this pulled up, we'll keep this running but I'm going to open a second terminal, so I'll just click the plus sign here. Of course if you're using a terminal outside of VS code, you could just open a second command line in this same directory. And now to run localtunnel I'll say lt and then I'm going to pass it the port that I'm using for my development web server which is 3000. So we're just telling localtunnel to create a tunnel and expose port 3000. And you can see what it does is it returns a random url to me. So now I can take this url and share it with anybody, and anybody with internet access could see my local running application at this url. If I go over and change the application in any way, I could come over here to index.html and add a question mark and hit save and now if I come back over here and reload we can see that reflected on this public url. So this is a great way to share my work-in-progess in a very low-friction way. And I can even make this more deterministic by adding in a second parameter here. And I can specify the sub-domain that I'd like. Let's say I'd like a sub-domain of cory. So it's a little easier to type and tell other people than this random set of characters. Now as long as no one else has requested this particular sub-domain at the time that I'm using it, then I should be able to come over here and hit it. If I had chosen a sub-domain that someone else was using then localtunnel would warn me of that at this time. But this is a nice way to get a shorter url that I can share with others. Now also, we're using Express for our web server, but I wanted to share an interesting trick. If you choose to use Browsersync for your web server instead, that synchronized browsing experience that if offers will continue to work when you're using localtunnel. So this way even if your mobile devices and your development machine are on separate networks, they can still interact in a synchronized browsing experience. So I'm not going to walk through it in this course but it's worth nothing that if you have a wall of devices that you want to test against your locally hosted work-in-progess, combining Browsersync with localtunnel is a compelling combination. And then once you're done with your work, just come back to the terminal where you're running localtunnel and hit ctrl+C to stop it. Now that I've stopped it, I come back over here and try to refresh, we can see that it no longer loads. And it will say that there's no active client anymore at this sub-domain. Let's wrap this module up with a short summary.

## Summary

In this module, we considered a variety of development web servers including ultra-light options like http-server and live-server, more full-featured options like Express, bundler-specific options like budo and Webpack dev server, and the amazing Browsersync which integrates with various options like Webpack dev server and Express. We also saw many low-friction and free ways to share our work-in-progess on the public web. We used localtunnel, since it offers a low-friction way to share our work. But ngrok, now, and Surge are all great options for quickly sharing work-in-progess as well. And depending on your needs, ngrok and Surge are both potential options for production hosting also. Now it's time to begin automating tasks, so in the next module let's explore popular options for task automation.

# Automation

## Intro

Hey, we're developers, so of course we don't want to do things manually. Automation is a necessity to assure that development builds and related tooling are integrated and utilized in a consistent manner. If we do our job right, all the team will benefit from rapid feedback along the way. So here's the plan. In this module we'll review the options for automating our development environment and builds, and after discussing the options I'll introduce npm scripts which is the automation tool that I recommend for gluing all of our tooling together in an elegant and repeatable manner.

## Automation Options

Of course you don't necessarily need any custom tooling to perform automation. You could simply use your operating system's command line, but as automation gets more complex, shell scripts may mean you're reinventing the wheel. So there are a variety of options for automating your development, environment, and build process today. The three most popular options for JavaScript automation today are Grunt, Gulp, and npm scripts. Let's briefly discuss each of these in more detail. Grunt was the first JavaScript task runner to become popular. Grunt is configured via a grunt file which focuses on configuration over code. The grunt file is basically a big chunk of JSON that configures Grunt to work with your plugins. Now Grunt is file-oriented, Grunt writes files to the disk after each step in an automation process, so it reads from disk as input to subsequent steps, and since it's been around for a long while, Grunt has a large ecosystem of plugins. However many people have since moved on to more modern task runners like Gulp. Gulp improves on Grunt's plugin model in a few key areas. First it focuses on in-memory streams which Gulp calls pipes. In short, with Gulp you don't have to write to disk after each step in a task. Instead you simply pipe the output of a previous step to the next step in memory. This means that Gulp is typically faster than Grunt because it avoids writing to disk. Gulp is configured via a Gulp file which unlike Grunt is code based rather than configuration based. So in Grunt you configure the tool, in Gulp you write actual JavaScript code in your gulp tasks so you have more

declarative power. Like Grunt, Gulp has a large plugin ecosystem. Gulp is very popular, and for a good reason. I'm a big fan of Gulp, there are already excellent courses on Pluralsight on both Grunt and Gulp if you're interested in learning more. However I've moved on to just using npm scripts. Npm scripts are declared in the script section of your package.json file. With npm scripts you can directly leverage all the power of your operating system's command line. And you can directly use anything that you can install via an npm package. This can be useful if you want to write a cross-platform friendly npm script by using npm packages for tasks that differ on Windows versus Mac and Linux. Examples of this include things like deleting files and parallelizing commands. Now you can call separate Node scripts when a single command line call gets too complicated, and npm scripts also offer convention based hooks for running other scripts before and after your scripts. And by choosing npm scripts, you can select tools from the world's largest package manager. There's really an overwhelming selection of tools to consider. It's a great problem to have. Now as I mentioned, although I enjoyed working in Gulp, I've moved on to just using npm scripts. So why npm scripts over Gulp or Grunt? Well, I appreciate the simplicity of using npm packages directly rather than through an abstraction layer of a plugin. See by choosing npm I avoid taking on an extra layer of abstraction that's required by Grunt and Gulp plugins. For example on a previous project I used Gulp, and I also used Gulp eslint which is the Gulp plugin for eslint. However there was a strange bug where it would stop watching files once I had a certain number of files in my project. So I had to figure out if the bug was in Gulp, eslint, the Gulp eslint plugin, or within my configuration. That's a lot of points for potential failure. In contrast with npm scripts I can eliminate my dependence on plugin authors. This means I don't have to wait for someone to create or update a Gulp plugin when a new tool or version is released. With npm, I just use the tool directly. I find that this makes debugging easier. In Gulp I was often trying to determine if the bug was in the underlying tool, the plugin, or my implementation. With npm scripts, I have one less point of potential failure to worry about. I find the documentation is also easier to work with. When working in Gulp I'd often have to switch between the Gulp plugins docs and the underlying tools docs. With npm I have one place to check for all my docs. I don't have to glue two different documentation sites together in my head. I found that npm scripts were quite easy to learn arguably even easier than Gulp and if you're not yet familiar with node in npm, that's okay. I'll fill you in as we go along. As I said it's quite easy to learn. In summary, npm scripts allow you to make command line calls, utilize npm packages, or even call separate scripts that use node so they give us all the power that we need for creating a robust build process. You can read more about why I prefer npm over Gulp and Grunt at this URL.

Demo: npm Scripts

So we've made our pick. Okay you're right, I made the pick. But I think it was a pretty easy call. Okay, anyway, let's automate with npm script so you can see how straightforward this is. Now npm scripts allow you to make command line calls, utilize npm packages, or even call separate scripts that use Node, so they give us all the power that we need for creating a robust build process. When working with npm scripts, it's popular to call separate files from the script section of your package.json file, right here. In a previous module we created source server.js which configures express to serve up our index.html. Now let's create an npm script that will start our development environment. By convention this script should

be called start, that way by convention we can just type npm start and it will run this command. And what we're going to do is just put the exact command that we've been typing manually right here within our package.json file under our script for start. And now we can open up our command line and type npm start. When we do, we can see that it starts up our application just like before but now we have less typing to do, we have something repeatable that we can work with. It's also worth noting that you don't need to type the run keyword when you're calling npm start or npm test. Both of those are commonly used so npm doesn't require you to type the run keyword for either of those scripts. In the next clip let's augment our application startup by displaying a friendly message when our developmental environment starts up.

Demo: Pre/Post Hooks

It would be nice if we received a helpful message when starting up our development environment. To do that lets create a file called start message.js here in build scripts. And I'll just paste in the content here, you can see I'm going to reference chalk which is a library that allows me to specify the color of the output that I'm displaying to the console, and we'll just display a message that says that we're starting the application in dev mode. Now I'd like to run this before our web server starts up so I'm going to use a handy convention if I read a script called pre start, then by convention it will run before our start script. And I'll just place it right here. And don't forget the comma here at the end. Because this is JSON. See npm scripts support convention based pre and post hooks so any scripts that you prefix with the word pre will run before the script with the same name. And any scripts you prefix with the word post will run after the script with the same name. So if I created a script called post start, it would run after start, and since this is called pre start, it will run by convention, before my start script. So now let's come down here and hit control c to kill the previous process, and now when we re-run npm start, we can see that the first thing we see is a nice message that says that we're starting the application in dev mode. Great. And now that we have the basics of npm scripts down, let's use it to automate some of the other processes that we've been doing manually in previous clips.

Demo: Create Security Check and Share Scripts

Now that we've chosen npm as our method for handling automation, we can automate some of the commands that we've run manually in previous steps. First, let's create a script for running Node security check. And we can call the script security check. Now, sure, typing npm run security check is more typing than simply saying nsp check, so you might be wondering why this is beneficial. Well let's think about the advantages to this approach. First, it's more descriptive. Npm run security check is a lot more descriptive than nsp check. Second, and this is the most important, remember how we had to install Node security check globally in a previous module so that we could run it directly on a command line? Well with npm scripts you don't have to do that, you don't need to install tools globally to run them. So let me clarify why we don't need to install it globally anymore. When you install node modules,

just saying npm install to install all the packages that are listed in package.json they get added to a dot bin folder right here, and all of the items here in dot bin are automatically in path when I call them from npm scripts, so you should see the nsp is down here. So this script, nsp, is in the path. All of these scripts in the bin folder are automatically added to the path so that way nobody needs to install these packages globally. Creating this dedicated npm script will allow us to automatically run this as part of the application startup process if desired. We'll see how to handle concurrent tasks in the next clip. So we just created a reusable script for running our security check, let's do the same thing now for local tunnel. So we will call this script share and it will just run local tunnel and open it on port 3000. Now just to prove that it works we can come down here and type npm run security check and see that that runs nsp check for me. And no known vulnerabilities are found. So we're in good shape. Of course it would also be nice to be able to run certain scripts in parallel, so we'll look at that in the next clip.

Demo: Concurrent Tasks

The pre and post hooks are handy when you want to run something before or after a script, but you'll likely find that you also want to run multiple things at the same time. To do that, we'll use a package called npm run all. This is a handy way to run multiple items at the same time in a cross-platform way. So in other words, it'll run on Linux, Unix, and Windows. Let's say that we'd like to run the security check each time we start the app at the same time we start up the web server. To do this we can change our start script to instead call npm run all and specify that we want to run some tasks in parallel. There's two tasks that we want to run here. We want to run the security check task and we'd also like to start up our web server. So let's take this and put it in a well-named script below and we'll call it open source. And now we can reference it right here on our start script. So now our start script is saying run all the tasks that I list over here on the right-hand side in parallel using npm run all. So let's try this out. I'll come down here and type npm start. And when I start we can see that it starts up our application as expected and it also ran our security check as part of that start script. And of course since we have a prescript set up, it also said it was starting the app in dev mode. You'd see we also get some other noise here, if this noise bothers you, you can always type npm start minus s, which means silent, and now we'll have a silencing of most of that noise, so all we see now is the messaging that we've explicitly asked for which is starting the app in dev mode and then any output from our security check. And we can see that it starts our application just fine. So great, we're now running different npm scripts in parallel using npm run all. Now there's another place that I'd like to run tasks in parallel for convenience. I'd like to have a single command that starts up the web server, and shares my work via local tunnel so that I don't have to manually open two separate terminals like we did in the previous module. To do this let's create a script called local tunnel. I'm going to rename this to local tunnel since that's specifically what it's doing is local tunnel, and then I'll create a new script below that actually handles the entire sharing process because what we want to do is run two things in parallel. We want to open up our development web server and start local tunnel, so that way local tunnel will expose port 3000. Test this out, I'm going to open up the terminal and then say npm run share. We can see that it starts up our application running at 3000 and it also starts up local tunnel in parallel. So I can copy this and see that our application's also exposed on a

public URL. So great this is a pretty handy way to share our work in progress with a single command. And that's the power of npm scripts in a nutshell. All right, let's close out this module with a summary.

## Summary

In this module, we reviewed the options for automating your development environment and built process including Grunt, Gulp, and npm scripts. We saw that Grunt popularized the idea of JavaScript based task automation, Gulp came along later and refined the process, and then npm scripts proved popular as well. We implemented a few simple npm scripts just to show how it works, and throughout the rest of the course we'll expand our usage of npm scripts to enhance and fully automate a robust build process. In the next module it's time to break away from plain old ES5 JavaScript and take a look at the many transpilers that we can consider when working with JavaScript.

## Transpiling

### Intro

People have complained about JavaScript for years. And justifiably so. It languished with no innovation for a full decade. It's a wonder it survived such stagnation. Yet, with the advent of ES6 in 2015, the language has finally grown up in a big way. And annual releases mean that we can look forward to more goodness every year. This is great, but it's also one of the many reasons transpiling has become so common. In this module, we'll begin by look at JavaScript's history and its future. This will help set the stage for understanding why transpilers are so desirable today. And why we're likely to continue using them for the foreseeable future. Then we'll take a glance at the long list of interesting transpilers to consider and finally we'll configure my recommendation for transpiling today, Babel. All right, let's begin by taking a short walk down memory lane.

### JavaScript Versions

I believe that to understand where JavaScript is and where it's headed, it helps to understand its history. Version one of JavaScript was born in 1997, created in a few short days by Brendan Eich. Who would've thought that it would have such longevity? A year later, ES2 was released. And a year later, we saw ES3. So far, so good. But then they tried to tackle a much more aggressive release. No one could come to an agreement so ultimately, ES4 never happened. So JavaScript stagnated for a full decade before ES5 was finally released in 2009, with a number of notable enhancements. And then, the waiting game began again. It took another six years, but ES6, also known as ES2015, was well worth the wait. We picked up arrow functions, const, let, modules, classes, destructuring, generators, and so much more. And the other huge piece of news was a commitment to annual releases. But the ES6 release was so huge that apparently it wore everyone out, because ES7 only had two features. The exponent operator and

array.prototype.includes. Yay? Hey, no complaining. This sure beats ten years of stagnation. Now ES8, also known as ES2017, is expected to add some long-awaited features, including Async Await, class props, and object spread. Many people are already enjoying these features today by using transpilers that shim in the support. And this is a good segue for us to talk about choosing a transpiler.

## Transpilers

So finally, modern JavaScript is great but you don't have to write your app in JavaScript. You can choose from a long list of transpilers. And if you're not already overwhelmed by choice, here is the point where things get just plain silly. Okay, hold on to your seat. There are literally over 100 languages that compile down to JavaScript today. So choosing is truly overwhelming. There's Dart, Elm, GWT. There's libraries that compile Haskell, Lisp, PHP, C#, Scala, Perl, Ruby, Erlang and many more. You can view the full list here at this URL. This feels overwhelming, but when you look at the options with significant adoption and maturity, only a few stand out to me. The two most popular ways to transpile JavaScript today are Babel and TypeScript. But Elm is an excellent example of a language that gives you a luxurious development experience but also compiles down to JavaScript. Let's talk about the merits of each of these. Babel's big obvious selling point is it allows you to enjoy all the new features of JavaScript, even those that are currently experimental in a standards-based way. So Babel has a clear mission. It transpiles the latest version of JavaScript down to ES5 so that you can use all of these new features, but run them everywhere that ES5 is supported. TypeScript is a superset of JavaScript. So just as ES6 added features to JavaScript, TypeScript adds additional functionality to JavaScript. Most notably, type annotations which add type safety to JavaScript. This type safety means that you can enjoy rich autocompletion support and safer refactoring by leaning on explicit type signatures. Ultimately, TypeScript can aid maintenance on large code bases by clarifying developer intent. As I mentioned, TypeScript is a superset of JavaScript. So just as ES6 expanded JavaScript's power, TypeScript further expands JavaScript's power. TypeScript supports ES6 and it adds in TypeScript-specific enhancement such as interfaces and obviously type annotations. Choosing between Babel and TypeScript is a tough call. Both are excellent options. And there are loyal fans on both sides. So let's consider the merits of each. The big sell with TypeScript is not surprisingly the usual benefits of type safety. You receive enhanced autocompletion support. And an enhanced code readability because you can clearly see the exact types expected for each argument. You can also refactor more safely because you have a type-safe compiler to lean on. TypeScript also offers additional features that aren't in the JS spec at all, like interfaces. So TypeScript is pretty awesome. And it looks like a slam dunk. Yet Babel continues to be popular as well for a variety of reasons. When you choose Babel, you're writing standardized JavaScript. This means that you can leverage the entire JavaScript ecosystem. Now that's a big win because it means any editors, libraries, or frameworks that you select are likely to be compatible with your code by default because you're writing completely plain JavaScript. Since TypeScript is a superset of JavaScript, some tools can't handle TypeScript. So with TypeScript, you may not be able to leverage all of the JavaScript ecosystem. One example is React. When it first came out, you couldn't use TypeScript with React because the JSX syntax clashed with TypeScript syntax. So React developers had to wait many months for TypeScript to add support. Another example is ESLint. At the time of this recording, ESLint

doesn't support TypeScript, though it's expected to finally add support here soon. And although TypeScript and Babel both allow you to write standardized JS, Babel has historically been quicker to add support for stage 0 through 4 features, also known as experimental features. This means that you can often try out experimental features more quickly in Babel than with TypeScript. And of course for TypeScript to do its magic, type definition files and type annotations are required. So as you add new libraries, you need to also assure a type definition file is available in your project. If one doesn't exist, you need to create one to enjoy TypeScript's benefits when working with that code. Now I'm a fan of both of these options, but personally, I found that both type safety and autocompletion are quite good when working in ES6 in a modern editor like WebStorm. Remember, since ES6 imports are statically analyzable, the editor can deterministically index your entire code base and thus provide reliable intellisense support on imports and the functions inside. When this is combined with the power of automated tests, Linting, Babel compilation, and great libraries like React that support declaring property types, I find that I rarely deal with tricky type-related issues at runtime. That long list of goodness that I just described catches the vast majority of issues for me at compile time even though I'm working in Babel instead of TypeScript. Bottom line, both of these are excellent options to consider. So what about Elm? Well, Elm looks nothing like JavaScript but it transpiles down to JavaScript. Since Elm isn't JavaScript, you can enjoy a variety of enhancements over writing in JS. Elm offers a very clean syntax. No semicolons, parenthesis on function calls are optional, and everything is an expression. With Elm you get immutable data structures automatically. And Elm provides helpful error messages at compile time. Many of them are written in a very approachable yet descriptive style. Elm's compiler strives to be more like an assistant than an adversary. And here's wild fact. It's supposedly impossible to get a runtime error in Elm. If the application compiles, then it will run. You certainly can't say that about Babel or TypeScript. And you can still interop with existing JavaScript as needed so you're not required to use Elm for everything. This is a pretty compelling sales pitch. Of course, the downside of Elm is you do have to get everyone on your team on board with learning a completely new language and getting comfortable with functional paradigm. So there's definitely a learning curve involved in shifting over to Elm. But if you're thinking about choosing a language that compiles down to JavaScript, Elm should be at the top of your list for languages to consider. In summary, there's no wrong answer here. All three of these are excellent options but for this course I'm going to use Babel since it's currently the most universally popular.

## Babel Configuration

Babel offers two methods for configuration. A .babelrc file and within package.json. The most common approach is to use a dedicated .babelrc configuration file. This configuration file should be placed in your project's root. The advantage to using the .babelrc configuration file is it's not npm specific. So even if you're working on a project that doesn't use npm, you can configure Babel using a .babelrc file. It's also often easier to read the configuration in a separate file since it's isolated from all your other configuration and package.json. However, the advantage of placing your Babel config inside package.json is less file sprawl. It reduces the number of config files in your project and if you choose to configure Babel via package.json, you place the configuration in a section called babel. So it ends up

looking something like this. It's also worth noting that Babel can transpile experimental features. You can view links to the different experimental JavaScript features under the Plugins page on Babel's website. As you can see, JavaScript features progress through different stages. They're initially just ideas, which is called stage-0. Stage-1 is a formal proposal. Stage-2 is a draft spec. Stage-3 is a completed spec with the initial browser implementations. So as you can see, risk goes down as the stages progress. Bottom line, if you want to use these experimental features, you'll want to install the appropriate plugin and configure Babel to use this plugin. For this course, I'm going to use only standardized JavaScript features but I wanted you to be aware of this option if you prefer to use experimental features. And as a quick note, if you're deploying JavaScript code to a known environment such as Node or Electron, then you're lucky because you know a lot about your deployment environment. Remember, Electron apps run in Chrome behind the scenes and since Node is a server-side technology, you make the call on what version of Node to run on your server. So this gives you the power to intelligently only transpile the new parts of JavaScript that your target environment doesn't understand. Both Node and Electron's environments have recently added excellent support for ES6, so there aren't many features that you need to transpile. So to avoid transpiling features unnecessarily, you can select from one of these two plugins for Babel. They both accomplish the same thing, but the one on top is Node-specific because it determines what needs to be transpiled by sniffing for your Node version. The one on the bottom sniffs for features instead and thus, isn't Node-specific. Both options are intelligent enough to only transpile the pieces that your environment doesn't currently support.

## Transpiling Build Scripts

Another tough decision is whether to transpile your build scripts or to continue using ES5 and Node's common JS style for your build scripts. ES5 is the most efficient approach since transpiling does slightly slow down your build scripts. And it also helps avoid depending on a transpiler at all. But of course, using ES6 or newer is attractive because we can enjoy all of JavaScript's latest features. And this also means that all the code in our project uses the same style. This makes it easier to read and understand and avoids confusing people with intermixing ES5 and common JS in our build scripts with ES6 or newer code and module import styles in the rest of our application. Transpiling our build scripts also means that we can use the same linting rules everywhere, as you'll see when we set up linting in an upcoming module. And finally, as Node's modern JavaScript support improves, depending on the features that we use, we can eventually remove the need for transpiling altogether. For this course, I'm going to transpile our build scripts as well so that we can enjoy modern JavaScript when writing our build tooling.

