Introduction

Introduction

Welcome to this course on using npm as a build tool. My name is Marcus Hammarberg, and I'm very excited to present this course. Using npm as a build tool has simplified and sped up my development considerably. Also, I've had more fun in doing so. I hope that you will feel the same after the course. I remember when I first started to write applications. We were coding away on our VB 6.0 application and after about six months we built it into an installer and we shipped it on a floppy disk and we waited for the bug reports to pour in. The tempo in that process has gone up considerably since then, and now many companies release their software several and sometimes many times per day. At the core of this is a fast build pipeline that transforms your application to a deployable unit. For modern JavaScript and web development, that includes a variety of tools, steps, and processes to compile, test, minify, bundle, and deploy your application. This has led to the development of several tools like Gulp, Grunt, Bower, to mention a few. In some cases, even full ecosystem of subtools to handle the complexity that a build process can encompass. While most of these tools are really great and worth looking into, I have come to realize that for many applications that I'm writing, you don't have to use an external tool, you can use what Node itself is written with, npm and the package.json file. But really, why should I even care about this when there is an abundance of great tools already? Well, first of all, npm is baked into your process already when using Node. No more tools are needed. Secondly, any command that you run with a Command Prompt to today can be moved into your package.json file with one line of code. Thirdly, the Node Package Library that you most likely are using anyway is vast, and contains just about anything you need for your automation. Fourth, keeping your scripts in npm and the package.json files means that everyone can use it, no additional configuration is needed. Just npm install and then use the scripts. Related to this is that you now only have one place to keep updated. For many of the build tools, there's a lot of plug-ins, each separately updated in the configuration of the build tool. Basically, I would recommend that you just use npm and the package.json file; it's simpler and faster. Npm, the Node Package Manager, is a tool that is very powerful indeed. In fact, this entire course will be just around a couple of its many, many features. It's also through npm that we can see the real reason Node has taken off and established itself as a leading web platform so quickly. I'm talking, of course, about the many Open Source libraries and packages that we can install and use with npm. The package.json file is at the heart of every Node application, since it's a manifest file that describes our application, our dependencies, the version, and other features of our application. With only these native tools to our disposal, we will throughout this course create a series of scripts that we will put together to a fully functional build tool. A small disclaimer though, the ecosystem of Node is vast, so I can almost guarantee that I will miss your favorite tool, or one of them at least. My ambition is, of course, not to cover all of them, but I will make sure that you get the principles so you can plug in your favorite test reporting minifier tool with no hassle. Enough talking. Let's get started right away. You will be happy to know that this course is very heavy on the code and light on the slides.

Our Example: npm init

In order to demonstrate the things we're going to work on during the course, we need an example. Let's create one by using, you guessed it, npm, the tool of choice for the entire course. By the way, should I be a little bit sloppy or forget something that you think is important, or if you want to dive deeper, the documentation on npm is about as good as I've ever seen one. It's at docs.npmjs.com. Also, for this course, I'm using Node 4.0. That's a version leap for the books, by the way, 0.12 to 4.0, and my npm version is 2.14.2. You can see what you are on by running Node --version or npm --version. Installing Node and npm is well-described on the Node.js.org homepage. Finally, I'm on a Mac, but everything I will show you works fine on Windows, too. If there are things that are different, I will make sure to point them out. Now, let's code. All but the very trivial Node application starts with the package.json file and ours will, too. Let's use npm to create it for us. Npm init will run through a little Wizard at the Command Prompt. Let's do that and set up a little demo application. We will create a stupid little website and I'm running npm init in a directory called stupidlittlewebsite. The only tricky part of that Wizard is really the test command. Let's skip that for now and just fill out some sensible values for the rest. When you have created the package.json file, it looks something like this. There's a few things of importance to point out in this file. The name is what defines our application. Should you want to publish it on npmjs.org, the name has to be unique. The version field shows which version we're building right now. The description should probably say something about what the application does so we understand it. The main file of our application, this is the main entry point of our application. Author is you, or, well, me in this case, and license should also be set to something sensible. Not setting a license is actually more strict than choosing the MIT license, for example. The last node is the one where we will spend most of our time in this course, the scripts node. Here we can define any arbitrary number of scripts that we can use npm to execute, and that's what we're going to deep dive in the next module. But look at us, we're using npm already. This is easy.