## Demo: Set Up Babel

All right, decision made. It's time to set up Babel. Let's update our build process to transpile our code the moment that we hit Save. Since we've decided to use the latest version of JavaScript in this project we need to transpile down to ES5 to assure that it runs in environments that don't yet fully support the latest versions of JavaScript. To do that, we'll use Babel. Now Babel's configured via .babelrc so I'm going

to create a new file in the root of our project called .babelrc. If we look in package.json, we can see a number of Babel-related packages. To actually configure Babel doesn't take much code. All we need to declare in here is one preset that we are using latest. This is a shortcut for us saying we want to use all the latest standardized JavaScript features. So at the time of this recording, ES2016 is the latest version, but when ES2017 comes out we can continue to just have this preset here and we'll be running the latest version of standardized JavaScript. And that's all it takes to configure Babel to transpile our code. So let's confirm it works by using Babel Node to transpile one of our build scripts. Now remember, all of our code is currently written in ES5, but let's go into startMessage and change one item to ES6. Currently we're using the common JS pattern, which is used for Node, but we can now use the module syntax instead. And say import chalk from chalk. So this is a standard that was implemented in ES6. And now if we open the command line and try to run our start script, we should see it fail and it will fail because we are calling some code that Node doesn't understand. It doesn't know what to do with this import keyword. So to fix this, we can use Babel Node instead of calling Node directly. So on my prestart command, I can say babel-node. And now will transpile our start message code before passing it on to Node. So with that change, I should be able to come down here and re-run npm start and see that we get our starting app just like before. So now it all runs just fine even though I'm now using some ES6 code right here on line one. So now we know that Babel is successfully transpiling our build script, but in the next module we'll see how to set up bundling and as part of that process we'll transpile some application code as well. And now that we have Babel available, I'm going to go ahead and use Babel on our source server JS as well. And now that I am, it means I can go into srcServer and convert this to use the import style syntax or JavaScript modules. And then also I can use the const keyword, which was introduced in ES6 as well. And Babel will just transpile all this down to make sure that Node can work with all of this. And we can of course prove this now by reopening the terminal and saying npm start. We can see that it still starts the app, still does the security check. Everything passes, so we know that our conversion over to ES6 worked just fine. It's great, now we can use modern JavaScript in all of our build scripts. Now you're probably wondering how we're going to use this for actual application code. Well, in the module, I'll answer that as we talk about setting up bundling as part of our process. That way, we'll be able to transpile our application code using Babel as well.

Summary

In this module, we looked at JavaScript's history so that could understand the present and where it's headed in the future. Annual releases mean we'll likely be transpiling down to some baseline level for the foreseeable future because there will always be some new features that aren't yet supported in all environments. We saw the overwhelming list of over 100 languages that compile down to JavaScript. And we spent some time looking at three excellent options. Babel, TypeScript, and Elm. But there are dozens and dozens more to consider. I ultimately chose to use Babel for this course. We reviewed some decisions for configuring Babel, including the merits of using .babelrc versus package.json. We saw that Babel supports transpiling experimental features via a rich plugin model and we decided to transpile our build scripts as well so that we could enjoy modern JavaScript in both our application code and our build scripts. After all that foundation, we successfully set up Babel to transpile our code down to ES5 to

assure that it runs in all recent browsers. Great, our development environment is really taking shape. And now that we've selected npm as our package manager, npm scripts to handle automation, and Babel for transpiling, it's time to select and set up a bundler that will package our code for the proper environment. So let's do that next.

## Bundling

### Intro

These days, when you write JavaScript, it likely needs bundled up for usage. Why? Well, if you're a JavaScript developer, it's important to understand that NPM packages use the commonJS pattern. node can handle this just fine, but browsers don't understand it. So you need to bundle NPM packages into a format that the browser can consume. But bundlers aren't just for apps that run in the browser. You may use a bundler to package any JavaScript into a single file. Or strategically into separate files, for different portions of your app. Imagine you've created an app with five separate pages. A powerful bundler can intelligently create separate bundles of JavaScript for each page. That way, the user only has to download the relevant JavaScript for the first page on initial load. This saves bandwidth and speeds page loads. Finally, remember that bundlers aren't just for the Web. You may want to use bundlers if you're coding in node as well, since node's require is slow. But bundling your code for node, you can compile away the require calls, which can often improve performance. So in this module, let's begin by considering the various module formats available in JavaScript, including AMD, CommonJS, UMD, and ES6 modules. Then we'll discuss the various bundlers to consider, like Webpack, Browserify, Rollup, and JSPM. And we'll close out this short module by implicating ES6 modules and bundling them up with our NPM packages via my suggested bundler, Webpack.

### Module Formats

If you've been coding in JavaScript for long, you're probably not surprised that there are currently not less than five different module formats out there to choose from. That sounds overwhelming, but as you'll see in a moment, there's arguably only one option that makes sense for new development today. The five common module formats include Immediately-Invoked Function Expressions, also known as IIFEs. Asynchronous Module Definition, also known as AMD. CommonJS, which was popularized by node. The Universal Module Definition, which blends AMD with CommonJS. And finally, the ES6 module format. Let's look at examples of each of these approaches. First, we had good old global variables. We've known for years that global variables should be avoided, because it makes code harder to understand and maintain. When we write these, people pick on us in code reviews. And rightly so. So we came up with interesting techniques like Immediately-Invoked Function Expressions, also called IIFEs. This is a way to encapsulate our JavaScript. We also use tools like RequireJS, which utilize the Asynchronous Module Definition pattern to encapsulate our code, also known as AMD. However, all of these approaches should now be considered a thing of the past. They should be avoided because better,

more standardized approaches exist. So what are those formats? Well, these final two. If you're working in node, you can continue to use CommonJS. But if you're working in ES6, or in newer versions of JavaScript, you can finally enjoy the power of ES6 modules. Yes, we finally have a standards-based way of encapsulating our code. And for future application development, we should choose ES6 modules. I'll discuss why in the next clip.

## Why ES6 Modules?

ES6 modules are also known as ES2015 modules, since ES2015 was the official name of the 2015 release, which released this feature. Now as we discussed in a previous module, all future JavaScript versions will be named after the year of their release. So anyway, why should we choose ES6 modules? Well first, they're standardized. This means in the future, when the platforms you run on have full support for ES6 and modules, you won't have to transpile your code. It also means anyone joining your team is more likely to feel comfortable with your code. ES6 modules are also a win, because you can't declare them dynamically. Now that sounds like a drawback because it reduces power, but this reduced power was a deliberate design decision, because it makes our code statically analyzable. That's a fancy way of saying that it means that our code can be read and analyzed in a predictable way, because the behavior of our imports can't be changed at runtime. When code can be analyzed this way, we get benefits that I alluded to earlier when discussing editors. We get an improved autocompletion support, since your editor can determine clearly what functions are currently in scope from each imported module. This power leads to other wins, such as the ability to quickly alert you to invalid imports, to functions that don't exist, and so on. Effectively, choosing ES6 modules means that your code fails fast. You find out about your mistakes more quickly, and often in a clearer manner. ES6 imports also enable tree shaking, also known as dead code elimination. This is a feature coming soon in Webpack 2, and already available in alternative bundlers like Rollup, which we'll discuss in a moment. In short, this feature reduces the size of your final production code by eliminating unused code. And for tree shaking to work, we need statically analyzable code, which is exactly what we get with ES6 modules. ES6 modules are also easier to read than the more redundant alternatives, like AMD and UMD. And you can further clean up your code using named imports. Named imports allow you to easily declare variables that reference pieces of the file that you're importing. And default exports, which specify clearly how others can consume your module. Bottom line, although there are many ways to handle modules in JavaScript, if you're writing new code today, ES6 modules are the clear, logical, and attractive way to get things done. So ES6 modules are the format that I'll be using to modularize our code through the rest of the course.

## Choosing a Bundler

Now that we've picked a module format, our next decision is to select a bundler. Bundlers take all your JavaScript files, and intelligently package them for a target environment, such as a browser or node. And as you'll see, some add a variety of additional features on top of that. The first bundler to reach mass adoption was RequireJS. RequireJS popularized the AMD pattern, also known as Asynchronous Module

Definition, that we saw on a previous slide. However, since we're moving on to ES6 modules today, RequireJS has largely fallen out of favor. So let's shift our focus to more modern bundlers that we should be considering today. Again, no surprise, there's a number of options to choose from. Browserify, Webpack, Rollup, and JSPM, are just four compelling options to consider. All of these bundlers can handle the various JavaScript module styles that I mentioned earlier. And as their name implies, all of these projects can bundle your code up into a single minified file. Now, let's discuss what makes each of these unique in more detail. Browserify is special, because it was the original. It helped popularize the idea of using NPM packages on the Web. So at its core, Browserify is conceptually quite simple. Browserify basically takes all your code, and any NPM packages that you reference, and bundles it all up in a way that browsers can use. More precisely, Browserify bundles code that uses the CommonJS pattern, which node helped popularize. Browserify doesn't do much more than that out of the box, because its design is plugin-based. So Browserify boasts a large ecosystem of plugins for adding functionality like minification, linting, transpiling, and so on. Webpack is interesting, because it can handle much more than just JavaScript. Webpack offers a huge ecosystem of loaders, so you can easily teach Webpack to intelligently handle your CSS, images, fonts, and more. With Webpack, you can import all these file types just like you do JavaScript. And it can intelligently bundle your application accordingly. Webpack's even smart enough to inline images in styles, if they're small enough to justify saving an HKP request. And as we saw earlier in the development web server module, Webpack includes a built-in hot reloading Web server. Webpack serves files from memory, which speeds development, builds, and automatically updates client-side state to reflect code changes. And not just JavaScript, but also your styles, images, and HTML. Rollup is interesting because it's the first bundler to offer tree shaking. As I mentioned earlier, this means it eliminates code that you're not using from the final bundle. This can significantly reduce bundle size, saving you bandwidth and speeding page loads. Especially on mobile, which is slower to parse JavaScript. Tree shaking bundlers like Rollup have been shown to reduce bundle size by well over 20%. Even in a trivial app like TodoMVC. But Rollup isn't just about tree shaking. Rollup also produces code that's faster than Webpack and Browserify. Because Webpack and Browserify have to inline a module loader, whereas Rollup moves all code into the same scope. In fact, Dan Bucholtz on Twitter pointed out that in his testing, Webpack took around 1,100 milliseconds to start up, versus only 350 milliseconds with Rollup. But be forewarned, Rollup is still quite new, compared to established competitors like Webpack and Browserify. So choosing Rollup means you'll find fewer online examples and companion libraries, at this point. Now that said, Rollup is a great choice if you're building a library, but tools like Webpack and Browserify are currently better suited for application developers, because of features like hot reloading and code splitting, which aren't yet supported in Rollup. Finally, JSPM is interesting because it uses SystemJS behind the scenes. SystemJS is a universal module loader. And by universal, I mean it can load a variety of JavaScript module styles, including AMD, CommonJS, and ES6 modules. But what makes SystemJS interesting, is that unlike tools like Browserify and Webpack that bundle your code up at compile time, JSPM supports loading modules at runtime. JSPM actually has its own package manager. So with JSPM, you can install dependencies from NPM, git, and more. Finally, to add another interesting wrinkle, JSPM also uses Rollup in its builder. So you get to enjoy Rollup's tree shaking benefits as well. So which do I recommend? These are all compelling options for different reasons. Browserify is conceptually simple and approachable. And it's worth noting that you can also use Rollup with Browserify via a plugin. However, Browserify's simplicity

means that you have to do more plumbing for yourself for related problems like HTML, CSS, and images, than with more comprehensive and integrated tools like Webpack. Rollup is compelling, both for its ability to do tree shaking, and its enhanced performance characteristics due to its bundling style. But it's not as mature and full-featured as the other options, so I hesitate to recommend using it directly at this time. Instead, it likely makes more sense to consider using Rollup, along with tools like Browserify or JSPM. Or perhaps using Rollup by itself, if you're a library author. Finally, JSPM is interesting because of its runtime loader, and the fact that it's the only one on this list with its own built-in package manager. So we can leverage NPM, git, and more. Of course, I have to choose one of these for the course. And Browserify and Webpack are certainly the most popular options on this list. Webpack is the most full-featured and powerful of the bunch, so Webpack's what I'm going to use in this course. But let's talk a bit more about why I'm choosing Webpack. First, as I mentioned a moment ago, what really makes Webpack special is its ability to intelligently bundle and generate your CSS, images, fonts, and even your HTML. This means you can do things like inline images via Base64 encoding, when they're small enough to justify saving an HTP request. And it means you can hot reload your CSS changes in memory, via the built-in Web server. Webpack offers strategic bundle splitting. This helps avoid your users downloading all of your JS up front. Instead, you can generate separate bundles for different sections of your app, so they're downloaded on demand. We'll see how to do this in the production build module, later in this course. Webpack also includes a built-in Web server that supports hot module reloading. Which means depending on the library or framework you're using, you may be able to hit save and immediately see the changes without having to do a full refresh. This helps speed development because you don't lose client-side state. This is especially helpful on complicated user interfaces and multi-step forms, because you don't have to refill the form every time to test your changes. The code is hot reloaded in place, and your existing input field values and so on are maintained. Finally, Webpack 2 is nearly here, and will offer tree shaking as one of its key new features. And, for what it's worth, I posted a poll on Twitter. And after over 2,000 votes, Webpack was the clear leader, with over 78% of people saying that they're using Webpack. So although all these bundlers offer unique benefits, I'm going to use Webpack in this course.

Demo: Configuring Webpack

Alright, now that we've chosen Webpack for the path forward in this course, let's set it up to bundle our code. Alright, time for Webpack. Webpack does so much that it's hard to describe in a sentence, but basically Webpack will bundle all our assets up into a single file that runs in our target environment. For our demo app, our target environment is the Web. But as you'll see later, we can also use Webpack to generate multiple bundles instead of just one. Now, Webpack is configured via webpackconfig.js. By convention, it's placed at the root of your project. So let's create that. Now, I'm going to call it webpack.config.dev.js. Later in the course, we'll create a production build that gives you a clear picture of how to use all this tooling to create a lightweight, bundled, and minified app for production. But since this config file is for dev, we'll add some development-specific configurations here. Since we have Babel configured to transpile our code, we can write our Webpack config using ES6 features. I'm just going to paste this in, and then we can walk through the configuration. Webpack is configured via a single object that we define here in Webpack config. The Webpack docs go into great detail on the wide variety of

options that you can configure, but I'm showing a very simple setup for this course. Now, let's walk through each setting briefly, so that you're clear what I'm doing. As you can see, Webpack is configured by just exporting an object. And I set debug to true. This enables some debugging information as we run our build. And I set the devtool to inline-source-map. There are a number of devtools to consider. The basic trade off here is compilation speed versus quality. Higher quality sourcemap settings take longer to generate. And I'll talk more about the sourcemap settings in a later clip, when we focus on sourcemaps. I'm setting noInfo to false. Yes, this name is unfortunate, but setting it to false means that Webpack will display a list of all the files that it's bundling. I typically turn off this data during real developments, since it adds a lot of noise to the command line. But we'll have it on at first, just so we can see what's going on. Now we need to define the entry point of our application. Now, you can pass Webpack in array of entry points. And this is a good way to inject middleware for hot reloading. But I'm going to ignore hot reloading, just to keep things simple here. And just define our application's entry point as src/index. And I use the magic global __dirname, which is part of node. So that I can make sure that we get a full path here. And I'm using the path package that comes with node, to get this done. And you can see, I use the same approach anywhere that I'm resolving paths within this file. And we're targeting the Web for this demo app. But we could, of course, set this to node if we were using Webpack to build an app running in node. And that would change the way that Webpack bundles our code, so that node could work with it instead of the browser. And there are other notable targets that you could include here, like electron, which is useful for building desktop-style apps with JavaScript. Now we need to define the output. Here, we tell Webpack where it should create our dev bundle. Now, this is confusing because as you'll see, with our development configuration, Webpack won't actually generate any physical files. It will just create a bundle and memory, and serve it to the browser. But we need to define the path and name, so it can simulate the physical file's existence. We'll use node's dirname variable to get the currently directory, and specify that our app will ultimately run from the source folder. But again, note that this won't actually write any files. We'll set up a build process that generates files for production, later in this course. And you can see that we're going to call our bundle, bundle.js. Okay, we're almost done. Now here, we could optionally define some plugins to enhance Webpack's power. Examples include hot reloading, catching errors, linting styles, and much more. We'll add a plugin here later in the course, but for now we don't need one. So we'll just leave this array empty. Alright, one last property. We need to tell Webpack the file types that we want it to handle. Webpack calls this concept loaders. Loaders teach Webpack how to handle different file types. As you can see, we want to handle JavaScript. The great thing about Webpack is, it can handle a lot more than just JavaScript, as you can see with the second loader. I've taught Webpack to handle CSS as well. I could add other loaders here to handle SASS, LESS, images, and more. So why teach Webpack to handle more than JavaScript? Well, adding a loader here means that I can import those file types at the top of my JavaScript files, and Webpack will intelligently bundle the files together for me. It's also worth noting that there are various alternative syntaxes that work here. So other examples you see may look a little bit different. Now, I'm just scatching the surface on all you can do with Webpack. I show a more complex approach in my React with Redux in ES6 course. Or, there's an excellent Webpack course here on Pluralsight, if you want to dive deeper. Now, if you've never seen Webpack before, that probably felt intimidating. But I've come to really appreciate how terse Webpack's setup is. We just declared many decisions in under 30 lines of code. Webpack is designed to cover our use case really well, which means we don't have to write much code to get a lot of power

from our built process. Okay, we've set up Webpack to bundle our app's JavaScript, but we need to update our development Web server to run Webpack and serve up our bundled JavaScript. So let's do that in the next clip.

Demo: Configure Webpack with Express

We've configured Webpack, but to actually put it to use, we need to also set up our development server to serve our Webpack bundle. So let's configure express to make that happen. To do so, we'll go over into the buildScripts folder and open srcServer, since this is where we configure express. And first, I'll add two imports to the file. We will import Webpack, and we will import the Webpack config that we just created. Then down here, below where we have created an instance of express, let's call Webpack and pass it to config, that we referenced up here above. So now we have a reference to the Webpack compiler. And we can put that to use down here on line 11. And what we'll do is call app.use, which is a way for us to tell express other things we'd like to use. We're going to tell express to use our Webpack dev middleware. And we'll pass it, the compiler, that we set up up here on line 9. Inside of here, we'll define two options. We'll it not to display any special info, and then also we'll configure our public path. So here, we're just referencing a variable that we defined when we set up our Webpack config. And that's all it takes for us to integrate Webpack with express. Of course, this isn't very useful yet, since our demo app doesn't contain any JavaScript yet. Our application does nothing but say, Hello World! So in the next clip, let's fix that by creating an application entry point to test all this out.

Demo: Create App Entry Point

We've wired up Webpack to bundle our JavaScript, and we've set up express as our dev-server to serve our app, but we haven't actually written any JavaScript yet. So now let's create our JavaScript file, and see Webpack in action. When we set up Webpack config, we set the entry point for our application to index.js. So let's create index.js in the root of our source directory. Now, in an upcoming module, we'll build an app to show how the entire development environment works. But for now let's do something simple, just to show how Webpack is able to bundle two JavaScript files into a single bundle. So here you can see that I am using the numeral package that was installed at the beginning of the course, with all our other dependencies that are listed in package.json. This is a handy library for formatting numbers. So I set a constant with a value of 1,000. And then I formatted it, giving it this format string here. And I called this courseValue. And then I'm just going to use console.log to output a message right here. Okay, the statement that it outputs might be a slight exaggeration. Also, take note I'm using the ES6 template string feature, so make sure that you're using backticks here. This tells JavaScript to parse any variable placeholders that it finds inside. So now we've actually written some JavaScript for Webpack to bundle. But of course, we need to reference the final bundle in our HTML file. So let's open that up, and we'll add the appropriate script tag right here. And remember, when we configured Webpack, we came over here and told it to place bundle.js in the root of our source directory. Again, it's not going to actually write a physical file for development, but it will simulate its existence in this directory. And that's why

right here, I can just say bundle.js. Alright. Let's fire up the app and see if we've made magic happen. Before we do, I need to make sure to save my changes to srcServer. Alright, let's fire up the app and see if we can make magic happen here. Okay, we can still see our Hello World! message, but if we hit inspect, now we can see that we are writing the console.log, like we expected to. And if we go to the Network tab and refresh, we can see that bundle.js is getting sent down, like we would expect it to. So numeral-js, the library that we use to format this number, should be here inside of our bundle. And I'm going to scroll down, so that we can look at our source code. If we go to line 63 in the bundle, here's the code that we wrote. You can see that the template string that we used has been transpiled down to ES5. And if we scroll down a little bit further to line 70, you can find the numeral.js package. So we know that all of our JavaScript is now getting bundled into a single file by Webpack. Now, remember how we also taught Webpack how to handle CSS, by defining a CSS loader in our Webpack config? Let's put that to use as well. Let's handle a style sheet with Webpack in the next clip.

## Demo: Handling CSS with Webpack

In a previous clip, we taught Webpack how to handle CSS by defining a CSS loader in our Webpack config. So now, let's put it to use. Let's create a simple style sheet in the project root, called index.css. And I'll just paste in two simple styles, so that we can prove that this is working. Now, since we taught Webpack how to handle CSS, now all we have to do to put this to use is add a single line in our application's entry point, which is index.js. And all I have to say at the top of this file, is import index.css. Just like I would import a JavaScript file, I can import a CSS file. Looks pretty weird, but trust me, it works. So I'll just open the terminal. Restart our app. And now we can see that our Hello World! has a different font applied, because our CSS style sheet is now being applied. So you might be wondering, how did that work? Well, behind the scenes, Webpack parsed our style sheet. And then it used JavaScript to inject the style sheet onto the page. Let's come over to the Network tab and reload. If we pull up bundle.js, let's see if we can find our style sheet right in here. Search for the word "padding". And there it is, on line 102. You can see that our style sheet was bundled into our JavaScript, and then is being injected onto the page. Now, we could use a similar approach to handle SASS, LESS, images, and more. Of course, for production you'll likely want to generate a traditional separate file. And I'll show how to do that in the production build module at the end of this course. Now, as I scroll through the code, you might notice that it doesn't look like our source code. That's because this is the transpiled and bundled code, that Babel and Webpack produced. So you might be wondering how we can debug our transpiled and bundled JavaScript code. Well, the answer is sourcemaps. So let's explore sourcemaps in the next clip.

## Sourcemaps

Now that we're bundling our code, there's another important tool that we need: Sourcemaps. Once we start bundling, minifying, and transpiling our code, we create a new problem. Our code becomes an impossible-to-read mess when it's running in the browser. So you might be wondering, how do I debug?

Hey, that's commendable that you're concerned. Thankfully, this is a solvable problem. The solution is to generate a sourcemap. Sourcemaps map the bundled, transpiled, and minified code back to the original source. This means that when we open our browser developer tools and try to inspect our code, we'll see the original ES6 source code that we used. It's like magic. Now, the sourcemaps can be generated automatically as part of our build process. You might be wondering how minifying the code actually saves any bandwidth, if we have to generate a big map back to the original source. That's a good question. Credit to you, very perceptive. The beauty of sourcemaps is they're only downloaded if you open the developer tools. So this way, your users won't even download the sourcemaps, but they'll be available for you in case an issue arises, in either your development environment, or in production. So effectively, sourcemaps give you all the benefits of being able to read your original code, without any additional cost to regular users.

Demo: Debugging via Sourcemaps

Alright. Now that we've set the stage, let's configure our build to automatically generate sourcemaps as part of the bundling process. When we set up Webpack for development, we told Webpack to generate a sourcemap by specifying the devtool setting that you see here. There are many potential settings to consider, but I'm using inline sourcemap for this course. I encourage you to experiment with the different settings to find one that's best for you. As you can see from this table, the basic trade-off is between sourcemap quality and speed. Higher quality sourcemaps take a little more time to create, so you'll notice a little more delay on larger apps. Let's jump back to our source code. My preferred approach for setting breakpoints, is to type the word debugger on the line where I'd like the breakpoint to hit. So let's set debugger right here. I'll hit save. And now, let's jump back over to the browser and reload. And when we do, we can see our breakpoint hit. As you can see, the browser sees the debugger statement, and it breaks on the line where I typed debugger. And since we're using sourcemaps, the original code that we wrote is displayed in the console. Even though the actual code that's running in the browser looks like the code that I showed you earlier, that was hard to read. And this is really handy, because in a later module, we'll set up a production build that minifies our code for production. And still, because of our sourcemap, we'll be able to easily read the code just like you see here. Because we'll be seeing our original source code. And again, if you want to see the source code that's running in the browser, you can click on bundle.js, to see the transpiled and bundled code that's actually being parsed by the browser. Alright, let's wrap up this module. Time for another summary.

Summary

In this module, we began by considering our options for bundling our code under a usable and encapsulated modules. We briefly looked at IIFEs, AMD, UMD, and CommonJS. But we saw that ES6 is the future, because it's standardized and statically analyzable, which enables rich features such as autocompletion supoprt, deterministic refactoring, and reduced bundle sizes via tree shaking, assuming that you select a bundler that supports it. Then we moved on to discussing bundlers. I chose Webpack

for this course, because it's very popular, extremely powerful, and highly configurable. But Browserify, Rollup, and JSPM are all excellent alternatives to consider. Then we implemented ES6 modules and bundled our code via Webpack. And we closed out this module by discussing and generating sourcemaps. Sourcemaps are awesome because they allow us to see our original source code in the browser when we open the developer tools. This way we can set breakpoints and debug in the browser, even after our code has been transpiled, bundled, and minified. And since they're only requested by the browser when devtools are open, they don't slow down the customers' experience in any way. In the next module, let's protect ourselves from mistakes, and enforce consistency in our code base. It's time to set up an automated linting process, so that we're notified when we make typos and errors, as soon as possible.

## Linting

### Intro

You shouldn't have to keep track of your team's entire list of coding standards in your head. As much as possible, we should automate the task away. And we should be notified immediately when we make a typo. JavaScript linters can deliver on both of these concerns. Today's linters are so powerful that they can catch many errors at compile time. It's wonderful finding out that you made a mistake the moment you hit Save, rather than waiting until run time to hunt some cryptic issue down. So in this module, we'll begin by asking why do you need a linter at all? Then we'll quickly consider the linting tools available. We'll spend the rest of the module discussing the most popular linter, ESLint. We'll discuss the long list of configuration approaches to consider. And we'll set up linting to run all our rules every time that we hit Save. Simple enough. Let's get rolling.

### Why Lint?

So, why do you need a linter? I see two core reasons. First, a linter programmatically enforces consistency. Once you've chosen JavaScript coding standards as a team, a linter can help enforce those programmatically and provide rapid feedback so issues are caught during development instead of potentially slipping by during code reviews. Examples include enforcing the position of curly braces, or warning about the use of built-in features that your team has decided to avoid, like confirm and alert. Many teams prefer to use nicely styled dialogue boxes instead of the native implementations of these features. Linting helps programmatically restrict their usage. Or leaving in a trailing comma, or forgetting to add a trailing comma if your team prefers trailing commas as a standard. Or declaring a global variable, or disallowing the use of eval since it's potentially dangerous and often misused. All of these are examples of enforcing consistency through a linter. Second, a linter helps avoid mistakes. Just consider the long list of potential mistakes that you can make writing JavaScript today. Such as adding an extra parenthesis when wrapping a statement. Or overwriting a function when you don't realize that the function with the same name already exists. How about performing an assignment in a conditional

when you almost certainly meant to perform a comparison instead? It's really easy to leave out that extra equals sign. Or forgetting to define a default case in a switch statement, which can lead to hard-to-debug fall through issues. How about leaving debugging-related junk in your code, like debugger or console.log? It's easy to accidentally leave these in and have them slip into production. With a linter, you'll know immediately when you do, and you can even fail to build on your continuous integration server when a developer overlooks warnings and commits any of these issues. This list just scratches the surface. So assuming you're sold on using a linter, the next question is which linter should you use? Let's explore that in the next clip.

## Linters

Okay, now that you're hopefully sold on the benefits, it's time to pick a linter. JSLint was the original, created by Douglas Crockford many years ago. It's extremely opinionated, and while some may consider that a feature, the public has largely moved on to more powerful and configurable alternatives today. One such alternative is JSHint. JSHint is an improvement on JSLint which offers more configurability. But in recent years, the most popular linter by far has become ESLint. In fact, it's become so powerful and configurable that I'm not aware of any good reasons to choose JSHint or JSLint anymore. ESLint has become the de facto standard. So, bottom line, this choice is really quite easy. My suggestion is to just use ESLint. Now, actually, I can't quite say just use ESLint, because if you choose to use a language that compiles to JavaScript, then you might need to choose an alternative linter. For example, at the time of this writing, none of these linters support linting typescript. Although ESLint is expected to add support in the near future. If you're working in typescript, then for now you'll need to choose TSLint instead. And now that we've chosen ESLint, let's explore our different options for configuring it in the coming clips.

## ESLint Configuration Decisions Overview

Much like the rest of life in JavaScript land, ESLint is filled with decisions. Let's explore some of the key decisions that we'll make configuring it. Choosing ESLint was the easy part. The hard part is this next set of decisions that we need to make. We need to choose a config format. We need to decide which built-in rules that we'd like to enable. We need to decide whether to use warnings or errors. We need to consider extending ESLint's power by adding plugins for the framework of your choice and, finally, if all this just sounds overwhelming you can just punt on the whole thing and use a preset instead. Let's discuss each of these decisions so you're clear how to configure your ESLint best for your needs.

## Decision 1: Configuration File Format

Let's begin with decision one. Where should you put your configuration? The number of different ways to configure ESLint is just plain silly. There are five different file names that it currently supports for configuration or you can add your configuration to package.json so how do you decide? Well, the most

common universal approach is to create a dedicated .eslintrc file using one of the file names and formats mentioned on the previous slide but, assuming you're already using npm you can also configure ESLint in package.json. The advantage of configuring via a separate file is it's universal; it is not tied to npm. But, using a package.json avoids creating yet another file in your application. To configure ESLint via package.json add a section called eslintConfig. The contents of this section will be the same as the .eslintrc approach and we'll walk thought the contents of eslintrc in a moment.

Decision 2: Which Rules?

After choosing a configuration method, decision two is, which rule should we enable? ESLint catches dozens of potential errors out of the box. Comma dangle, no duplicate arguments, no extra semi-colons. I suggest gathering as a team and deciding once and for all which of these rules are worth enabling. Yes, it will be a painful meeting but once you get it done and in your starter kit, it's settled and you can enjoy the benefits.

Decision 3: Warnings or Errors?

Now that you've decided which rules you want to enable you have yet another decision to make. Which of your rules should merely produce warnings and which rules are a big enough deal to justify throwing errors instead? Let's consider the implications of warnings versus errors. With a warning, it doesn't break the build so you can continue with development without fixing the issue. In contrast, errors actually break the build which can be helpful when the linter finds a more critical issue that should catch your attention immediately and keep you from moving forward. But, since warnings don't stop development they can ultimately be ignored. This is handy in the moment when you're focused on implementing a feature and don't want to stop your flow to fix a minor issue But, it also means that warnings can potentially be committed because they may not break the build. Errors, in contrast, are a clear wall to moving forward. They can't be ignored. Due to these trade-offs, I've seen some shops only use warnings because they favor moving as fast as possible and I've seen other shops use only errors because they favor stopping any work that isn't good enough to commit at that moment. Now, I suggest using both. Warnings are good for minor stylistic issues and errors are useful for items that are likely to produce bugs. The bottom line is, it's important that your development team agrees that warnings are not acceptable. If you commit code that produces warnings then quickly, your linting output will get so noisy that it's useless and it will mask other helpful output like test results that also display on the same command line. If you choose errors then you don't have to worry about people ignoring linting issues. They will be forced to comply because the application won't build. In summary, I recommend choosing carefully based on context. You'll likely decide that only some rules weren't throwing an error versus a warning.

## Decision 4: Plugins?

Now that you've configured ESLint's built-in rules you have another decision to make. Should you enhance ESLint with some plugins for your library or framework of choice and if so, which ones? For instance, I primarily work in React and my react course is, you can see how I use ESLint-plugin-React to perform a number of additional checks that are specific to React and similar plugins are available for other popular frameworks like Angular as well as for node.js There's a handy list of ESLint configs, plugins, parsers and more, at this URL. As you'll see, there are plugins for many popular frameworks and libraries available. These plugins are useful because they help enforce a consistent style in the way that your team works with your framework of choice.

## Decision 5: Preset

Did those first four decisions feel pretty overwhelming? If so, then this option allows you to avoid the decisions that I just talked about. Instead, you can simply decide to use someone else's preset. There are multiple popular ways to handle declaring your ESLint rules. The most obvious option is to start from scratch like we just discussed. This way, you can take the full list of rules, work through it one by one, and build up your own list of settings that's perfect for the way your team wants to do development. But, if everything we just talked about sounds like too much work, there are certainly some good ways around it. ESLint comes with a preset that implements a logical set of defaults that can save you a lot of time. It decides what is a warning and error for you and enables the rules that it thinks make the most sense for most people. I prefer to use ESLint's standard rules as a good shortcut to get started, then I tend to tweak a few of the settings based on our team's feedback. This is a nice compromise because it avoids the work of starting from scratch, but it still offers complete power in tweaking the settings as your team sees fit. Or, you can even go a step farther and use an existing set of rules like airbnbs, XO, or standard JS. Assuming that you don't mind the decisions that they've made, this is a great way to avoid spending a lot of time arguing about all the decisions that I just discussed. It's also worth noting that standard JS isn't actually a standard. In fact, ironically, many of the rules that it enforces like disallowing semi-colons and only allowing single quotes for stings, are actually quite unpopular in the JavaScript community. All of these options use ESLint but they enforce strong opinions so you don't have to make any decisions configuring rules. In fact, with standard JS, you can't change any rules at all but assuming that you're willing to accept the strong opinions in these presets on the right these are all very low friction ways to get started and, now that we've talked about configuration of ESLint we also need to discuss a couple of issues that are likely to trip you up.

## Watching Files with ESLint

Now that we have decided that we are using ESLint there are a few different ways to actually run it. Of course, the simplest way to run ESLint is via the command line, however, there's one obvious limitation with this approach. ESLint doesn't currently include a watch setting built-in so if you want to

automatically run ESLint each time that you hit Save running ESLint by itself won't work. Here are two ways to get around ESLint's lack of file watching capability. Fist, since we're using webpack, one option is to use eslint-loader so webpack will run ESLint each time we run our build. The advantage of this approach is all files being bundled by webpack are re-linted every time that you hit Save. So, you see an ongoing summary of any linting issues. However, I recommend going a different route and using and npm package called eslint-watch. This npm package is simply a wrapper around ESLint that adds file watching capability. So, this stands alone and isn't tied to webpack in any way so you can use this approach to linting regardless of the bundler that you choose. eslint-watch also adds some other nice tweaks like better looking warnings and error messaging and it displays a message when linting comes back clean unlike eslint-loader, which is completely silent when there are no linting issues. But, finally, the biggest win with this approach is that you can easily lint all your files even if they're not being bundled as part of your app. So, this means you can lint your tests, webpack config, and any build scripts as well. I really like this so that I can ensure that all the code in my project is held to the same standard and has a consistent style.

## Linting Experimental Features

Another issue you may run into is that ESLint doesn't support many experimental JavaScript features at this time. Now, ESLint supports all of ES6 and ES7 natively and is expected to continue adding support for standardized features as JavaScript progresses. It also supports object spread even though that's currently and experimental feature. But what if you wanted to use other experimental JavaScript features? Well, then, you'll likely want to use babel-eslint instead, because it also lints stage zero through four features. You can via links to the different experimental JavaScript features under the plugins page on Babel's website. As you can see, JavaScript features progress through different stages. They're initially just ideas, which is called stage zero. Stage one is a formal proposal. Stage two is a draft spec. Stage three is a completed spec with initial browser implementations. So, as you can see, risk goes down as the stages progress. Bottom line, if you want to use these experimental features you'll want to use babel-eslint. For this course, I'm going to use only standardized JavaScript features, so we'll use plain ESLint and run it via eslint-watch but I wanted you to be aware of this option if you prefer to use experimental features and of course, if you choose to use experimental features you'll need to configure Babel with the appropriate plugins to transpile your code as well.

## Why Lint Via an Automated Build?

Now, as we've been talking through this process you might be wondering why we should be bothering to lint via an automated build process, because many editors offer ESLint integration built-in, so they just monitor your code and output results inline right there within the editor. However, I prefer to integrate ESLint with my build process for multiple reasons: First, outputting all feedback on my code to the command line gives me one single place to check for all the feedback related to my code quality. This means I have one place to check not just for linting issues but also for any compile time errors or

any testing errors. This is especially helpful on teams where developers all use different editors. We all have the same development work flow because we all utilize the same starter kit and a command line. Pair programming is also easier when everyone has the same development process, and most importantly, ESLint should be part of your build process so that the build is broken on your continuous integration server when someone commits any code that throws a linting error. This helps protect your application from slowly getting sloppy. Even if a developer ignores ESLint locally the build can be rejected automatically by your continuous integration server.

Demo: ESLint Set Up

Alright, let's jump back into the editor and configure ESLint so that we get rapid feedback as we code. We just went over a long list of decisions so here's the plan. We're going to use ESLint's built-in recommended rules so we don't have to waste time configuring individual rules and we'll use eslint-watch to add watch capability so that our files are linted and reported to the command line the moment that we hit Save. Let's make it happen. To help us quickly catch mistakes, maintain consistency and enforce best practices, we're going to use ESLint to lint our code. Every time that we hit Save it will run. To configure ESLint we can either place a .eslintrc file in the route of our project or we can configure it in package.json. I prefer the separate file so let's create a .eslintrc file in the root of our project. So I'll say new file .eslintrc.json Now, as a quick note, in my previous course on React and Redux in ES6 I used a .eslintrc file without an extension, but recently, ESLint has declared the file name without an extension to be deprecated, so I've added the .json extension that you see here. To get the rules that we're going to use I just recommend coming over to GitHub Gist at this shortened URL. Click on Raw and then copy the rules that we'd like to put in. Save you a little bit of typing. So let's walk through our .eslintrc First, you can see that I'm declaring this as the root ESLint file. By default, ESLint will look for configuration files in all parent folders up to the root directory. So, just in case you have an ESLint file in some parent directory I'm adding this for safety. This tells ESLint that this is the project root so it shouldn't look in any parent folders for other configuration files. So this will help ensure that you don't get odd behavior. This will be the only ESLint file applying to our project and as you can see, we're using ESLint's recommended rules. This enables many warnings and errors based on ESLint's recommendation and as we saw in the slides, we could, of course, use alternatives like airbnb's rules as a base line if preferred. We can also augment the recommended settings with plugins that provide enhanced linting. In our case, I'm going to add plugins for enhanced linting of ES6 imports. This assures that if we create an invalid import statement we'll find out about it the moment that we hit Save. Now, the parser option section defines the version of JavaScript that we're using. We're using ES7, also known as ES2016, since that's the latest standard at the time of this recording and I'm declaring that we're using standard JavaScript modules. The environment section declares some different environments that ESLint should be aware of. These environments tell ESLint to expect certain global variables. As you can see, we're expecting to work with the browser with node, and to run our tests with mocha. Now, if you view the full list of environments on ESLint you can see that many other frameworks are listed including QUnit, Jasmine, Jest and a variety of other environments. Finally, we can define any rules that we want to override down here at the bottom. Let's add a single rule down here just so you can see how to override ESLint's standard rules.

I'm going to set no-console to a warning. One means warning, two means error, and zero means off. So, if you feel strongly about a rule, you can set it to two and break the build. But since writing to the console can be useful during development, I prefer to set this to a warning so that it doesn't break the build. Alright, now that we have ESLint configured let's set it up to run via a npm script. So we'll come back over here to package.json Now, we could simply run ESLint directly in a npm script but ESLint lacks watch functionality so we use a handy npm package called eslint-watch. eslint-watch adds file watching functionality to ESLint and also offers enhanced command line output. So let's create a npm script that calls eslint-watch. And to do so, we'll call it Lint and we'll call eslint-watch and we want it to watch webpack.config.* since soon we'll be adding a production version of our webpack config. We want it to watch everything in source and also everything within our build script's folder. So this should effectively lint all of our JavaScript. So, to clarify, esw is the executable for eslint-watch and we're just passing eslint-watch the list of files that we'd like it to watch. And here's an important note that can trip you up. If your editor has linting built in be sure to disable it, otherwise your editor's built in linting may override the linting rules that we're going to define. So if you have any linting related plugins in your editor please disable them now. Let's go back to my eslint.json because I didn't save it and now we should be ready to run this on the command line. Say nmp run lint and we can see that we do get some errors back from source server and from start message. Since we configured ESLint to throw a warning for the no-console rule, rather than an error which is the default in the ESLint recommended rules and you can see that it's currently just displaying black text, but if we come back to package.json and add --color on the end of this command now we'll tell lint to display our issues with colors. Now we get nice yellow warnings here and some color here on our counts on the right. Now, this first linting error is from source server so let's go over to build scripts to source server and it's complaining about this line where we're doing a console.log One way that I can handle this is just to add a comment to the top of this file that disables this rule in this file since I'm OK with writing to the console in any build script file. I just want to make sure that I'm not writing to the console in my actual application. So, we can see that this fixed one of the linting issues. Now we can go to start message and we can see right here we're writing to the console. The other way that I could disable this is to put in a comment and say eslint-disable-line and then put in the name of the rule that I would like to disable. So, this can be useful when there are rare exceptions to the rule that we've defined. So, now, if I rerun it again, we should see that linting comes back clean, and it does. It reports back clean with this nice green arrow. This is another thing I like about eslint-watch is we get confirmation that it's clean. I like this little extra piece of output. Linting is now running as expected but it's only running once and then stopping so let's set it up to watch our files in the next clip.

Demo: Watching Files

Let's create another handy script. Oddly, eslint-watch doesn't watch our files by default. Instead, you have to pass it a command line flag to enable watch. So, let's create a separate npm script that will watch our files. We'll call it lint:watch and I'll place it right here below our lint script and in this we'll say npm run lint and here's where things get weird. I'm going to say -- --watch so we're passing the watch flag along up here to our lint script, so this is saying run the npm lint script but pass the watch flag to

eslint-watch. So let's hit Save and see if it works. npm run lint:watch So, now, if I come over to source server and take out this disable, hit Save, we can see ESLint reruns immediately and throws a warning about me using the console statement in this file right down here, and if I put this back in and hit Save then ESLint runs again and then reports everything back clean. Excellent, so, now we have linting watching our files and now that linting's set up, we'll know when we make many common mistakes in our code. The linting errors will display immediately in our console when we hit Save. Now, there's one final piece that's missing here. We'd like ESLint to run every time that we start our app so we just need to add our lint:watch task here to our start script, and it will run in parallel since we're already using npm-run-all and telling it to run any of the scripts that we list over here in parallel. Now, if I type npm start -s we should see that linting is part of our start now. We get our message, we get our security check and there our linting is running as well, So now when we type npm start it displays our start message, runs webpack, starts our development web server, opens the app in our default browser, lints our files and reruns webpack and ESLint any time that we hit Save. That's a lot of power in so little code. Let's wrap up this module with a quick summary.

## Summary

In this module we saw two core reasons to lint. Linting helps enforce consistency so that our code is easy to read and it helps us avoid many common mistakes related to typos, globals, and accidental assignments. We saw that there are multiple linters but we chose ESLint because it's currently the most popular, configurable and extensible linter available. We reviewed a variety of configuration choices including the config format, the rules that you enable, whether you use warnings versus errors, which plugins you should add and, if you're overwhelmed, you can just select a preset instead such as airbnb's configuration or the standard JS config and we wrapped up by enhancing our development environment to use ESLint's recommended rules and run ESLint every time we hit Save using eslint-watch. Our development environment is really coming together now but we haven't covered a critical aspect yet. What about testing and continuous integration? Let's explore these topics in the next module.

# Testing and Continuous Integration

## Intro

It's a shame how uncommon automated testing and continuous integration are in JavaScript and I believe it's because people don't see a clear picture of how to get quickly started. Since JavaScript has no built-in opinions on handling testing, you need to spend a lot of time browsing the web and investigating strategies before you can get rolling. So in this module, I'd like to help outline the landscape to help you understand the key decisions that you need to make. Because if you're new to automated testing, just picking your tools and deciding how to use them is a major hurdle. So we'll begin by reviewing six key decisions that you need to make, including testing frameworks, assertion libraries, helper libraries, and more. And once we've clarified our testing stack, we'll jump back into the editor

and set up our test environment and write our first example tests. We'll close up this module by discussing continuous integration services so that we're notified immediately anytime someone breaks the build and we'll set up two different continuous integration servers. This is a big topic with a lot of ground to cover, so let's get rolling.

## Test Decisions Overview

Comprehensively covering JavaScript testing would require multiple courses, so in this module, I'm going to focus on the style of testing that's most commonly configured in JavaScript development environments today, which is unit testing. Unit testing focuses on testing a single function or module in an automated fashion. Unit tests often assert that a certain function returns an expected value when past certain parameters. Unit tests mock out external dependencies like APIs, database calls, and file system interactions, so the results are fast and deterministic. But there are other types of useful, automated testing styles for JavaScript that I won't have time to cover in this module. Two other styles that are worth looking into that we won't cover in this module are integration testing, which focuses on testing the interactions between multiple modules and automated UI testing, which tests the application by automating clicks and key strokes within the actual UI and asserting that it interacts in expected ways. Tools like Selenium, which automate browser interactions are popular in this space. There are various other testing approaches as well, but in this module, I'll focus on automated unit testing. There are no less than six important decisions you need to consider when setting up automated unit testing in JavaScript. You have to choose a framework, an assertion library, helper libraries, you have to decide what environment that you want to run your tests in, you need to decide where to place your test files, and finally, decide when to run your tests. In the next six clips, let's run through each of these decisions.

## Decision 1: Testing Framework

The first decision we need to make is what testing framework to use. There are a wide variety of JavaScript testing frameworks available. Let's quickly consider the top six. Mocha is the most popular because it's highly configurable and has a large ecosystem of support. Jasmine is nearly as popular as Mocha and quite similar but Jasmine includes an assertion library built in. I find Mocha to be more configurable than Jasmine, so I personally prefer Mocha over Jasmine. Tape is the leanest and simplest of the bunch. Its simplicity and minimal configuration are its key strengths. QUnit is the oldest on this list and actually created for testing jQuery by jQuery's creator, John Resig. However today, other frameworks like Mocha and Jasmine are much more popular. And AVA is a new framework that offers some interesting features. AVA runs your tests in parallel and it only reruns impacted tests, both of which helps speed results. Finally, Jest is from Facebook. It has recently become quite popular for React developers, but since it's really just a nice wrapper over Jasmine, it's actually quite useful for anyone. Jest has code coverage, JSDOM, and popular conventions for finding your test files all built in. It's also recently gotten much better, so if you looked at Jest before and were turned off, it's time to look again.

Of course it's easy to feel overwhelmed with all these options, but I like to think of choosing a testing framework like choosing a gym. Sure, some gyms are nicer than others, but you can greatly improve your health at any gym. So the important part is picking a gym, any gym, so that you can start exercising. In the same way, the right answer here is to quickly review this list and pick one. There really isn't a loser here, so don't worry much that you've picked the wrong framework. Just like gyms, it's easy to switch to a different one later. In fact, switching frameworks often involves merely trivial syntax changes. The only clear wrong choice is to be this guy, coding and praying, and don't laugh, because we all do it. I've spoken to countless developers on this topic and I believe that decision overload is largely what's holding people back, so don't be this guy, pick one of these six on the left and get moving. For this course, I'm going to use Mocha because it's popular, mature, flexible, and boasts a large ecosystem of support.

Decision 2: Assertion Libraries

Many test frameworks such as Jasmine and Jest come with assertions built in. But some, such as Mocha, don't come with an assertion library, so we have to pick our own. So what's an assertion? An assertion is a way to declare what you expect. For example, here I'm asserting that two plus two should equal four. This is an assertion because I'm telling my test what I expect to happen. If the statement is false, then the test fails. There are many potential ways to declare assertions and test. Some frameworks might look like this example, others might use a key word like assert instead of expect. Don't let the minor differences confuse you. Most of the choice between assertion libraries come down to minor syntactic differences. The most popular assertion library is Chai, but there are other assertion libraries out there to consider like Should.js and Expect. Most frameworks include their own assertions built in, but since Mocha doesn't, we need to choose one. Again, the core difference between these are relatively minor syntax differences, so don't spend too much time worrying about this. In this course, we'll use Chai for assertions because it's popular and offers an array of assertion styles to choose from.

Decision 3: Helper Libraries

There's another question to answer before we start writing tests. Should we use a helper library? And if so, which one? JSDOM is one interesting library to consider. JSDOM is an implementation of the browser's DOM that you can run in Node.js. So with JSDOM, we can run tests that rely on the DOM without opening an actual browser. This keeps your testing configuration for automated tests simpler and often means that tests run faster because they're not reliant on running in the browser. So JSDOM is useful when you want to write tests that involve HTML and interactions in the browser using Node. We'll write a test using JSDOM later in this module. Cheerio is another interesting library worth mentioning. You can think of Cheerio as jQuery for the server. This is really handy if you're using JSDOM because you can write tests that assert that certain HTML is where you expect it. And the great news is, if you understand jQuery, you already know how to work with Cheerio, because it uses jQuery selectors for querying the DOM. Imagine you wrote a test that expects a specific DOM element to exist on the

page. With Cheerio, you can query JSDOM's virtual DOM using jQuery's selectors. If you already know jQuery, this can save some typing compared to writing traditional DOM queries.

## Decision 4: Where To Run Tests

Feeling overwhelmed with options yet? Hopefully not, because we're in JavaScript land, so we also need to decide where to run our tests. There are three popular approaches to running JavaScript-based tests. The most obvious option is running our tests in the browser. Karma and Testem are popular test runners for testing in an actual browser. However, opening an actual browser requires more configuration and is slower than the alternatives, so I prefer to avoid this approach. Instead, we can utilize a headless browser like PhantomJS to run our tests. So what's a headless browser? A headless browser is a browser that doesn't have a visible user interface. PhantomJS is a full real browser running the V8 JavaScript engine behind the scenes. But you can't see PhantomJS because it has no visible interface. This is useful because often, writing automated tests you don't need to see the actual interface. You just need something fast that simulates a real browser. I've used this approach successfully in the past as well. The third option is to utilize an in-memory DOM. As we just discussed, JSDOM is a library that simulates an actual browser by creating a DOM in memory that we can interact with. You can think of JSDOM as a lighter-weight alternative to PhantomJS because JSDOM doesn't have a full browser behind the scenes. It's just focused on simulating a DOM in memory. The advantage to this approach is it's fast and quick to set up. That said, both PhantomJS and JSDOM are great options to consider. We're going to write our tests using JSDOM in Node in an upcoming clip.

## Decision 5: Where Do Test Files Belong?

Decision five is, where should I put all my tests? There are two popular schools of thought on organizing your test files. Let's review the merits of each approach. One popular approach is to centralize all your tests within a folder called tests or something similar. So all your tests are completely separate from your source code. Mocha pushes you in this direction because it defaults to looking for tests and the root of your project in a folder called test. Now the primary benefit that I hear people claim about this approach is that it avoids adding noise to your source code directory. But I find this mindset misguided. Tests aren't noise, they're complementary to the files that they're testing. They're important. To me, if they're worth writing, they're worth seeing regularly within my source instead of tucked away in a separate folder. Another common reason that I hear is, I don't want my test files deployed to production. As you'll see in the production build module at the end of this course, this concern is unmerited. We'll only deploy the final bundled HTML, JavaScript, and CSS for the app to the actual production server. Ultimately, I think much of the reason people are using a separate test folder is simply inertia. It's popular to create a separate test folder project in many ServerSide technologies for other reasons, but the separation just doesn't make sense on JavaScript apps. So I prefer to place my tests alongside the file under test. Here's why. I find it makes importing easier because the paths are trivial to work with. Since the test and the file under test are in the same path, imports are clean. It's

always ./file under test. Sure beats managing a lot of dot dot slashes to reference some source code folder that's in a totally different spot. Second, it provides clear visibility to our tests. They're not buried in a separate folder. They're right there in our source. So it's quite easy to notice file that lacks a corresponding test file. Again, to me, test file visibility is an asset, not a liability. Third, placing them together makes it easy to open them both at the same time. If you open a file to write code, you should probably be writing some tests at the same time. So co-locating files that you work on at the same time just makes sense. Co-location also avoids having to maintain two separate directory structures. When you separate tests from your source, you often end up having to create new folders with the same name in two different places. I'm not a fan of repeating myself, or of saying the same thing twice. Okay, bad pun, moving on. Finally, it's more convenient when we refactor and move files as well. When tests are centralized, moving the file under tests requires updating the path in a corresponding test file. When tests are placed alongside the file under test, it's easy to simply drag both files to their new location without making a path change. The relative paths remain the same. This is a minor side note, but I was also curious about the naming conventions people are using for naming JavaScript files. As you can see, naming test files with a suffix of .spec and .test are both very popular conventions.

Decision 6: When Should Tests Run?

Okay, one final decision to make regarding testing. When should our tests run? Well, if we're talking about unit tests, the answer is simple. Unit tests should run every time that you hit save. This rapid feedback loop assures that you're notified immediately of any regressions. And running tests each time you hit save facilitates test-driven development, since you can quickly see your tests go from red to green just by hitting control S. If you run your tests manually, it creates unnecessary friction. When the test suite is run manually, it's easy to forget to run the test suite after making a change, so make it automatic and reduce friction. Finally, running tests on save increases the visibility of the tests that do exist. It helps to keep testing in the forefront of your mind, so I believe this is an easy decision. Your unit tests should run automatically when you hit save. I know what you're thinking, but I can't run my test suite every time I hit save, that'll be way too slow. Well, I should emphasize. I'm talking about unit tests here. Unit tests should run extremely fast. Integration tests are also useful and admittedly slower, so you'll want to run those separately. But your unit tests should be fast because they shouldn't hit external resources. Now let me back up for a moment and clarify the difference between unit tests and integration tests. Unit testing is about testing a single small unit of code in isolation. Integration testing is about testing the integration of multiple items. So unit testing often involves testing a single function all by itself while integration testing often means firing up a browser and clicking on the real UI using an automation tool like Selenium and often making actual calls to a web API, though you can of course write integration tests using just Node and JSDOM, for instance. And since unit tests seek to test a small portion of code in isolation, they run extremely quickly, quick enough that you should be able to run all your unit tests every time that you hit save. In contrast, integration tests are slower because they often require real external resources like browsers, web APIs, and databases, which take much longer to spin up and respond than native function calls. Now since unit tests run fast, they should be run every time you hit save, and if your unit tests don't run fast enough to rerun every time that you hit save, that's

often a sign that they're not really unit tests. But since integration tests typically interact with slow external resources, they're often run on demand or in QA. In summary, the answer to when your tests should run comes down to whether you're writing unit tests or integration tests. We're going to write unit tests in this module, so we'll run our tests every time that we hit save.

Demo: Testing Setup

Okay, so we've walked through the six big decisions you need to make for JavaScript testing. Now let's summarize the decisions I've settled on for this course. I'm going to use Mocha for the testing framework, Chai for the assertion library, JSDOM as a helper library so that we can utilize an in-memory DOM in our tests. We'll run our tests via Node, and we'll place our tests within our source code folder instead of storing them separately. And finally, we'll run our tests every time that we hit save. And now that we have a plan of attack, let's jump into the editor and set up testing. As you just saw, one of the hardest parts of JavaScript testing is just choosing an approach. But now that we have a planned path, let's set things up. We'll begin by creating a file that will configure our tests. We'll put it under buildScripts and call it testSetup.js. And I'll paste in the two lines of code with some comments. So this file does two things. First, we are requiring babel-register, so this will tell Mocha that first babel should transpile our tests before Mocha runs those tests. And then second, we're going to disable any webpack-specific features that Mocha doesn't understand, in this case the CSS extension, because remember, in our index.js, we are requiring index.css, this is a feature that webpack understands but Mocha does not, so we're just telling Mocha that if it sees this, just treat it like an empty function. Now, there are other things that we can do here like set up a JSDOM environment, but instead of doing that here, I'll show how to handle it in an example test. And now that we have our initial test setup script configured, let's jump over to package.json and add a script that will run our tests via Mocha. And again, I'll just paste it in here and then we'll talk through how this works. First, we'll specify the reporter we want to use. The reporter setting determines how the test output should display. I prefer to use the progress reporter because it's clean and simple, but Mocha offers a variety of interesting reporters to choose from. However, I recommend sticking with progress for our setup because many of the other reporters write so much information to the terminal that it can make it hard to see the lenting issues that are also being reported in that same terminal. Next we tell Mocha to run the test setup script that we just set up, and then after it's finished running that, it should run any tests that it finds within our source directory and any subdirectories. And we define test files as any file that ends in .test.js. And now that we've set this up, we should be able to open the terminal and type npm test to run Mocha. Now when we do, we see it fail, and it fails saying it cannot resolve path or pattern. This is Mocha's way of saying that it cannot find any test files, and that's not surprising because we haven't written any tests yet. So let's create one test. We'll assume that we're writing tests for our index file so I'll create a file in the same path that's called index.test.js. So we're following the convention of naming tests after the file under test, but with test.js on the end. Some prefer spec.js. Whatever you like is fine. Now Mocha doesn't come with an assertion library, so we're going to use Chai, and more specifically, we'll use the expect style that comes with Chai, so I will use a named import so that we have a reference to expect. And now we can describe our first test. And we'll provide it a function. You can of course use the

function keyword if you prefer, I'm just using an arrow function here for brevity. And inside, let's add our first test. We will say that it should pass. And again, I'll provide an arrow function here. And inside we can put the body of our test. I'll just put in expect true to equal true. So now we have our first test and that means if we come back down here and run npm test again, we should see it pass. And we do see it passing right here. And this is the output of the progress reporter. And we should also see that if I set this to false, that our test fails. And sure enough it does, we get a useful error message that shows the line that it failed, it was line five where our assertion happened. We thought we would get true, but we got false instead. So I will undo my change there. So we have our first test passing. In the next clip, let's create a test for something in the DOM using JSDOM.

Demo: DOM Testing

Let's add a second test that puts JSDOM to use. To do that, let's first import JSDOM and we'll also need to import fs, which comes along with Node, stands for file system. It lets us interact with the file system using Node. And now we're ready to write our test using JSDOM. So let's describe this one as index.html because that is going to be the file that we're wanting to test in this case. And inside we're going to say that it should say hello. Remember that we have a hello world sitting inside of here, so we're just going to write a test that confirms that markup is there. So first, let's get a reference to our index.html file and hold it in memory. To do that I'm going to say const index equals fs.readFileSync and then put in a reference to our index.html file. And we will also specify that it is in utf-8. So now we have the contents of our index.html file held in memory within a constant called index. So we're ready to now use JSDOM. So let's say jsdom.env, and this is our way of defining the JSDOM environment. And we will pass it, our index.html file, 'cause this constant represents the content of index.html. If you want JavaScript to run as part of your JSDOM environment, you can pass an array of JavaScript files as the second parameter here. But note that any of those files utilize fetch, you need to use isomorphic-fetch instead because fetch is a browser feature, so it won't be available by default in the Node environment. We don't need any JavaScript for this particular test, so I'll just omit the second parameter. The second parameter for this is a callback function, which is run after JSDOM is completed, pulling index.html into memory making a virtual DOM in memory. And it takes two parameters, an err and a window argument. And I will close this with a semicolon. The window here represents the window in the browser, just like you could say window when you're in the browser, now you can do so right here in Node because we have a virtual DOM in memory. So we're going to write a test that confirms this text is there. To do so, we want to get a reference to that h1. So let's define a constant called h1 and then say window.document.getElementsByTagName and the tag name that we're looking for is h1. Now this returns an array-like object so I'm just going to say give me the first h1 on the page because we know that our h1 is the first one on the page. So we now have a reference to the h1 on the page, we're ready to write our assertion. So we'll say we expect the innerHTML of h1 to equal its value, which is hello world with an exclamation. Finally, we'll go ahead and close the window just to free up the memory that was taken when we created our in-memory DOM. So we'll hit save and now we should be ready to come down here and say npm test. Or if you want to save some typing, npm t does the same thing. And you can see that we have two tests passing. Now just to prove that this is working, let's go ahead and change

our exclamation point to a question mark, hit save, and then rerun our test. We should see it fail. And we don't. Interesting. And you're probably surprised like I am that that's still passing. But there's a good reason for this. When we called JSDOM, there's an asynchronous call that occurs here. We have to set up our test to be asynchronous. To do that, when we call it, our function that we define here takes a parameter and we'll call this done. What we need to do is tell Mocha that our test is done, and then it will run the expect and report our results after it sees done here. If we do this and rerun our test, we should see it fail. And indeed now we do. So the world makes sense again. You can see it was expecting it to be an exclamation point but what it received was a question mark. So just remember when you're doing an asynchronous test, one that involves having a call back here, you need to add this done parameter so Mocha knows that it's now safe to evaluate whether your expect is true or false. So we can change this back, hit save, rerun, and we can see it's passing. Of course, I'm just scratching the surface here, but now we have a pattern for testing real DOM interactions without having to fire up a browser. Now that we have a couple of tests written, it would be nice if our tests ran every time that we hit save, so let's set that up in the next clip.

## Demo: Watching Tests

Another detail is missing in our tests. We shouldn't have to run our tests manually. So let's run them every time we hit save. To do that, we'll add a script down here called test watch and you should see this looks almost identical to lent watch, we're using the same pattern of telling the test script to run but we're passing another parameter to it with this dash dash space dash dash watch syntax. So it's just as though I'd taken this flag and added it up here. Now of course we also want to call this as part of our start script. So now let's run the app and see how it works. Say npm start minus S. And we can see the app starts up just fine. Of course there's much more to testing than this. You could set up mocking, code coverage, reporting, and more. Now that we have testing set up, it would be nice if we could fail the build if someone commits broken tests. So in the next clip, let's begin by exploring continuous integration.

## Why Continuous Integration?

While we're talking about assuring quality with testing there's another important practice to consider, continuous integration. When your team commits code, it's handy to confirm immediately that the commit works as expected when on another machine. That's what a continuous integration server is for, or CI server for short. So let's wrap up this module by setting up a continuous integration server to assure that we're notified when someone breaks the build. I'm sure you've heard that annoying guy Jimmy say this to you multiple times. Weird, it works on my machine! Well thanks Jimmy, that was super helpful. Wouldn't it be nice to find out right away when someone else has broken the build or when you made a bad commit that has broken the build and ruined someone else's day? That's what a continuous integration server is for. Now the question that you might be asking is how do we end up in a situation where it works on our machine but it breaks on the CI server? Well the CI server catches a number of

potential mistakes. Have you ever forgotten to commit a new dependency? Have you ever installed an npm package but forgotten to save the package reference to package.json? Or maybe you added a new step to the build script but it doesn't run cross-platform. Perhaps the version of Node that you're running locally is different from the one you're using in production, so the app may build just fine with the version on your local machine but fail on the continuous integration server. Maybe someone just completed a merge but made a mistake along the way that broke the build. Finally, perhaps someone on your team committed a change without running the test suite. In this case, I typically recommend covering their entire desk in aluminum foil, but the good news is, with the CI server, you don't have to worry. It will catch the culprit and notify the developer of his or her transgression. These are just a few great reasons to run a continuous integration server. The point is, a CI server catches mistakes quickly.

## What Does Continuous Integration Do?

So what does a CI server do to provide all these benefits? Well first it builds your application automatically the moment that you commit. This assures that your application builds on another machine. Sure beats the all too common alternative where hours or days later, someone gets latest and complains that someone broke the build. A CI server makes it clear who broke the build by checking every commit. It also runs your test suite. Of course, you should be running your tests before committing but a CI server assures it always happens and it assures that the tests pass on multiple machines. If the tests don't pass, then your commit has issues. So it's important to have a separate server run your tests to confirm they pass on more than just your machine. A CI server can run tasks like code coverage and reject a commit if code coverage is below a specified threshold. And finally, although it's not required, you can even consider automating deployment using a CI server. With this scenario, if all these aforementioned checks pass your application is automatically deployed to the production.

## Choosing a CI Server

There are multiple continuous integration servers to consider that work great for JavaScript apps. Travis CI is a Linux-based continuous integration server. Appveyor is s a Windows-based continuous integration server. Jenkins is another popular and highly-configurable option. There's also CircleCI, Semaphore, and SnapCI, which are interesting players to consider. Performance features and configurability of course differ between these options. Travis and Jenkins are the most popular and thus have the largest ecosystem of support, but Travis has a hosted solution while Jenkins is a good choice if you prefer to host your CI server on your own. Appveyor is notable because of its Windows support. In this course, we'll set up two continuous integration servers, Travis CI and Appveyor. Why two? Because Travis CI runs on Linux and Appveyor runs on Windows. This means we can be assured that our build process runs on Mac, Linux, and Windows. On my current team, developers run both Mac and Windows, so using both Travis CI and Appveyor helps assure that our build runs on both platforms.

Demo: Travis CI

Let's get back in the code and set up continuous integration using both Travis CI and Appveyor. Travis CI is a popular continuous integration server. It offers handy integration with GitHub, which makes it quick and easy to add to your project. So assuming that you're using GitHub for your source control it's quite straightforward to integrate Travis CI as your continuous integration server on your JavaScript app. Now, as you can see, I've been using GitHub for this demo. I showed how to install Git and set up a repository in the intro module of this course. Now you can sign in to Travis CI using your GitHub account and when you do you should see an empty list of repositories over here, obviously you can see I have some existing repositories. You can click this plus sign to add a new repository. Now when you do, you'll see a list of all your repositories and I have to scroll down a bit because I have a lot of them. The one that I want to turn on is js-dev-env-demo. To turn it on, I just click the X and that enables Travis CI. I can click this gear icon to see some settings, but we don't need to change anything in here, we do want it to run on build pushes and for build pull requests. The defaults are just fine. And now that we've set this up on the website we can finish configuring Travis by going back into the editor and creating a configuration file for Travis. Travis is configured via .travis.yml also known as .travis.yml. Inside, we're going to declare just two things. We're going to declare that the language that we're working with is node_js and then we can add a list of versions. I'm just going to check version six, but I could have add other lines here to have it check version five, version four, and so on. And that's all it takes to configure Travis for our continuous integration server when we're working with GitHub. Now before we commit our changes to fire off our first continuous integration build, let's open up index.html because I'm going to change line seven to have a question mark instead of an exclamation point. This should break our JSDOM-based test. In this way, we can see what it looks like when a build fails on our CI server. And now we're ready to commit our changes, which should fire off our first continuous integration build on Travis CI. Now there's Git integration built into VS Code and many popular editors, but I'm going to use Git via the command line because it will work the same for everyone. If I do a git status right now, I can see that I have all my changes for this module that are listed. I'm just going to add all of these as stage changes and the minute that I do you can see that they are now listed as stage changes here in my editor. So now they're ready to commit, and I will say git commit minus M module, I'm going to call this module nine work in progress, because it's not done yet. And I'll hit enter, when I do this commits. Now this just committed locally. If we actually want to push our changes up to GitHub, we need to do a git push. So now I did a git push and we can see that it wrote and pushed my changes up to GitHub.com, and we can prove that that's true by coming back over here and refreshing the page, seeing that there is now a new commit, module nine work in progress. Now the other thing that's interesting is, we should be able to come over here to Travis CI and see it working. We can see that it is running, so it was watching for our commit. If I click on this, we can see the details of the status in progress. We can see that it's installed the latest version of Node six and is displaying the version. Once this is complete, we'll be able to see the results, so I'll just pause for a moment until this is done. Okay, it finished, and oh, what do you know? The build failed. And this is a great example of why a CI server is useful. It ran our tests, it installed Node, it ran the tests, and we can see that it failed. Remember that I deliberately changed index.html to make the CI build fail. And if I say npm t here to run our tests, we should see it fail right here as well. So now I can go back here, open my index.html file, change this back, hit save, now when I run it here, it

succeeds. And it should also succeed in Travis CI once I commit my change. So now if I do a git add, we should see, if we do a git status now that the one file that I just added has been staged so I can do a git commit, fix broken test, will be my message, it's what the dash M stands for is my commit message. Now I will do a git push, this will push my change up to the server, so we should see if we come back over here that Travis CI has kicked off again, so we'll see whether I have fixed my test. Great, and now we can see that it came back green. It installed Node, it ran my tests, so everything looks good. Now we know that our application runs not just on my machine, but on a separate continuous integration server. This gives us more confidence that the application is ready for production. So now we know that our app builds and our tests pass within a Linux environment, but what if you're working in a Windows environment? For that, let's check out an alternative CI server called Appveyor.

Demo: Appveyor

We just saw how to set up continuous integration on a Linux server using Travis CI. Now it's time for Windows, so we're going to use Appveyor as an alternative to Travis CI that runs on Windows. Now, as you can see, here on Appveyor.com, you can click to sign up for free, and once you do you can sign in using your existing GitHub account, and you will need to authorize it, so I'll click authorize application and actually I can't do that because I have an existing account so I'll go click sign in instead and log in. Once I do, we can see some different projects that have run fairly recently. Of course for you with a new account, you won't see any existing projects. I'm going to click new project and as you can see Appveyor supports a number of different online repositories, but we're going to look at GitHub. Now I can see a list of all my GitHub reposts. I'll come down here. This should look pretty familiar compared to Travis CI. Here's what I'm looking for and I'm just going to click add. So now we're redirected to a page where you can view the build history, deployments, and settings. Let's click on settings. Here you can change a long list of settings, but again, the defaults here are just fine for our purposes. And just like Travis CI, we need to jump back into the editor to finish configuring Appveyor. Now Appveyor is configured with a file called appveyor.yml and it should again reside out in the root of your project. The recommended Appveyor configuration is a little more involved, so I'll just paste this in and we can talk through it. As you can see, we're telling Appveyor that we should be using node.js version six. We could add other versions here below with another dash if desired, and the rest of this boilerplate is recommended by Appveyor so that we can declare that we want to install our npm packages and also run our tests. We're telling it the specific npm tasks that it should run. And this output is just here because it's useful to see the Node and npm version that are being run when we're trying to debug. And with this file saved, we should be able to open the terminal and say git add dot, so we'll add this file to staging. We say git status, we should see that this is now staged for us, so let's commit this file. We'll say git commit minus M add Appveyor CI, hit enter. Now that's committed locally, let's push that up to GitHub by saying git push. When we do, let's go back over here and click on latest build. So we can already see that our build is in progress. Great, and it looks like our build succeeded. Let me scroll back to the top. We can see the green bar, so it all worked. Just like with Travis CI, if our build had failed, then we'd have received an email notifying us that we had broken the build. And just like Travis CI, we can see that it installed Node, installed our dependencies, and ran our tests successfully. So great, we can now feel confident that our

development environment runs properly on both Linux and Windows. Alright, that's it for this module, let's summarize what we just learned.

## Summary

We just saw the long list of decisions that you have to make to handle testing in JavaScript. You have to choose testing framework, but remember this is like choosing a gym. What's important is that you just pick one and start exercising. Then we quickly reviewed assertion libraries. Some frameworks like Jasmine come with assertions built in. Others like Mocha don't include an assertion library, so we used Chai. And you might want to use some helper libraries to get things done like JSDOM which provides an in-memory DOM and Cheerio, which provides a jQuery-like interface for querying the DOM. We discussed options for where to run your tests including using the actual browser, a headless browser, or doing in-memory testing. I showed the in-memory approach using JSDOM. We also reviewed the merits of placing your tests alongside the file under test, because it speeds navigation, increases visibility, and avoids having to recreate and maintain a separate file structure for your tests. And remember that unit tests should be run every time you hit save. If it's too slow for that, then it's likely an integration test, which is also useful, but should be handled separately. And we wrapped up this module by reviewing a few popular options for continuous integration. We set up both Travis CI and Appveyor, so we know that our app builds on both Linux and Windows. Great, so now we have testing and continuous integration configured, so we know we're ready to build a quality JavaScript application. Our development environment is working great, but some critical pieces are still missing. How do we make HTTP requests? And how do we generate mock data for rapid development when we don't have a completed production API? Nearly every JavaScript application needs to handle these issues, so let's explore this topic in the next module.

# HTTP Calls

## Intro

As we saw in the first module, JavaScript is eating the world, and when it comes to protocols, HTTP is eating the world too, virtually every JavaScript application that we've built today makes HTTP calls. So in this module, let's explore the libraries for making HTTP calls via JavaScript. These libraries are essential for making Ajax calls in modern applications. Then we'll shift our focus to Mocking HTTP Calls. We'll discuss why mocking is useful so we can code without hitting any actual APIs, rapidly try different response shapes, code offline and much more. And we'll wrap up by considering various approaches for doing mocking HTTP calls. We'll implement a compelling mock API that generates realistic fake data and simulates a database using a simple generated file full of JSON. This module is about making our development environment interact with the real world, and these days, the world talks HTTP, so let's dive in.

HTTP Call Approaches

There are at least half a dozen popular ways to handle HTTP calls in JavaScript. The library options depend on where you're running your app. Node provides a built in package called http, it's a low level library that provides basic functionality for making HTTP requests. Now request is a popular higher level library that makes it simpler to make these calls in Node. It provides a streamlined API that many prefer. Now if you're running JavaScript for the browser, you have a different set of options. You can of course use plain old XML http requests, also known as XHR for short, this is the native and original way to get the job done, and it's hard to believe, but the birth of XMLHttpRequest was over 17 years ago in 1999 and it's been broadly supported in browsers for well over a decade. Plain XMLHttpRequest looks like this, you have to manually check the ready state, use a verbose API to set the request header and attach to the on readyState change and error events to get the job done. As you can see it's a lot of plumbing. It looks pretty old and creaky these days, so most people prefer alternatives that offer a cleaner API. So for a long while people have often reached for jQuery to get this job done. jQuery's $.ajax object has been the workhorse of the web for many years, and if you're already using jQuery in your project it remains a pretty logical way to handle HTTP calls because it helps you avoid pulling in extra dependencies. And these days some more full featured frameworks like Angular include their own HTTP service, so you don't have to make this decision at all. But assuming your framework doesn't natively handle HTTP, another increasingly popular option is Fetch, which is a standard proposed by the Web Hypertext Application Technology Working Group, yes that's a mouthful, that's why people typically call them the What Working Group for short. Now, Fetch offers a streamlined API that elegantly handles HTTP calls, however some browsers lack native support, so you'll want to use a polyfill with this option. You can find polyfills for both the regular version of Fetch or the isomorphic version of Fetch which we're going to talk about in a moment, but it's also worth noting that Fetch is currently a streamlined API, so it doesn't offer all the features of raw XML HTTP requests, or the other libraries that extract XML HTTP requests away that we'll talk about in a moment. So for instance, you can't cancel a Fetch at this time, but this limitation is being actively worked, so although Fetch support is currently being added to popular browsers, it's feature set is expected to grow over time. Here's an example of using Fetch, as you can see, you create a request object and pass that to Fetch. And since Fetch uses process to handle results, you provide a success and error handler to the then function. Full featured libraries like Axios and SuperAgent that we'll talk about in a moment are great, but the new native Fetch API is likely to provide all the power that you need. So finally, some packages work on both the client and the server, isomorphic-fetch is an NPM package that provides a Fetch-like API that runs on a server via Node and in the browser, that's why it's called isomorphic-fetch, more recently the term universal JavaScript has become popular to describe JavaScript that runs on both the client and the server. You can also choose to use XHR which is a package available on NPM, XHR provides a subset of the request library that we talked about over under the Node column, but the subset of features that it supports run on both Node and the browser. Now if you're looking for full featured options, SuperAgent and Axios are popular libraries that run on both NodeJS and the browser, both are elegant and popular, but I personally prefer Axios because it offers a clean promise based API. Here's an example of a call using Axios. This code is

simple, easy to read and nicely declarative. I enjoy the promised based API. James K. Nelson describes Axios as the XMLHttpRequest library which takes all the good parts from Angular's $http and throws out everything else, but SuperAgent is quite popular as well, it even has it's own plug in ecosystem. Now, that's a lot of options, so let's sum this up. If you're working only on Node, you'll probably want to use requests unless you have a good reason to avoid taking on another dependency. If you're in the browser, then Fetch with a polyfill is the most future proof approach since you won't need a polyfill once all browsers finish adding support. However this also assumes that you can live with the limitations of Fetch. Finally if you're building an app that needs to render on both the client and the server, then any of the libraries on the far right are a great choice. Choosing between them largely comes down to whether you prioritize file size or features, SuperAgent and Axios weigh a bit more, but also offer more features. And now that you hopefully have a good feel for what library is the best fit for your needs, in the next clip, let's discuss why it's important to centralize API calls.

## Centralizing HTTP Requests

Here's an important key I see people often overlook when making API calls. Make sure they're handled in a single spot. So why is this important? Because it centralizes key concerns. First, it gives you one place to configure all your calls, this way you can configure important configuration like base URLs, preferred response type and whether to pass credentials in a single spot. You make sure all git put post and delete calls are handled consistently, when asynchronous calls are in progress, it's important that the user is aware. This is often accomplished via a moving preloader icon, commonly called a spinner. By centralizing all your calls, you can keep track of how many asynchronous calls are in progress, this assures a preloader continues to display until all async calls are complete, centralization also gives you a single place to handle all errors, this ensures that any time an error occurs, your application can handle it in a standardized way, perhaps you want to display an error dialog, or log the error via a separate HTTP request. By centralizing your API calls, a single method can assure that this occurs for all calls. Finally, centralizing your API calls gives you a single seam for mocking your API. Centralizing your calls means you can point to a mock API instead of a real one by changing a single line of code that points to a different base URL. We'll discuss this more in an upcoming clip.

## Demo: Fetch

Okay, enough talk, let's make some decisions. For this course, we're just building a web app, so let's set up Fetch along with the associated polyfill from GitHub so that we can assure that our code runs cross-browser, and while we're doing so, let's centralize our API calls for all the reasons that I outlined in the previous slide. As I mentioned in the slides, I prefer to centralize my API calls, centralization assures that I handle all API calls in a consistent way, it creates a clear seam for mocking and it makes displaying a preloader while asynchronous calls are in progress trivial. But we have a problem, our app doesn't currently have an API so let's use Express to create a single API call. For simplicity let's just serve the API using the same express instance that's serving our app during development. So let's open srcServer.js

and add a new route. Let's create a simple end point that returns user data. As you can see when we hit slash users, it should return a hard coded array of a few records. Of course, in a real app, this would hit a database and perhaps be served by a different machine on a different web server, but, I'm just going to hard code some data here instead of setting up a real database or separate API. And now that we have this call, we should be able to jump over to the browser and confirm this works, so let's say NPM start, and if I put in /users, we can see the JSON getting returned from express as we expected. Now let's update our app to call the API using Fetch and display the results on the page. Again, I like to keep all my API calls centralized, so let's create a folder called API under the src folder. And inside, let's create a file called userApi. So this is where our API call to get users will reside. And I'll paste this in, and we can see that at the top I'm importing what working group fetch, so this polyfill will ensure that this code runs in browsers that don't yet have Fetch support natively, and as you can see I'm only exporting one public function, get users, all the other functions below are private. The actual call that's using Fetch occurs here in the get function. Over this small amount of boiler plate setup, adding other get requests will require very little code because we only need to provide the URL. Fetch, along with promise resolution and error handling are an abstracted away behind this private get function. Right now I'm only supporting get but you might want to add functions for handling put, post and delete requests as well we'll add support for a delete requests in a following clip. You can think of this file a bit like the repository pattern but in JavaScript, on the server the repository pattern is often used to abstract away data access using a course grained API, and that's basically what we're doing here too except instead of abstracting away a database, we're abstracting away our web API from our application. And the beauty of centralizing our calls here is we have one place to consistently handle all of our Ajax calls. If one fails, we have centralized error handling and if we wanted to show a preloader, we have one spot to keep track of any calls in progress. And, as you'll see in a moment, if our base URL changes in different environments, we have a single place to configure. Okay, now we have our API set up to use Fetch, let's call it to do so, let's jump over to index.html. Let's create a table inside index.html, I'm going to remove the h1 that says Hello World and instead paste in a simple structure for a table. You can see that we have headers for ID, first name, last name and email, because this is the data structure that we're expecting to receive from our API call. And the table body tag deliberately has an ID here of users because I'm going to reference this in some JavaScript that we write next. Now let's jump over to index.js and write some code to populate this table with the results from our API call. First, let's add a reference to the API call that we need, which is getUsers, and then let's remove this code because we no longer need it, instead, our application is now just going to display user data. And here's some code to populate a table of users using our API call, you can see that on line 6, I make the call to getUsers and I use the then function on the promise to handle the result that we receive from our API call. I then loop through the list of users returned and returned a string of HTML which I then place within the inner HTML of that users table body which we created in index.html. So this code will end up populating our HTML table. Of course in a real application I'd likely do this using React or Angular, but I wanted to use plain vanilla JavaScript here to avoid adding the complexity of a framework since that's not the focus of this course. And now when I hit Save we can see that our test fails, and our test fails because remember we wrote a rather silly test that was making sure that our Hello World message was within the h1. So let's go ahead and change this to, say, should have h1 that says Users, and then we'll change our expect down here to say Users because now that's what our h1 says, and the moment I hit save, we can see

that we now have our test passing again. And yes this is a silly test but I want to leave it here so that we have a working example of how to interact with jsdom. So now if we go over to the browser and load our application we can see that our data is coming back, and if we inspect and go to the Network tab, I'll just reload, so now we can see that request going through, this is the request to users on our API, we get a 200 OK and the response is the JSON that we're returning from Express, and now we know that our call is going through using Fetch as we expected. Now you might be wondering about the idea of sending a polyfill to everyone, let's discuss that next.

## Selective Polyfilling

Now you might be wondering if Fetch is already supported natively in some browsers, why are we sending our polyfill down to all browsers? Well in short, we did so because it was easy and it's also quite common. The idea is that you can remove the polyfill altogether later when all the browsers that you care about have added support for Fetch. But if you want to send a polyfill only to browsers that need it, there's a handy service called Polyfill.io which does just that, it offers a wide array of polyfills. Here's an example of using polyfill.io to polyfill only the Fetch feature, so if we put this at the top of index.html, Polyfill.io will read the user agent and use that information to determine if the browser requires a polyfill for the feature or features listed. Since I'm using Chrome it will send back an empty response since my browser doesn't need it, pretty slick. Now what if we need a wide variety of data to build our app and the services we need to call don't exist yet? That's just one of the reasons that you might want a robust mock API, so in the next clip, let's discuss approaches for mocking APIs and why it's so useful.

## Why Mock HTTP?

We've now set up our development environment to handle making HTTP requests, but it's often helpful to mock HTTP. Why? Well maybe you want to unit test your codes so that your tests run quickly and reliably, or maybe the existing web services in your development or QA environment are slow or expensive to call. Mocking HTTP means that you can receive consistently instantaneous responses. Or maybe the existing service is unreliable, with a mock API you can keep working even when the services are down. Maybe you haven't even created any web services yet. If you haven't decided how to design your web services, mocking allows you to rapidly prototype different potential response shapes and see how they work with your app. Perhaps a separate team is creating the services for your app, by mocking the service calls, you can start coding immediately and switch to hitting real web services when they're ready, you just need to agree on the API's proposed design and mock it accordingly, finally, maybe you need to work on a plane, on the road or in other places where connectivity is poor. Mocking allows you to continue working while you're offline.

## How to Mock HTTP

So does that sell you on the benefits of mocking HTTP? Assuming so, here's a few ways to get it done. If you're writing unit tests then Nock is a handy way to mock HTTP calls on your tests. You tell Nock the specific URL that you want to mock, and what it should return. Nock will hijack any HTTP request to the URL that you specified and return what you specified instead. This way, your tests become deterministic and no longer make actual HTTP calls, but if you're wanting to do day to day development against a mock API, you'll want something more, if you've already centralized all your API calls within your application then you can use this centralization to your advantage by pointing to a static file of JSON encoded data rather than making the actual HTTP call. Or you can of course create a web server that mocks out a real API. Thankfully there are libraries that make this easy such as api-mock and JSON server. I personally use JSON server, with JSON server you create a fake database using static JSON, then when you start up your JSON server it creates a web service that works with your static JSON behind the scenes, so when you delete, add or edit records, it actually updates the file. So this provides a full simulation of a real working API but against local mock data that's just sitting in a static file, this is really useful because the app feels fully responsive and you don't have to go through the work of standing up a local database and web server by hand. What if you want to use dynamic data instead of the same hard coded data? Well that's where JSON Schema faker comes in handy. JSON Schema faker generates fake data for you. You specify the data type you like, such as a string, number or boolean, and it will generate random data which you can write to a file. And you can specify various settings that determine how it generates the data such as ranges for numbers or useful generators that create realistic names and emails. Finally you can just go all out and wire up a fake API yourself using your development webserver of choice such as Browsersync or Express, of course, this is the most work, but it also provides you with the most power. So how are you going to decide between these options? Well in short, as you move to the right, you have to spend more time up front configuring, but in return, you enjoy more realistic experience and more power to customize. See, with Static JSON, your app will load the same data every time, and if you try to manipulate that data in any way, it won't be reflected upon reload. Now JSON Server actually saves the changes that you make to the data, so it increases the realism of your mock API. Now you can make your mock API more dynamic by using JSON Schema Faker, JSON Schema Faker can create different fake data every time you start the app. This can be really helpful for catching edge cases in your designs such as pagination, overflow, sorting and formatting and finally, setting up a full mock API from scratch using something like Express and a real time database filled with mock data of course gives you all the power to customize as desired. But if you don't already have a service later and a database you're on the hook to do all that hard work up front before you can enjoy a rapid front end development experience. In summary, if there's already a solid service layer available then I suggest putting it to use. But if a separate team is building a service layer and you haven't build it yet, I suggest trying a mock API so that you can move quickly without being reliant on a real API backend. The lessons you learn with your mock API can often help guide your API design. So now that we've talked about the different decisions, in the next clip let's talk about our plan for mocking HTTP in our starter kit.

Our Plan for Mocking

For this course, let's use a three step process to create a mock API, we'll build a few handy open source projects to use, first we'll declare the schema for our mock API using JSON Schema Faker, this will allow us to declare exactly what our fake API should look like, we'll declare the objects and properties that it will expose including the data types. Step two involves generating random data, JSON Schema Faker supports generating random data using a few open source libraries, faker.js, chance.js and randexp.js. Faker and chance are very similar, both of these libraries offer a wide variety of functions for generating random data including realistic names, address, phone numbers, emails and much more. Randexp focuses on creating random data based on regular expressions. Now JSON Schema Faker allows us to use faker, chance and randexp with our schema definitions. So we'll declare exactly how each property in our mock API should be generated, this will ultimately produce a big chunk of JSON, and the nice thing is, that big chunk of JSON will contain different data every time that we run JSON Schema Faker, and that's where our final piece comes in. JSON server creates a realistic API using a static JSON file behind the scenes, so we'll point JSON Server at the mock data set that we dynamically generate. The beauty of JSON Server is it actually supports create, reads, updates and deletes, so it saves changes to the JSON file that's being created by JSON Schema Faker. This way, the API feels just like a real API but without having to make an actual over-the-network HTTP call or needing to stand up a real database. This means that to get started on development, we just need to agree on the calls that we want to make and the data shape that those calls should return. Then the UI team can move ahead without having to wait on a service team to actually create those associated services, everyone can code to an interface and get back together later. Now that we've talked about the high level plan, let's explore the technologies that we're going to use in a little more detail in the next clip.

## Mocking Libraries

JSON Schema is a standard for describing a JSON data format. Of course, since JavaScript is the Wild West, this is just one of many so called standards for describing JSON structures, there's also JSON Content Rules, JSON LD, RAML and API related technologies like GraphQL, Falcor and O Data that specify their own standards for JSON structures, but in this course, we're going to use the JSON Schema standard that's being outlined here at json-schema.org. This is the standard that we'll be following to declare the shape of our mock data, and here's why, JSON Schema Faker is a handy tool that uses the JSON Schema standard. And enhances it by using some open source libraries for generating mock data. As I mentioned, the three libraries that we'll be using are Faker.js, chance.js and randexp.js, all three of these libraries come bundled with JSON Schema Faker. For more information on Faker, check out the GitHub repo, and also the GitHub.io site for more documentation. But I also recommend using Faker's interactive example, this is a great way to see all the different data that Faker can generate, and each time you click on the labels in this form, you'll notice that it generates different data. And Chance.js also has a nice dedicated website with a long list of detailed examples as well. There's a lot of crossover between Faker and chance.js, but again, they both come bundled with JSON Schema Faker, so it's your choice which one you want to use for a given call, with that said, it's worth carefully reading the JSON Schema Faker docs on GitHub, they provide a long list of examples for how to call Faker and Chance within your schema definition. As you can see here, they're using Faker to generate an email address.

The trickiest part is understanding how to convert the documented function calls for Chance and Faker into the JSON that you see here. Thankfully there are many examples within JSON Schema Faker docs, just be sure to carefully read the docs on Faking values if you get tripped up on how to call Faker, Chance or Randexp from within your JSON schema definition. This was the part that confused me most, but the examples helped me work it out. Also, be sure to check out the JSON Schema Faker repl online this way you can easily try different schemas and instantly what JSON they produce. I found this to be a great way to rapidly learn how to structure my schema. And here's the JSON repository, as you can see from the number of stars, it's hugely popular. The sales pitch is simple, get a full fake REST API with zero coding in less than 30 seconds. Now the biggest caveat with using this tool is it has strong opinions on what a FAKE REST API should look like. If you're a fan of hyper media, this won't do it. And if you have an existing API that has dramatically different assumptions than this makes, then this won't be very helpful, but if you haven't built your service yet or your service follows the same popular conventions as this library, then JSON server can help you stand up a mock API shockingly quickly. You're about to see how fast all these tools can be glued together into something seriously useful. I know there are a lot of moving pieces here but I think you'd be surprised how easily this all composes together.

Demo: Creating a Mock API Data Schema

Alright, it's time to mock some HTTP, here's the plan. Let's use JSON Schema Faker to declare a fake schema, it comes bundled with three handy libraries that we'll use to generate a random data, faker, chance and regexp. And once we've created our schema and generated our mock database file, we'll use JSON Server to serve it up and simulate a real API. Let's dive in. As we just discussed in the slides, we're going to use a combination of useful open source projects to create a mock API, to begin, let's define a schema that describes what our mock data should look like. Let's create a file called mockDataSchema within our buildScrips folder. And I'll just paste in the schema and then talk through the structure. If you don't want to type this you can grab the snippet from this Url. Now as you can see I'm exporting a chunk of JSON and this JSON describes the shape of our mock data. I'm declaring at the top level that our data structure is an object, and that object has a set of properties, the first property is users, and that users property has a type of array. I'm specifying that I want that array to contain between three to five items, and then below, I define the shape of the items that should sit inside the users array. I'm saying that inside the users array I should find an object and then again I define the properties for that object. As you can see there are four properties I'm defining, ID, first name, last name and email, the ID should be a number, I'm saying it should be unique because I'm trying to mimic a primary key in a database, and I want that minimum value to be 1, I don't want any negative numbers. Then I have a first name and a type of string, and you can see this is where things get interesting, I start using the faker library and I start asking for a fake first name, I do basically the same thing with last name, and then I also use faker on email to say that I would like a fake email address returned. Finally, down here at the bottom, I say that all four of the properties that we defined up here above are required, that means that they will always be populated. If I forget and I leave one of these out of the array, then it will occasionally leave one of these out, so that we can simulate an API that doesn't always send a property if it's not populated. And I also specified that my one top level property which is users is also required, so in this

case, our schema will always return all of these properties since I've required them all. Pay close attention to these required properties. This really confused me at first when I was wondering why some of my properties were occasionally not showing up. That's it, with only 34 lines of JSON we declared detailed rules about how our mock data should be generated. And now that we've declared how it should look, let's use it to generate mock data in the next clip.

Demo: Generating Mock Data

We just wrote the schema that declares the shape of our mock data, now we can use JSON schema faker to generate some mock data using this schema, to do that let's create another file in buildScripts and we'll call it generateMockData, this file will use JSON Schema Faker to generate a mock data set and write it to a file. And as you can see I'm pulling in JSON schema faker, I'm referencing the mock data schema that we just created and then I'm using FS which comes with Node and chalk to be able to color our output. I began by calling JSON.stringify on results of JSON Schema Faker, as you can see, I pass the schema that we just defined to JSON Schema Faker, so effectively, JSON Schema Faker is going to look at that schema, generate a lot of randomized data based on our schema, and then I'm going to convert that into a JSON string using JSON.stringify, so now we have a string of JSONs stored on line 14. Then we use Node's built in fs to be able to write our database file which I'm going to place in the api folder, we'll call it db.json, if any error occurs then I'll log it to the console in red using chalk, and if it succeeds, then I will say mock data generated and I will output that in green using chalk. And now that this is set up, let's write an npm script that makes all of this easy to call, so we can jump over to package.JSON and inside let's create a new script called generate-mock-data, I use babel-node to call my generateMockData file that we just created, and of course I use babel-node because I wrote it in ES6 just to make sure that Node can parse it. When we run this script it should write a random data set that matches the schema we defined to our API folder. So let's save our changes and see if this works. Npm run generate-mock-data. We got our green message so that's a good sign, and now we can see that db.json was written to the API folder, and if we open it up we can see that random data was generated that honors the shape we just defined. We can see that we're getting randomized Ids and realistic first and last names and email addresses. We can also see that there was an array of users generated as I requested, great, so we now have a simple repeatable way of generating random data that suits our specific needs. In the next clip let's put this to use on a mock API.

Demo: Serving Mock Data via JSON Server

Now that we have the mock data we need, let's start up JSON Server and tell it to use our mock data, now the great thing about JSON server is it will parse our JSON file and make a mock API for each top level object that it finds. So let's create a new npm script to start our mock API server. As you can see I'm telling it to use the db.json file that we generated and to surf the api on port 3001. Again, pick a different port if 3001 isn't available on your machine, but I'm deliberately choosing a different port than port 3000 which we're using to host our app. So let's open the command line and try it out, say npm run

start mock api. When you do, we can see the list of resources that JSON Server is exposing, in this case it found our top level object, users, but if we added more top level objects it would create an end point for each one, slick. Now let's take this URL and go back to browser. We'll open up a new tab and paste it in, and there we go, awesome, we can see an array of users is being returned as expected, so this is the mock data that's sitting in db.json but now it's getting served up over HTTP on a mock API, now I prefer for my mock data to change every time that I open the app, this way we're constantly viewing different potential edge cases in the system. Randomized data helps simulate the real world and it captures issues in development such as edge cases, empty lists, long lists, long values, it also provides data for testing, filtering, sorting and so on. So let's generate new mock data every time that we start the app, to do that, let's go back to package.json and we'll create a script that should run before we start the mock API, so I'll place it right before start mock api we'll call it prestart mock api, and remember by convention because this starts with the word pre but otherwise has a matching name, it will run before start mock api, and what we're telling it to do is to generate mock data before it runs start mock api, finally let's update the start script to start the mock api each time that we start the app. Simple enough, so now every time that I start the app it will generate new mock data and start up the mock api that serves the data, and the interesting thing about JSON Server is if we manipulate the data by making calls to edit or delete records it will actually manipulate the data file behind the scenes, this means you can even use this for integration tests or reload the page and see your changes reflected, it does a great job of mimicking a real api with an actual database behind the scenes. Of course to see this in action we need to update the application to hit the new mock api instead of that express api call that we created earlier in this module, so let's assume that the express server that we set up here is for our real production API and that the mock API that we set up is what we want to use during development. So what we need is for the application to intelligently point to the proper base url in each environment. To do that, let's create a file called baseurl.js in the API folder. This file will look at the host name to determine if the application is running in development. If it is it will point our mock API which is hosted on port 3001, if it's in production it will point at that production api that we set up that serves from Express. Alright so let's put this new file to use in our user API file, I'm going to add an import for getBaseUrl, here at the top of baseUrl and then I will store that in a constant right here, and of course we need to use this information in the API call below, so I will say baseUrl + Url, this way it will change based on the environment, and assuming this worked, we should now be able to start our app again and see that it's pointed at our mock API because we're in development. And now that it's up if we come over to the brower, we can see that our user data is displaying. And a quick note, since we're starting Express and the mock Api at the same time, the app may fail on the first load if it tries to call the mock API before it's up. If so just hit F5 to refresh. We can see that it is different data than we were seeing before so we know that we're hitting our mock API. We can also confirm this by coming over here to db.json and seeing that the first record is Kole Kessler and that is what we're seeing right here, so we know that we're getting the data from db.json served up into our app. We can also see this if we reload that we're making a call to port 3001, so our application is hosted on 3000, our mock API is on 3001, and it's returning that mock data that we just generated. And of course you'll have different mock data than me because every time we run the application now, it's going to generate realistic looking mock data. Now you'll notice that delete link? Well it doesn't work because we haven't wired up yet. But this is where things get really interesting. JSON Server supports manipulating data as well, so if we submit a request to delete our add

records it will write to the db.json file so our changes are persistent on reload. So the data will remain in db.json until we restart the app. So let's wire up these delete links in the next clip.

Demo: Manipulating Data via JSON Server

To prove that we can save changes to our mock data, I could just generate an HTTP request to the mock API that tries to delete some data, I could install a handy tool like Advanced Rest Client which is a crone app but let's go ahead and enhance the interface that you see here to support deleting a user when I click the delete link. First, let's go back to the code and create the necessary API call, so we'll open userApi.js and we'll export a new public function deleteUser, as you can see it looks at users and then passes ID that it receives. And you can see that it is delegating to a separate function called del which I haven't created yet, let's go ahead and do that, and we'll place it down here below the private get function, now this might seem rather redundant but this is the same pattern that we followed when we were setting up get and get users. This private del function gives us a centralized spot to handle all our delete calls, so if we add other deletion functions related to users, each public function call is nice and short, and I had to call this del because delete is a key word in JavaScript. And now we've set up the functions that we need within our user API, let's shift our focus over to the UI. We'll go to index.js and add some code to make these delete links work. After we make our call to delete users, we're currently populating the table, but let's do a little more work in here, I'll go ahead and paste this in, and what we now want to do is get a reference to all of the delete links on a page, and to do that we're going to look for anything with a class name of delete user, as you can see, all the delete links have a class of delete user. So now I will have an array like structure that I can iterate through, so I'll use array.from to be able to iterate through the list of delete links and then attach a click handler to each one, I'll prevent defaults so that the click doesn't actually produce any change to the URL, I'll call deleteUser and then I will remove the row that we just clicked from the dom. And again this will all be potentially cleaner in React, Angular, other popular frameworks, but I just want to use plain vanilla JavaScript here to avoid adding confusion. And one final touch, since I'm calling deleteUser right here we need to add it as an import from our user API. And with that the UI should now support us deleting a user from our mock database, so let's give it a shot, looks like we do have an unexpected token on line 35 in index. So I was just missing a curly brace, and parenthesis, and save, looks like that solves our syntax error. Now let's go back over here and refresh. And now when I click delete we can see that works. We can watch the network down here and see the call go through to delete the different users, if I click on one of these and look at the headers, we can see we get a 204 no content, we can see that the delete is going through as our request method as expected, and here's the cool part, if I hit Refresh right now, only one record is here because when I click delete on those two, it really did write to db.json, if we come back over here, we can see now our db.json only has one record when before it had three. We can also see that JSON Server is continuing to log all the different calls that are being made to our mock API. This is really handy when you're debugging calls along the way, and the great thing is this data will persist until we restart the app and new random data is generated. And a quick note, you may have to hit CTRL+C multiple times to kill the running process, since now we're running multiple processes on the same command line. You'll notice that JSON Server throws an error when you kill it this way but there's no impact so you can ignore

it. If you prefer, you can kill the terminal and open a new instance. So wow, we just covered a lot of moving parts but the result sure is handy, now let's close out this module with a short summary.

## Summary

In this short module we began by reviewing how to choose an HTTP library. We saw that HTTP is the low level option in Node, but you'll probably want to use requests with Node due to its streamlined API. In the browser, you can choose the old standards like XMLHttpRequest and jQuery but Fetch is probably what you should reach for since it's the new standard that streamlines the clunkyness of old XMLHttp requests. Just remember to pull in the Fetch polyfill so it will work properly cross browsers, or if you're looking for a full featured library, especially one that works in both Node and a browser, then isomorphic Fetch is the most future friendly approach since it utilizes the browsers built in Fetch support if available, however you can also consider using the XHR library on npm, SuperAgent or Axios. All of these are excellent options regardless of whether you need to run on both Node and the browser. And we closed out this module by exploring HTTP call mocking, if you're testing the way to get that done is Nock, and if you're needing to mock an API for development then the simplest way to get that done is likely just some hard coded JSON. If you have a small app that's perhaps all that you'll need but if you want to simulate interactivity then a custom web server approach involving JSON Schema Faker and JSON Server likely makes more sense, we saw that JSON Schema Faker is quite powerful and contains enough built in intelligence to create realistic fake data for a wide variety of scenarios, and of course if you want to go fully custom you can configure development web server of choice to simulate a real API. This is certainly the most work but also offers the most complete flexibility. Now we have HTTP request taken care of, we have a powerful foundation for building real applications, so in the next module let's put all this to use, we'll discuss key principles for project structure, we'll learn why demo apps are so important and we'll build a quick demo app that helps convey best practices. And in the final module we'll wrap up the course by creating an automated production build.

# Project Structure

## Intro

It's been a long road, but we're well on our way to enjoying a seriously robust, comprehensive, and luxurious JavaScript development experience. But we still have two major pieces left to consider in these final two modules. In this module, let's explore how to put all this to use by discussing project structure. In this short module, I want to begin by explaining why I believe your team starter kit should include a demo application. Then, we'll move on to discuss three specific project structure tips to keep in mind as you're building, not just your demo app, but any future JavaScript app. Alright, let's begin with my sales pitch on why your team needs a demo app.

## Why a Demo App?

I believe a demo app should be considered a critical piece of your team's starter kit. Why? Because many people learn best by example, so a working example really helps. Let's just review some decisions that are clearly conveyed via an example app. A demo app conveys expectations around suggested directory structure and file organization. It helps encourage consistency by codifying patterns for how developers should work with the libraries and the frameworks you've selected. It shows example tests that are passing, so developers have a point of reference for various testing scenarios, naming conventions, file placement, and mocking strategies. It provides a realistic example of a mock API, working in your domain. This gives developers a big head start, since referencing examples is often quicker than pulling up docs for desperate packages. It provides a working, automated deployment, so there's already a clear recipe for use, for future apps. And it gives you a single place to codify your decisions. The demo app should reflect your coding standards. It's a place to update as you learn new techniques and patterns that you want to share with the team. Finally, and perhaps most importantly, it offers an interactive example of the starter kit working in a realistic scenario. This helps new team members understand what life is like working in the selected stack. Okay, so assuming you're on board with creating a demo app for your team's starter kit, let's consider a few tips for structuring JavaScript projects in the next clips.

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

Before we build the demo app, let's take a moment to consider project structure. I have three important tips I want to share to help you avoid some of the pitfalls I often see in JavaScript code bases. My first tip is simple: JavaScript belongs in a .js file. But, when doing web development, some of you are tempted to simply slap JavaScript code in a script tag. Of course, you'd never do that, right? It's understandable. It's so easy to just slap a script tag onto the html page and start coding. But let's pause for a moment and consider why this should be avoided. There's a long list of downsides. When I do something like this, it's important to understand what I'm losing. The answer is that I'm losing a ton of power to do my job better. I'm losing the ability to leverage all of the goodness that we just set up. See, if I do this, how do I write automated tests for this? How do I link this code? How do I reuse this? Oh, and if you just mumbled "copy and paste," ehhh, wrong answer! Now what if I want to use ES6, typescript, or some alternative language that transpiles the JavaScript? What if I want to be explicit about my code's dependencies by using ES6's import keyword? The answer to all of these questions is, you can't. By putting your code inline with an html, you're losing all of this power. And these are just a few of the fundamental issues with this approach. So my advice comes down to a simple maxim: JavaScript belongs in a .js file. Writing JavaScript inline within an html file should be avoided. And please, for the sake of everyone's sanity, don't use some server-side language to generate your JavaScript. This example from Stack Overflow makes me want to cry tears of mourning for the maintenance programmer. Let's just think about the developer experience here. All of the code is the same color. There's no auto completion support. Your editor can't highlight any typos, so you won't find out about your mistakes until run time. And you don't get to enjoy any of the goodness that we worked so hard to set up in this course. Yet, I

see this pattern all too often, when server-side developers who are new to JavaScript need to perform some custom logic based on data that's in the database. So, let me clarify a better way to get that done. If you need your code to respond differently for different users, instead, inject JSON from the server into your application. I call this the Configuration Object Pattern. Now, I'm far from the only person suggesting this pattern. As you can see, Stack Overflow is using it here. Since Stack Overflow is using C# behind the scenes, you can imagine that there's a C# class which contains all of this data, likely pulled from their database. And they use a JSON serializer in C#, likely JSON.net, to generate this JSON. Their static JavaScript code looks at this JSON and this is what customizes my Stack Overflow experience. Bottom line, avoid dynamically generating JavaScript code. Instead, dynamically generate some data that your JavaScript code can use. And this really shouldn't come as any surprise. We don't generate custom C#, Java, Ruby, or Python, and so on, just to dynamically provide custom behavior for each user. Instead, we pull data from the database and we use that data to determine what logic should run. JavaScript is no different. The solution is to use data from the server to fork your code's logic as necessary.

Tip 2: Consider Organizing by Feature

Time for my second tip. On larger, more complex projects, consider organizing by feature instead of by file type. There are two popular ways to organize your code: by file type or by feature. When you organize by file type, all files that serve the same purpose are placed together. This is a popular approach when working with MVC frameworks, which commonly expect you to use model, view, and controller folders to organize your application. However, the downside of this approach, is you end up having to bounce around the file system to open and work with related files. So, I recommend organizing by feature on larger projects. The larger the project, the more organizing by feature pays off. Because you can go directly to the feature that you're working with and all the related files are sitting inside.

Tip 3: Extract Logic to POJOs

My third tip is to strive to extract as much logic as possible into plain old JavaScript. Some would call this POJOs. When I say POJO, I mean Plain Old JavaScript Objects. Java and C# developers would recognize this term, since Java developers also use the term POJO to describe Java classes that have plain logic inside and no framework specific concerns. .Net developers use the term POCO to describe the same thing, though, in that case it stands for Plain Old CLR Object. The point is, these files should contain plain logic, that isn't tied to any framework. When structuring your application, strive to place as much logic as possible in plain JavaScript. For instance, if you're working in React, much of your logic should exist outside of React components. This makes your logic easy to test, easy to reuse, and helps minimize your ties to the framework that you've selected. This minimizes the impact of switching to a different framework down the road, because, hey, we're in JavaScript. Let's face the facts, we'll probably be doing that. To see an example of this philosophy, check out the demo app in the React Slingshot starter kit on

GitHub. You'll see that key logics, such as date formatting and core calculations are handled in plain JavaScript in a folder called utils. Although this project uses React, these are plain JavaScript functions that aren't tied to React in any way.

## Summary

Let's wrap up. It's really helpful to include a working example app in your starting kit. This gives everyone clarity on recommended approaches for directory structure, file naming, framework usage, testing, API calls, deployment, and more. It provides an interactive example of what it's like to work on your team. Then we moved on to a few ground rules for structuring your project. Put your JavaScript in a JavaScript file. This principle is foundational. If you place JavaScript inline with an html, you lose all the benefits that we've worked so hard to set up throughout the course. Consider organizing your demo app by feature, instead of technology or file type, especially if your team typically builds large and complex JavaScript applications. And third, extract your logic into plain old JavaScript objects. Avoid embedding too much logic in framework specific files. Extract your logic to pure functions that are easily testable. I recommend building your demo app using your preferred JavaScript frameworks and libraries. And be sure to select a domain that's related to your business. Keep these tips in mind as you structure your demo app. Alright, there's one very important piece left. We need to prepare our app for production. So, in the next module, let's create an automated production build, including minification, bundling, bundle splitting, and more.

# Production Build

## Intro

Of course our application isn't very useful until we actually prepare it for production. So in this module, let's create an automated production build. We'll cover a variety of considerations including minification to speed loads, with source maps generated to support debugging in production. Hey, let's be honest, you and I both know this happens. We'll setup dynamic HTML handling for production specific concerns, and cache busting to ensure that users receive the latest version of our code upon deployment. We'll setup bundle splitting so that users don't have to download the entire application when just part of it changes. And finally, we'll setup error logging so that we know when bugs sneak their way into production. Now this sounds like a lot of work, but as you'll see, this moves fast. Alright, let's dig in.

## Minification and Sourcemaps

Let's begin by discussing minification. Minification is about speeding page loads and saving bandwidth. So how does minification work? Well a JavaScript minifier uses a number or tricks. It will shorten variable and function names, remove comments, remove whitespace, new lines, and more. The code

still functions the same, but the resulting file size is much smaller, which helps speed page loads. Minification basically removes all the things that only humans care about, and leaves all the things that computers need. And some of the newer bundlers like RollUp and Webpack2 go even further by eliminating unused code via a process that's commonly called tree-shaking. We're using Webpack1 for this course, but this feature will soon help reduce the size of bundles by excluding any unused code from our final bundles. And through the beauty of source maps, we can still debug our minified code. As you'll see in a moment in the demo, we'll continue to see our original source code in the browser as we debug. For more info on source maps, refer to the bundling module earlier in this course.

Demo: Production Webpack Configuration with Minification

Now let's put Webpack to work to bundle and minify our app code for production. To begin configuring our application for production, let's make a copy of our development webpack config. And we'll call it webpackconfig.prod.js. And now let's tweak some settings for production. First, we'll change the Dev tool setting. Remember that this setting specifies how source maps should be generated. We explored source maps earlier in the bundling module. Let's change the Dev tool setting to sourcemap, since that's what's recommended for production. It's a little bit slower to build but it provides the highest quality sourcemap experience. This will assure that we can still see our original source code in the browser, even though it's been minified, transpiled, and bundled. That's the beauty of source maps. And we're going to write our production build to a folder called dist so let's change the output path. When building the app for production, we'll write physical files to a folder called dist. This is a popular convention and it stands for distribution. Next, let's set up minification. We want to minify our code for production so let's add our first plugin to the array of plugins. We're going to use a plugin called UglifyJs. And I like to put a comment above each of my plugin references. So you can see we're calling webpack.optimize UglifyPlugin. And now that we're calling some specific webpack features we need to add the import up here. And before we minify, let's use another handy webpack plugin that will eliminate any duplicate packages when generating the bundle. This plugin's called the Dedupe plugin. So this will look through all the files that we're bundling and make sure that no duplicates are bundled. Now we'll enhance the webpack config with additional features throughout this module but this is a good start. Let's now shift our focus to writing a script that will run our production webpack config build. So we'll go over to buildScripts and create a new file called build.js. And here's all it takes to run our webpack build for production. We're importing webpack, our production config, that we just defined, and chalk so that we can color our output. And then we're calling webpack and passing it that webpack config. We're handling any errors that might occur, otherwise we return zero which signifies success. So this is pretty simple, but in the real world you'll likely want to add a little more to this. So let's enhance this script a little bit. First, let's go up here above the call to webpack and declare that we are running Node in production mode. Although it's not necessary for our setup, I'm adding this line here because this is important if create a Dev specific configuration for Babble in your .babblerc file. See, Babble and potentially other libraries you may use look for this environment variable to determine how they are built. And before we get started running the production build, I like to output to the console just so we can see that the production build has started. Since we're doing minification, as you'll see, the

production build does take quite a few seconds to run. So it's nice to get some notification that it is doing the job. I like to display some stats to the command line. Now this looks like a lot of code and it isn't required but this ensures that warnings, errors, and stats are displayed to the console, and at the bottom we display a success message if everything worked. So this is for displaying any errors that occur, this is for displaying warnings that occur. I display the stats right here. Finally, we just output a message so that we know that our production build has succeeded. Great, so there's quite a bit of code here but it's really conceptually simple. This just runs our production webpack config. I've added some extra goodness here, just to improve our experience. Let's save our changes and in the next clip let's try this out by setting up an automated production build.

Demo: Configure Local /dist Server

This isn't required, but I like to run the final production version of the app on my local machine, just so I can make sure that everything looks good. This can be really helpful when you need to debug an issue with a production build. So let's create a file called distserver in the build scripts. We already have a source server that serves our source folder, now we'll have a distserver that serves our dist folder. So let's just copy the content of our source server and paste it over into distserver, because we're only going to make a few minor changes here. First, let's remove any webpack calls here, because we're no longer going to be interacting with webpack for our distserver. We're going to be serving up just the static built files. So we'll remove the two webpack related imports at the top, we'll also remove the call to the compiler, and the calls to configure webpack dev middleware. So our file gets simpler. And then, the other thing that we need to add is to now add support to express for serving static files. So we'll add line saying app.use and we'll tell it to serve static files in the dist directory. And for production we'll be serving index.HTML from the dist folder rather than the source folder. One final tweak that I like to make for our dist server is enabling Gzip compression. Your production webserver should be using gzip, and if it's not, pause this video and go turn it on. Now anyway, I like to enable gzip so I can see the final gzip file sizes when I'm serving the app locally. This gives me a clear understanding of the file sizes that will be sent over the wire to the user. To do this, let's import compression, and then down here above our call to express.static we'll add a line in to use compression. Make sure that you add the parentheses so it's invoked and with those two lines of code we've now enbaled gzip compression in express. And that's all we need to do to configure our dist server for serving up our production app locally. Again, this is not for use in a real production scenario. I'm only creating this so I can serve the app on my local machine just to confirm that the production build works locally. Then, it's a separate decision to move all these files up to serve them on some host. Perhaps a cloud provider. And also yes, I'm leaving in the hard coded data for users. Again, just pretend that this is hitting real data and production. And speaking of API calls, we also need to decide what API we'd like to use when we're checking out our production build locally. So let's work on that next.

Demo: Toggle Mock API

I prefer to hit the real API when we're testing out the production build locally. So let's open up baseURL.js which is in the API folder. Remember, this file contains logic that points the API to either our mock API or the real API that's getting served by express. To do so, it currently checks whether the app is running on local host. Let's rework this logic so it instead looks for the query string parameter use mock API. So I'm going to replace this function with a one liner and this one liner will say get query string parameter by name and it's going to look for a query string parameter of usemockAPI and if that exists in the query string, then it's going to point to our mock API. Otherwise it will point to the real API that's being hosted by express. Now this getQueryStringParameterByName doesn't exist so let's paste that in. Yeah, sadly this big chunk of code is necessary for getting a parameter from the URL. Honestly, I just grabbed this function off of stack overflow, so cross your fingers, let's hope it works. Of course there's other ways to do this with libraries but again, I want to use just plain JavaScript here. There are easier ways to get this done, for instance, with jquery, but this function will do the trick. Now with this change, we should be able to easily swap between the real and the mock API during development by just adding use mock API to the query string. So let's open up the terminal and make sure that this tweak worked. We'll say npm start minus s. And our app starts up successfully. As we can see, it's not using the mock data, it's using the hard coded data that's coming from our express based service. So this is our production API. We can confirm that by coming over to source server and seeing our hard coded data here. Again, it's not an actual production API but pretend this is our production API. We're just hosting it locally here. So I should be able to come up here, add a query string, say useMockAPI=true, and when I do now we end up seeing our mock data instead. So now we have an easy way to switch between our mock API and our production API. I find this can be really handy, not just for working with our production build but also for development on a day to day basis, being able to switch between the production API and our mock API gives us flexibility that we need throughout the day. And now we're almost ready to run our first production build, but we can't quite yet because we haven't written the necessary npm scripts to automate it. So let's set that up in the next clip.

Demo: Production Build npm Scripts

In order to automate the production build process, let's add some npm scripts to tie all this goodness together. I'm going to use four small npm scripts to orchestrate the production build. Let's add them here at the bottom of the list. Okay, let's talk through how this works. The command that we'll run to build our app for production is npm run build. This will run the build script that we set up earlier, and by convention pre-build will run before that. That will clean our dist folder, which we can see deletes the dist folder and then recreates it. This way any previous files that were there are wiped away before we write to that folder. We also want to run our test and lint our code. Once all that's done, the build occurs, and then after the build is completed, the post-build script will run, and that is where we will start our dist server. So after our build's completed, we will start it up and serve it up locally. And now that we have our script set up, we're all set to give this a shot. Let's try running our production build and see if anything bursts into flames. We'll say npm run build minus s. We can see our test pass, our linting runs, we get our notification, and our final confirmation in green, but boom, flames indeed. We get a 404 because if we go back over to our dist folder, we can see there's no index.HTML inside. And if you

think about it, that makes sense. Webpack is currently configured to write our JavaScript files and handle our CSS but we never setup anything to handle writing out index.HTML to the dist folder. But the good news is, we can see that our minified JavaScript is being written as we asked. If we click on the bundle, that's not very readable at all, but that's a good thing in this case. Because the source map makes it all readable in the browser. And although you don't see any CSS files here, remember CSS is being bundled into our JavaScript and thus being generated via JavaScript. If you prefer to generate a separate CSS file, I'll show how to do that a little later in the module. Let's take a look at the detailed output on the command line. Webpack shows all the files that were bundled including their size. This output is handy because we can see precisely what files are being bundled and we can see the size of our final bundle. It's also common to see some warnings down here but this isn't code that we wrote, it's code from the libraries that we're using so we'll go ahead and ignore these warnings. Now clearly, we need to decide how to handle our HTML for the production build. There are multiple ways to handle this so let's explore this topic next.

## Dynamic HTML Generation

When you bundle your code, you obviously need to reference it. And if you're doing web development then of course you'll end up referencing your bundle in an HTML file. But what if you want to run some slightly different HTML in production than development? So why would you want to manipulate HTML for production? Well there are many potential reasons. If you're bundler's generating a physical file for you, wouldn't it be nice to automatically reference the bundle in your HTML file? And as you'll see in a moment, we'd like to generate dynamic bundle names so that we can set far expires headers in production in order to save HTTP requests. When bundle names are dynamic, we need a way to reference the dynamic bundle name in our HTML. And what if we want to inject some scripts or resources only for production. We'll see an example of this in a moment when we discuss error logging. Finally, maybe we just like to save a little bandwidth by minifying our HTML. The point is, there are a variety of reasons to manipulate HTML for production. Now when you're generating a bundle, a common question is how to setup your index.HTML file to reference the bundle. This example shows the simplest approach. A hard coded reference to bundle.js. This has been working great for us so far, but there are other more powerful approaches to consider for handling all the issues that we just discussed. I see three specific options for handling your HTML. If you have a simple setup, you might just want to hard code in a reference to bundle js, as we've done so far in the course. This is the simplest approach. However, maybe you want to dynamically add some other code to the page for production. That's when you want to dynamically generate your HTML file. One obvious way to do so is via Node. You can write a Node script that copies your HTML file and uses regular expressions to manipulate sections or replace place holders. Or if you choose webpack as your bundler, there's a powerful approach that I prefer to use called HTML-webpack-plugin. This plugin simplifies the creation of your applications HTML file. It can create the traditional HTML boiler plate for you. Or you can declare a template, which is my preferred approach. This plugin is especially useful if you're doing cache busting in webpack by generating a different file name each time your assets change. We'll set that up in a moment.

Demo: Dynamic HTML Generation

Alright, let's get back to the code and dynamically generate our HTML using the HTML-webpack-plugin. As you just saw in the previous demo, our webpack build isn't handling our HTML so we can't actually load the production build in the browser yet. Let's fix that. The simplest way to handle this is to simply copy index.HTML from the source folder to the dist folder using the command line call or Node. We can even use Node to tweak the HTML in various ways for production. But doing all that manually will just hold us back because we're going to use some other powerful webpack features in the coming clips. And the features we're going to use will be a lot easier to pull off if we setup webpack to handle our HTML. To do that, we're going to use HTML-webpack-plugin. This plugin will dynamically generate our index.HTML in all environments. So to begin, let's update our production webpack config to use HTML-webpack-plugin. First, let's add the import to the top, for HTML-webpack-plugin, and then let's add a reference to our plugins array. Don't forget the comma after defining it. So this will create an HTML file that includes a reference to our bundled JavaScript. We're going to declare that the index.HTML that's in our source directory is our template and then I'm setting inject to true which tells webpack to inject any necessary script tags for me. This means we can remove the script tag from our index.HTML. So let's go ahead and do that now. We don't need this anymore because the webpack plugin is going to dynamically add in any necessary script tags for us. You'll see why dynamically generating our script reference is important in a later clip as we setup bundle splitting and cache busting. And since we're dynamically generating our index.HTML now, we need to update our development webpack config to use the HTML-webpack-plugin as well. So let's copy the work that we did in our production config, over to our Dev config. I'll need the reference to the plugin, and I'll also need to call the plugin here in the array. Be sure to remove the unnecessary trailing comma. And save both of our webpack configs. And now that we have webpack configured to handle our HTML in both our development and production builds, we're ready to run our app in production mode. Let's open up the terminal and give it a go. Npm run build. And I like to put the minus s, just to engage silent mode so that we don't get all the noise to the terminal. We can see our test run, linting pass, and great, our app loads in production mode. If we open the network tab, we should be able to see the results. So let's go to network, and hit reload. And we can see that our minified and Gzipped bundle is only 4.6K. Now that's tiny. Note that the actual bundle size is 12K as we can see over here in the output. So gzip compression is helping us out. As we can see here, gzip is enabled so our work to enable gzip in express is operating as expected. And again, we've enabled gzip on express so that the size that we see here is an accurate representation of the typical production web server configuration. Since your production web server should have gzip enabled. Oh and just remember, these delete links won't actually work when we're pointed at our real API since we never wired up the real API to handle delete calls. Now let's jump back over to our webpack config and make another tweak. We can enhance the configuration of HTML-webpack-plugin to save us some more bandwidth. Since we're dynamically generating our HTML, we can configure HTML-webpack-plugin to minify our HTML. So let's add some configuration to further tweak it's output. There's a long list of settings here, but I've enabled them all. We're going to remove comments, collapse whitespace, remove any redundant attributes, remove empty attributes and more. So let's save our

changes and see how this works. We should now be able to run the build and see our HTML minified. If we view page source, we can now see that our HTML has been minified, we can also see that the reference to the bundle is getting injected by the HTML-webpack-plugin as we'd expect, very nice. I love how easy and declarative webpack makes this. So excellent. We have a simple, functional, automated build. But there's still room for improvement. What if we were building a large app. In that case, we'd likely want to split our bundle to help speed page loads and save bandwidth. Let's handle that next.

## Bundle Splitting

As you build larger apps, you may find it useful to split your JavaScript into multiple bundles instead of creating a single javascript file. This is commonly called bundle splitting or code splitting. So why bundle splitting? Well, what if we build a large app or an app with many pages that are routed client side. If we split our bundle into separate files for each client side page, we can speed the initial page load by only requiring the user to download the JavaScript necessary to render that page. And if we push updates to our application source code, it'd be nice if users weren't forced to download all vendor libraries we've chosen to use in our app. For example, if we build an application with angular, react, or use utility libraries like low dash, it would be nice if those items were bundled in a separate file that's cached separately. This way when we update the app, our users will only have to download the updated app code. So bundle splitting saves bandwidth and thus assures that our users have a higher performance experience. We shouldn't expect our users to download a large bundle on initial page load or to re download a huge bundle every time a small portion of the app source code changes.

## Demo: Bundle Splitting

Of course, the way that you split bundles will depend on the bundler that you select. But since we're using webpack, let's configure it to do bundle splitting as part of our production build. So far, we've setup webpack to bundle all our JavaScript into a single bundle. This works great on smaller apps, but as your application grows, it's helpful to split the bundles so the user only downloads the JavaScript they need for the current section of the app that they've loaded. Sometimes people split the bundle per page. Imagine we built a single page app with three pages. We could split our bundle per page. But another approach to consider on smaller apps is to split third party libraries into a separate bundle from our application code. The reasoning here is, if the application code changes, our users won't have to download all of the third party libraries that we're using again. The browsers will continue to use the cached version. Now webpack supports defining multiple entry points. We've only defined one here, but you can see that in array infers that we can add multiple. Now instead of defining an array, I'm going to define an object because we are going to define keys for each one of the entry points that we define. For this one, I'm going to call it main, because this will be our main JavaScript bundle. But we also want to define a second entry point, and I'm going to call it vendor, because it will contain all of our vendor code. We haven't created this vendor file yet, so let's go over to the source directory and create a new file called vendor.js. And here I'm going to paste in a whopping one line of code. Okay, that looks like

more than one line, but line 17 is the only actual code. I'm disabling ES lints warning for no unused variables since they're no usage of this variable here. And then I just added a comment at the top so that people are clear about why we created this vendor.js file. Of course, the only library that we're using for our silly demo app is the what working group fetch poly fill, but you get the idea. In a real application, you'd likely reference jquery, angular, react, bootstrap, and so on here. Any third party tools that you use could be listed in this file. And everything that we define here will be bundled up separately by webpack into a file called vendor.js. Of course, if you wanted separate bundles for different pages of your app, you could declare those using the same pattern. Just add one entry point to your production webpack config per page. So let's close vendor and go back to our production webpack config, because we're not quite done configuring code splitting. To actually perform code splitting, we need to reference another built-in webpack optimization that's called CommonsChunkPlugin. So let's add it to the list of plugins down here. Here, we're telling webpack to generate a separate chunk using the code that's referenced in our vendor entry point. So note that this name corresponds with the key that we declared in the entry point up here. Make sure these two are in sync or this will just end up generating an empty file. Now here's how this works, the CommonsChunkPlugin moves modules that occur in multiple entry chunks to a new chunk. And to clarify, I am now using the terminology that webpack uses. They call these chunks, I tend to call them bundles and talk about splitting bundles, webpack typically talks about splitting chunks. But chunks and bundles are synonymous in my mind. So in this case, the CommonsChunkPlugin will intelligently look at the items that we imported in vendor.js and it will leave them out of the separate bundle main.js. So since we referenced the FetchPolyFill in vendor.js, this plugin will assure that the FetchPolyFill is placed in our vendor.js file and omitted from main.js. And to clarify, without this plugin, splitting wouldn't actually help, because our vendor libraries would still be in our main bundle as well, causing people to download our vendor libraries twice. And there's one final detail. Now that we're generating multiple bundles, we can no longer hard code the name of the file that we're outputting here. Instead, we need to declare a place holder by using square brackets, and we'll call that place holder name. So this tells webpack to use the name that we defined in the entry point. So what it will now generate a main.js and a vendor.js And since we're using HTML-webpack-plugin, it will automatically write references to both of these files in our HTML file. Nice. Alright, that's all it takes so let's save webpack config and give it a shot. Npm run build. We can see the app starts up just fine. And if we come over here, and scroll up, we can now see that we generated two separate chunks, main.js and vendor.js. Remember before that main.js was 12K, but now vendor.js is holding 7.3K. So we can see that the size has been split between the two. We've also generated mapping files for both main.js and vendor.js. And of course, if we come over here, and view page source we can see that vendor.js is referenced as well as main.js in our code, just like we'd expect. And if we inspect here and look at the network tab we should see both getting requested. And we can see that their size after being gzipped are 2.9K and 2.6K. So this is a handy way to speed page loads and save bandwidth by avoiding requiring your users to download all of your JavaScript when only some of it has changed. But there's more that we can do. In the next clip, let's explore cache busting.

Cache Busting

To save bandwidth and avoid unnecessary HTTP requests, you can consider configuring your production web server so that your JavaScript bundle doesn't expire for up to a year. If you go this route, you need to enable cache busting. So why bust cache? Well first, you can save HTTP requests, because as long as you know that you can bust cache, you can set headers that tell your users browsers not to request your assets for up to a year. This means after someone downloads your JavaScript file, they won't make another HTTP request for that file for up to one year. So this both speeds page loads and saves bandwidth. Also, when it's time to deploy an update to your app, you can assure that the user immediately receives the new bundle by generating a new file name for that bundle. You can force a request for the latest version. So here's our two step plan for busting cache. First, we need to hash the bundle file name. This way the file name will only change if the bundle actually changes. This assures that if we rebuild the app, and there are no changes to the JavaScript bundle, it will continue to have the same file name. Second, since the file name is now generated dynamically, we need to make sure that the file name reference in the corresponding HTML file is set accordingly. So we'll generate our HTML dynamically and inject the proper file name as part of the build process. To make all this happen, we'll continue to use the HTML-webpack-plugin.

Demo: Cache Busting

Time for more command line fun. Let's use HTML-webpack-plugin along with a new tool called webpack-md5-hash to setup cache busting. To save bandwidth and avoid needless HTTP requests, it can be helpful to configure your web server to send far future expiration headers. This way your customers browser won't request your assets again for up to a year. I won't get into how to configure your web server to accomplish setting far future headers, but the basic idea is to configure your web server to send headers that specify that your applications JavaScript files shouldn't expire until some date in the distance future. But the problem is, when you do this, how do you update your app later? The answer is, you have to bust cache by deploying your application with a reference to a new file name. Webpack can make cache busting straight forward by generating a deterministic cache for each bundle and appending it to the file name. This way the file name only changes when the code actually changes. To accomplish this, we'll use webpack-md5-hash. This package hashes files and creates a deterministic file name so that our file names will only change when our code changes. To make this happen, we'll make three small changes in our production webpack config. First, let's add the necessary import at the top of webpack config, second we'll add the related plugin down here to the array of plugins, and finally we can put it to use by updating our file name format to use the hash that webpack-md5-hash generates. I'm going to do that by referencing a variable that it creates called chunkhash. So this format says name each bundle with a prefix that we defined up in the entry point, then add a dot, then add a hash, and finally add .js on the end. And with this setup, now our file names will change only when we change the code. So let's run the production build and see this in action. And now we can see that our file names have hashes placed in the middle of them. And since we're using HTML webpack config, if we open index.HTML, we can see that the references were dynamically written for us. So there's the vendor reference, and there's the main reference. This is a huge benefit of choosing a comprehensive tool like webpack. All of these concerns are handled in a cohesive manner with just a little declarative code.

Remember we just added three lines of code to make this happen. And if you rerun the build, you'll see that the file names stay the same, but if you change a line of code in rebuild the associated bundles name will change because it will hash to a new value. Slick. Now earlier I mentioned that our CSS is getting bundled in our JavaScript file. So you may prefer to deploy a traditional, separate CSS file with the same cache busting setup in production. So let's make that happen in the next clip.

## Demo: Extract and Minify CSS

Right now, webpack is embedding all CSS into the JavaScript bundle. That's why we don't see a CSS file generated in the dist folder. Our CSS is dynamically generated using JavaScript. Now I prefer to generate a traditional separate CSS file for production so that I can utilize the same cache busting techniques that we just saw. I've also noticed a flash of un-styled content when embedding CSS via JavaScript. So I believe it's worth taking a moment to configure webpack to generate a separate CSS file for the production build. To extract our CSS, let's use the extract text plugin. Setting it up requires just three lines of code. First, we'll go to the top and add the import, second let's call the plugin down here in the array of plugins, and we're using the same syntax here for naming our CSS as we did for our bundle. This way cache busting is enabled for our CSS as well. We'll only get a new file name when the CSS has actually changed. And finally we need to update our CSS loader down here at the bottom so that it will call the extract text plugin. So I'll replace the current loader with this. It's important to note that we don't need the style loader anymore and our CSS will be minified to save bandwidth so adding the query string source map here on the end declares that webpack should generate a CSS source map. And if we open the terminal and run the build, let's make sure it still works. Great we can see our styles are still applied. And if we look over here in our dist folder, we can see our CSS files are created along with a map. And if we look in index.HTML, we can see that the reason that our styles are working is because HTML-webpack-plugin is also automatically adding in the CSS link that's necessary up here at the top. Not bad for three lines of code. So if we look at the dist folder, we now have seven files, an HTML file, a bundle of our apps JavaScript, a bundle of our vendor libraries and source maps for our CSS, and our JavaScript. Our entire app is handled by these seven files. And I'd say this app is ready to deploy. But wait, before we do, how do we know if we made a mistake? So in the next clip, let's setup error logging so that we're aware when JavaScript errors occur in our users browsers.

## Error Logging

What happens today when your app throws a JavaScript error? Are you aware when JavaScript errors occur in production? There's a long list of services available to help in this area. Sentry, TrackJS, New Relic and Raygun are a few worth considering. Some services like TrackJS are specific to JavaScript, while others like New Relic, provide broader, performance related information. Personally, I use TrackJS on a few production apps and I've been quite happy. So how do you choose among all these options? Well, when you're evaluating error logging services, here's some key concerns to consider. Does it provide error metadata? For example, does it tell me what browser the error occurred in? Does it capture stack

traces? Does it capture previous actions that the user was performing so that I can reproduce the issue? Does it offer a custom API so I can augment error logging with my own contextual data? Does it offer notifications so I can receive emails when errors occur? And can I integrate with other popular platforms like Slack so we're notified there instead? Can I filter out the noise by aggregating errors together, filtering the list, and setting rules for when I should be notified using specific thresholds? And finally, how much does it cost? Most offer a free trial, but ultimately you'll end up paying by the month. When you consider all these concerns, it's easy to understand why people are increasingly paying for services rather than trying to handle this alone. And I don't recommend trying to solve this yourself. Doing JavaScript error logging well is a much harder problem than you think. Because errors are very hard to reproduce and fix without the rich metadata and filtering that these tools provide.

Demo: Error Logging

Alright, let's setup error tracking via my preferred error tracking service, TrackJS. There are many ways to handle error logging in JavaScript, but I prefer to use TrackJS because it's easy to setup, offers configurable notifications, and boasts an excellent web based interface. That said, there are many strong players in this market, so much like our conversation on testing, the important part here is to just pick one. So let's setup TrackJS to log our errors. To get started with TrackJS, you need to sign up on their website. And after you sign up, there's just two lines of code that you need to inject into your production app. The TrackJS docs suggest adding our tracking code in the head of the page to assure that it's loaded before any other JavaScript. This assures that it's loaded before any JavaScript errors occur. So let's paste this into the head of our index.HTML. And that's all it takes to get rolling. Let's start the build and try this out. And the TrackJS docs show how to throw our first error, we can just call TrackJS and tell it to track something. So let's open up the console, and paste this statement in, and if we did our job right, this error should show up in TrackJS. And there it is. We can see all the metadata about the error including the browser, time, URL, and even the telemetry of any previous activities that the user has performed. Like clicking on a button or making an Ajax call. This sort of information is really helpful for debugging issues. But there's an important tweak to make. Right now TrackJS will run in development as well, which would just add noise to our error log. So in the next clip I'll show how to use conditionals in your HTML so you can dynamically inject portions of HTML for different environments.

Demo: HTML Templates via EmbeddedJS

We now have TrackJS logging errors, but it would be nice if it only ran in production since logging errors in our development environment isn't useful and would just add noise to our error logging reports. I want to use this opportunity to show you a way to add conditional logic to your HTML so that this code is only added to index.HTML in production. So instead, let's use the templating engine support that's built in to HTML-webpack-plugin to add conditionals to our template. HTML-webpack-plugin supports a number of tip loading languages out of the box including Jade, EJS, Underscore, Handlbars, and HTML loader. And if you don't specify a loader then it defaults to embedded JS, or EJS for short. So let's just

use EJS since it's the default and it's easy to use. You can read about the EJS syntax at embeddedjs.com. And there's a handy _____ on this page so you can play around with the syntax and learn from rapid feedback that will display in this box. But for our purposes, we just need to declare a simple conditional. We want to inject the TrackJS code but only during our production build. Let's do that with a little bit of EJS. First, let's store the TrackJS token that we were just assigned on the website in our webpack config. Going to add it right here below the call to inject. Any properties that you define here within the HTML-webpack-plugin will be available within our index.HTML file. You'll see how to call this as we shift our focus over to index.HTML. And the token that I'm defining here is the token that you should've received right here when you setup TrackJS. And now let's shift our focus over to index.HTML. In here we're going to use EJS to declare that this section should only be rendered when we have a TrackJS token defined in webpack config. So let's say if HTMLWebpackPlugin.options.trackJSToken and then close our if statement, and we'll put the closing curly braces, right down here and now we can reference this variable instead of the actual token right here. Of course to reference it as a variable we need to wrap it in the angle bracket percent syntax. I'll close it out right here and be sure to wrap this in single quotes. I'll close the side bar just so we can see this better. So now, if a TrackJS token is defined within our webpack config, this section of code will be rendered into our index.HTML. Otherwise, it won't exist. And since we've only defined our TrackJS token within our production config, this section of code will only render for production. So this way, our errors are only tracked for production. Now of course, once you've added this code, your HTML file arguably isn't an HTML file anymore, it's now an EJS file. So you can consider changing the file extension to .EJS, but I prefer to keep the extension HTML so that editors will properly apply code coloring to the file contents. Let's go ahead and add a useful comment up here to the top of our HTML file, that just explains what's going on. Rather than changing the extension to EJS, I figure this comment is sufficient. And yeah it's a big comment but who cares. This will stripped out by the build process anyway. Okay, and with that set up, we should be able to Npm, run, build and make sure that this is getting injected as we expected. And if we look in the browser and view page source, we can see now that our call to TrackJS is here, and the token is getting injected into our page as expected. And with this, I think it's safe to say that we're at a point that we can confidently talk about shipping. Let's wrap up this module with a short summary.

Summary

In this final module, we wrapped up our development environment by creating a robust automated production build. We configured webpack for production and enabled minification, we generated source maps so that we can still debug the app when we're in production. And we used HTML Webpack plugin to minify our HTML and dynamically insert the necessary script references into index.HTML. We set up cache busting using HTML Webpack Plugin so that we can save bandwidth and still assure that users get the latest JavaScript when it changes. We enabled bundle splitting so users don't have to download the entire application when only part of the code changes. And we set up error logging so we know when bugs are thrown in production and we have rich metadata to help us reproduce any errors that occur. Finally, we saw how to use embedded JS for conditionally rendering portions of our HTML for production. We used EJS to assure that our error logging only runs in production. And all of this

goodness prepares us for the final module. We're headed for production. In the next module, we'll setup an automated deployment process and we'll also discuss methods for keeping your applications updated with your latest starter kit changes over time.

## Production Deploy

### Intro

Congratulations on making it to the final module, we're finally ready to discuss the last missing piece, Production Deployment. We'll begin this final module by discussing the merits of separating the user interface from your application's API, into completely separate projects. We'll briefly discuss the wide variety of cloud hosting providers. Then, we'll create an automated deployment for both the UI and the API, using popular cloud service providers. We'll wrap up the course by discussing approaches for keeping existing projects updated as your starter kit is enhanced over time. And I'll quickly provide some tips for further inspiration as you start designing your own development environment. And I'll close out the course with a short challenge. Alright, let's dig in, and wrap this up.

### Separating the UI and API

Throughout the course, we've hosted our production API in the same project as our front-end, this was useful for keeping our demo app simple, but I don't necessarily recommend doing this in a real app. Instead, I prefer to keep the front-end and the API in completely separate projects, here's why. First, a static front-end is easy to deploy, you just need to upload the static files that we wrote to the dist folder, to a public web server. And since you're only uploading the UI, you don't have to worry about regressing your service layer in any way, they're completely separate. Second, it separates concerns, it provides you the ability to have separate teams building the front and the back-end in parallel, without stepping on each other's toes. The UI team can code against the mock API we set up, and when the real API is ready, it can point there instead. Assuming that you have separate people focused on the UI and the API, your application becomes easier to understand, because a given developer can focus on one of these two concerns, in isolation. This clear separation also means that you can scale the back-end separately, this is especially useful when you create an API that's going to be consumed by multiple applications, since the traffic for the API may dramatically differ from the UI's traffic. A static front-end is also cheap to host, when your front-end is just static HTML, JavaScript and CSS, you can select virtually any host in the world, because all you need, is a host that can serve static files. In fact, a static front-end can be served from a content delivery network, also known as a CDN for short. Content delivery networks handle caching and scalability for you, a CDN is especially useful for high traffic sites, and applications that are used around the world, since CDNs intelligently serve assets from the closest physical server, to speed downloads. Finally, hosting separately of course means that you're free to use whatever technology that you like for the back-end. Our API is currently built using JavaScript, but if your team prefers to build APIs in a different language like C Sharp, Java, Ruby and so on, keeping the UI

and API separate gives you that option. And, now that I've set the stage for separating the UI and the API, let's talk about automated deployments.

## Cloud Hosting

Of course, the first question when considering an automated deployment is, where should we host the app? Virtually any cloud host can serve a JavaScript app these days with minimal configuration, popular services include Amazon Web Services, Azure, Heroku, Firebase, Google Cloud Platform, Netlify, GitHub Pages, and Surge. Most of these services offer the power to do far more than just host a JavaScript app, although Netlify, GitHub Pages, and Surge are unique options on this list, because they're focused solely on serving static files. The process for automated deployment will differ by hosting provider. For this module, I'm going to host the API and user interface separately. For the API, I'll use Heroku, because it's popular, powerful, and has an elegant automated deployment process. For the user interface, I'll use Surge, because it's simple to set up, and unlike many of these other options, it's focused solely on hosting static files, which is what our automated build for the UI generates.

## Demo: Automated API Deploy via Heroku

Alright, back to the editor, let's set up an automated deployment of our API to Heroku. We're going to create a completely separate project for handling this, to show how we can host and manage our UI and API separately. As I just discussed, there are many benefits to completely separating your UI and API projects. In our example app, we created an API endpoint, hosted via Express, so, we need to select a Node-friendly host for the API. Let's host our API on Heroku. Heroku offers a really slick setup for automated deployments that integrates with GitHub, and it offers a free option that's perfect for showcasing an automated deployment. Heroku's docs already do an excellent job of walking you through setting up an account, and creating a new Node.js project. If you go their docs here at devcenter.heroku.com, and click on Node.js, and then, click on Getting Started on Heroku with Node.js. So, if you want to follow along with me, please pause this video and go through the Introduction and Set up steps on the Node.js Getting Started page, then, come back here to continue. OK, for the rest of this clip, I'm going to assume that you signed up for Heroku, and walked through the Introduction and Set up steps for Node.js. On step three of their Set up process, which is called Prepare the app, Heroku provides a link to an example app that you can clone. However, instead of using this, I created a separate starter kit for you, that will work well with Heroku. This repository contains a slightly modified version of Heroku's starter kit, that includes our API, so this should help you get started quickly. If you want to use this repository to follow along, just click Fork up here to fork the repository. This will make sure that you have a completely separate copy that you can work with, so that you have the proper rights to work with it in Heroku. Now, I already have this repository pulled down on my local machine, so, let's jump back to VS Code, and walk through it. As usual, be sure to run npm install after forking the API repo. There are only five files in this repository so let's review each. First, package.json contains only two dependencies, Express and CORS. We'll use CORS to enable cross-origin calls, since we'll be calling

our Heroku-based API, from a different domain. Make sure that the repository field down here points to your repository, if you create your own. And, there's only one npm script necessary, which starts the app. index.js should look quite familiar to you, it's a slightly modified version of the dist server that we created in the previous module. To keep things simple, I'm using the common JS style up here on line one, to require Express, since that's the syntax that Node understands. I'm also referencing the CORS package, which we're enabling down here, to ensure that we can call the Heroku-based API from our UI, which will be hosted on a different URL. To clarify it, this is necessary because cross-origin resource sharing must be enabled to make Ajax calls to a different domain. We could of course transpile and bundle our code in this project, but, I'm keeping this project as simple as possible, so that you can see how to work with Heroku. If you diff this file with the dist server that we set up in the previous module, you'll also notice that we're calling open to open a browser there. However, this just starts up Express, and displays a message to the console. So, to try this out, you'll need to open the URL in your browser manually. I also left out Gzip compression, again, just to try to keep this as simple as possible. OK, this leaves us with two new files, that help us configure our app for Heroku. The first is app.json, which describes our app to Heroku. There are many potential properties that we can define here, but we're going to keep our app.json simple, we'll just define the name, a description, the repository where our project can be found, and then a few keywords. The other new file here is Procfile, the Procfile declares the command that Heroku should run. That's why there's just one line here, we're telling Heroku to run Node on our index.js file. And, this is all that Heroku needs to host our Node and Express-based API. Now, of course, I'm deliberately leaving out all the complexities of testing, transpiling, linting, bundling, and more in this project, so that you can clearly see what it takes to create an automated deployment to Heroku. But of course you can feel free to start adding those in, once you're comfortable with hosting on Heroku. So now we should be ready to complete an automated deploy to Heroku, I've already signed up for Heroku and installed the Heroku CLI. So now, let's open up the terminal, and type heroku login. At this point you'll be prompted for your email, and your Heroku password, and after entering your credentials you should see your email listed in blue, which shows that you are logged in successfully. And note that you may receive some warnings about file permissions, so, consider tightening file permissions for security, if you like. Now it's time to configure our app to work with Heroku, so we can type heroku create, this will prepare Heroku to receive our app. This command returns a URL, and if we load it, we can see a welcome message. Heroku generates a random name for your app, or you can pass a parameter to specify your own name. Now that we've created our app, we need to configure a git remote that points to our Heroku application. So, let's go back to the command line, and we'll say heroku git:remote -a, and then pass it the name of the app that we were assigned right up here. And now that we've set the git remote, we should be ready to publish the app. We can say git push heroku master. We can see the deployment output as it builds the source and pushes our app out to Heroku. It displays the random URL where our app is now hosted, so, you'll have a different URL than me, if you're following along. And of course, for a real app you'll want to specify a domain name that you've registered. But now, we should be able to take this URL, and go over to the browser, and when I load it up, there we go, we can see our Hello World! And if I go to \users, I can see the JSON coming back for our users, so we have our API now hosted in production. And any time that we make changes to our API, we'll just commit our changes, and then run git push heroku master, to be able to push our changes up to Heroku. Heroku will take the code from GitHub, and deploy it to our URL, slick. And, now that we

have our API running in production via Heroku, let's jump over to our UI project, and update it, so that it will hit our Heroku-hosted API. To do that, let's open up baseUrl.js. Note that right now, we are either using the mock API which is hosted at 3001, or we were assuming that we were hosting Express locally, for production. Now instead, we have a new URL to use for production. So, I'll just paste in the Heroku URL, that I was assigned, in the previous step. And, make sure that you include the trailing slash on the end, and now we can also open up distServer.js, and we can remove this section because we're going to be hitting Heroku, instead of local, when we do our production build. This way our production build is more realistic, we know that our production API will be hosted on Heroku, and we're going to host our UI on a separate service here in a moment. But now when we do a production build of our UI, it will hit our production API hosted on Heroku. In the next clip, let's set up an automated deployment for the user interface, so that we can see all this work together.

Demo: Automated UI Deploy via Surge

Okay, now it's time to code our automated UI deployment. Here's our goal for the process that we're going to set up for the front-end. It's a three-step process to get code into production. First, we run npm start to do our development. Once we're done coding, we type npm run build, to build the application in production mode. This opens the application's production build on our local machine, so that we can verify that everything looks good. If we're happy, then it's time to deploy to production, so we can type npm run deploy, this will automatically push the final built application up to our production host. Of course, we already set up steps one and two in the previous modules, so now, it's time to focus on this final step. Alright, it's time for our final coding session in this course, but this one is critical, it takes our front-end public. To do so, we'll host our static front-end on Surge, let's make it happen. I'm a big fan of Surge, because it's a low friction way to deploy a static front-end, and for all the reasons that I mentioned earlier, I strive to build static front-ends. Getting started with Surge couldn't be easier, typically you'd install it globally, using npm, but, we don't need to, because we already installed it at the beginning of the course, since it's listed in our package.json right down here. And, we also don't need to install it globally, since we're going to run it via an npm script. Remember, Node's bin folder is added to the path automatically, for npm scripts. So, to set up Surge, I'm going to add one whopping line of code here, in package.json. Yes, it's seriously that easy, and that's why I love Surge. Now, first of course, we need to call npm run build, so that we have something to push out to production. And when our app starts up in production mode, we can see the data coming back as expected. If we come in and Inspect the Network, we should see, when we refresh, we can see that we're making a call to Heroku, as expected. And with this set up, we can now hit Control + C, and type npm run deploy. We can see Surge gets called, it assigns a random domain and when we hit Enter it says success, and now we know that our app is up in production, at this random URL. So if I open a new tab and load it, there we go, we can see our app loading in production and using the Heroku API for the data, success! Of course, the Delete link won't work, since we never added that functionality to our Heroku API, but if you want a challenge, you could certainly add a database behind the scenes, to support adds and deletes. If we open the browser tools, and go to the Network tab, we can see that Surge serves our assets using Gzip, by default. And of course, you'll likely want to set up a custom domain, and Surge will let you use your own domain

for free, or if you just want to request a subdomain, you can specify it, via a command line flag. There are quite a few nice tweaks that you can make, but you get the idea. We should strive to build static front-ends, and if you do, Surge is awesome. Of course, your starter kit is likely to change over time, so, in the next clip, let's talk about how to keep existing projects updated as our starter kit changes.

## Starter Kit Update Approaches

Now, once you've created your team's starter kit, how do you keep existing projects updated over time, as you enhance your development environment down the road? Let's review a few approaches. Let me first run through a common scenario to help clarify the problem that we're discussing. Imagine that your team watches this course and creates a development environment that works great. In the first quarter, you launch your first project, using your new starter kit. Then, in the second quarter, you launch another project successfully, using the same starter kit. In quarter three, you learn some lessons, upgrade some libraries, and tweak your starter kit with various enhancements and bug fixes. The question is, how do you easily get these enhancements into the products that you launched earlier this year? Of course, one way is to simply manually update these previous projects by hand, and that works, but we're developers, so let's talk about some more automated approaches. Here are three more automated ways to handle updates to starter kits over time, Yeoman, GitHub, and npm. In the next few clips, let's discuss each of these approaches in more detail.

## Option 1: Yeoman

Yeoman is a handy scaffolding tool for starting new projects, so, once you're happy with your development environment, you can create a Yeoman generator. This makes it easy for people to create new projects by typing yo, and then the name of your generator. Yeoman hosts a long list of generators that will give you a headstart on some of what we've covered in this course. Few will cover all of the features that we just implemented but it's another great place to check for inspiration, or, as a good starting point for your framework or library. Assuming that you've created a Yeoman generator for your development environment, it's a three-step process to update an existing project later. First, be sure that you've committed all your code to your source control system. Then, rerun the generator on your existing project, Yeoman will prompt you for each file that's being overwritten. Then, you can diff the files, and manually resolve any conflicts that occur. Of course there's much more to know about Yeoman so, if you want to learn more, check out the Yeoman Fundamentals course by Steve Michelotti.

## Option 2: Github

Another approach for updating existing projects is to use GitHub. With this process, you begin by hosting your project of course on GitHub, and then you fork the starter kit, when you start any new projects. This way, you can pull changes from master, as the starter kit is enhanced over time.

Option 3: npm

Another approach is to wrap your starter kit in an npm package. With this approach, you abstract away the configuration and build scripts behind an npm package, and then you update the npm package to receive the latest changes. This approach has the advantage of abstracting complexity away, and it's also the simplest update, since you don't need to resolve conflicts like the other two approaches. However, this advantage also has an obvious downside, you're restricted from tweaking anything inside the npm package, for a given project. So, this approach is great if you want to programmatically enforce that all projects use the exact same config, but, some may find this approach overly restrictive. Let's talk about this approach in a little more detail, because it's becoming increasingly popular. Depending on the complexity of your starter kit, updating an existing project manually isn't much work, so you might consider this hybrid approach. Let's walk through the files that are in our demo starter kit. The most significant piece is in buildScripts. This is the easiest thing to centralize, you can move all of buildScripts to an npm package. The other big piece is package.json. There are two sources of complexity here, the scripts, and the dependencies. For the scripts, you can streamline your npm scripts to just call buildScripts instead. Since you can put all your buildScripts in an npm package, this means that your scripts in package.json are nothing but a list of function calls to your separate scripts. This effectively centralizes your buildScripts, allowing for bug fixes and tweaks in the future. Webpacks config is just a chunk of JSON, so it need not be stored in a webpack.config file. Instead, you can move it into a buildScripts npm package as well, so that it's centralized. The ESLint configuration can be centralized by creating your own preset. This way, each project can define its own .eslintrc, but use the preset that's stored in npm, as the baseline. So this covers most of the moving parts in our starter kit, so what's left? Well, the approaches I outlined on the previous slide, centralized all the items on the left using npm. That's a significant win, since it's the vast majority of the starter kit's code. So, what files would we still have to update manually? Well, editorconfig, which is unlikely to change much over time, babelrc which contains very little code, and the Continuous Integration server configuration, which is also unlikely to change over time. The final piece that we didn't centralize on the previous slide, is the package references in package.json, but, these are easy to update with existing npm tooling, so again, not necessarily that big of a deal. I'm a fan of this hybrid approach, it provides most of the benefits of centralization, without the cost of creating and maintaining a Yeoman generator. And it gives us a lot of flexibility as well, since we can decide what's worth centralizing, and what isn't. So that's three different ways to keep your projects updated over time. Let's close out the course by discussing some sources for inspiration, as you move forward on creating your own starter kit.

Inspiration

In this course, you saw a massive list of choices, and I shared my recommendations, but there are literally dozens of other ways to build a JavaScript development environment. So let me share some sources for further inspiration. If you work in React, check out Andrew Farmer's excellent list of React

boilerplates and starter kits. Andrew has catalogued over 100 starter kits into a curated, filterable list. My starter kit, React Slingshot, is just one of many in this list, and I walked through how to build it in module two of my React and Redux in ES6 course. If you work in Angular, check out the Awesome AngularJS list. AngularJS developers often call their starter kits seed projects, so check out the full list at this address. Of course there are many more popular JavaScript frameworks and libraries out there, so if you're looking for inspiration, just type the name of your favorite JavaScript framework into Google, along with one of these terms, development environment, boilerplate, starter kit, starter project, or seed. The point is, I've just shown you one way of getting this done, so, start searching. Let's wrap up the course now, with my final challenge.

## Challenge

I like to end my courses with a relevant challenge, and, this one is no different. My challenge to you is easy, just send a meeting invitation to your team. Why? Because the first step in this process is to start a conversation. In this meeting, you need to answer some questions. First, would you benefit from a starter kit? If you expect to start any new projects using the same technology stack in the near future, then it's worth discussing creating a starter kit. And regardless of your answer to this first question, the next question is, what pain points do we have in JavaScript today? Are we struggling with testing, broken builds, time consuming, manual, or faulty deployments, inconsistent coding styles? We just discussed a long list of options for solving all of these pain points. Would we benefit from a demo application? Are people who join our team clear about how we operate? Do they understand our opinions on directory structures, file naming, API calls, and more? A demo app clearly conveys many of these decisions. These questions should help provide your team with a clear vision on how to move forward. So hey, this is a pretty easy challenge, just send an email and get the meeting scheduled.

## Summary

And that's a wrap! In this final module, we began by discussing why separating the UI and the API can be useful, including the ability to deploy the UI and the API alone, separation of concerns, which allows separate teams to manage each of these, the ability to select cheap hosts that handle static assets, and, the flexibility to select whatever API technology that you like. We reviewed a list of potential cloud hosting providers, but ultimately created an automated deployment using Heroku to host the API, and Surge to host the UI. We discussed approaches for keeping your projects updated with bug fixes and enhancements, as your starter kit improves over time, including Yeoman, GitHub, and npm. And I quickly mentioned a few resources for inspiration, including some terms that you can use, to help you search for starter kits that are specific to your preferred technology stack. And I wrapped up with my very simple challenge, set up a meeting with your team, to discuss the path forward. And that's a wrap! Please share your links to your starter kit in the course discussion, I'm excited to see what you create. Thanks so much for watching.