Summary

This course is structured into six short modules, excluding the introduction and summary modules. This is the first one, introduction, where we just introduce the tools and give an overview of what is about to come, and that's soon over. In the second module, we'll get started scripting. We'll also extend the capabilities of our script by using packages from npm. It will be very easy in the beginning, so don't worry. It will be fun, too. In the third module, we will look into some of the utilities of npm and the package.json file and start to use hooks in our script. The fourth module is a little more case-based, and here we will build some scripts that can be handy when you develop and test your code. The entire module five is dedicated to task that watch your folders and files and run on changes, both for the server, the client, but also for reloading our browsers. It's a packed module, I promise you. In the sixth module, we'll take a look at some of the scripts that we can use when we build and deploy our code, and here we will call npm from npm, a little inception action going on there, and it will be crazy. Module seven puts everything together, and we will create one big task that runs many of the tasks we have created so far in sequence. I will also show you a couple of tips on how to refactor out big scripts into separate files. And finally, I will wrap the course with a short summary. Speaking of summary, let's summarize the introduction. Npm is a great option for use for JavaScript build automation. There's no

external dependencies. You are using it anyway as you build your application. This is how Node itself is built, and it's just JavaScript, the lowest common denominator for all tools and languages on the Node stack. Oh, yeah, it's also fast and fun. Let's get going scripting with npm.

Getting Started

Introduction

As I said, we will be spending most of our time in this course in the scripts node. And look, the Wizard has already created one for us with the test script. Right now it looks like we will get some kind of error message, and then exit with a 1. I want to work that rest. Let's try it out. Npm can be used to execute the scripts on the scripts node. The syntax is simple, npm run-script and the name of the script. In this case, that will be npm run-script test for the test script. Whoa, that threw a long-winding error message and created a log file. This is because the exit 1 part of the command, which is the way that npm script signals that something has gone bad. Let's take that part out. And now we run it again, npm run-script test. Well, we get a little log from npm telling us about the script that it ran, and then simply printed Error: no test specified. In fact, that is a bit long, npm run-script test, and there's a shortcut for running scripts that is just npm run and then the script name, which is the one that most people use. Speaking of shortcuts, the test script is one of a few well-known scripts. It actually has several shortcuts defined for it. You can write npm test only, or npm tst, or npm t even. See? They all run the same thing again, they run the test script. As I said, there are a few scripts that are so common that npm have special shortcuts for them. Another one is npm run start or npm start for short. Ah, when we run that, we see that we have no start script from the error message. Let's fix that by adding a new script called start. Now when we run npm start, we see the message Running start script. The start and test script are so common that many platforms and tools make use of npm start or npm test. For example, Heroku will execute your application by running npm start. So, whatever you put in here as a script is what will be used to start your application. The test script is, of course, used to run your automated test. This might be run by continuous integration platforms, like Jenkins. There are two more well-known commands, as in npm have shortcuts for them, the restart and the stop command. The stop command simply runs your stop script, and the restart, well, it's not so simple. If you have a restart script, then that is run, but if not, npm is smart enough to first run the start script and then the stop script, or only one of them if only one is defined. To prove a point, let's add a restart script, and run npm restart, and now only the restart script is run. The natural next step is, of course, to add your own custom script. It's about as easy as you would think. Let's create a script called start:dev, which could be used to start our application with development settings, for example. Adding the script is very simple. Just add a new scripts line in the scripts node. Don't forget the little comma at the end on the line before the script you just added, that gets me every time. Running the custom script is equally easy. npm run to run scripts, or npm run-script if you want to type more, and then the name of the script, npm run start:dev, in our case. There are no shortcuts like npm start:dev or something like that, that's only for the well-known scripts we already mentioned.

Using Packages in Scripts

Until now, we have just been using simple commands like echo. This is not often sufficient, and pretty soon we need to use other tools to get our work done. Luckily, Node has one of the largest Open Source libraries of packages in the world to our disposal with the npm package library. We can use all of those tools as part of our scripts. One of the first things that you want to do probably is to run your tests, so let's do that with a test runner called Mocha. Just to spice our Mocha up a bit, we'll use another tool for assertions called Should. These tools are installed as usual with npm install, but since these are development dependencies, you only need to use these tools if you are developing in our application. We save them in the package.json file under dev dependencies. The full command to do so is npm install mocha should --save-dev. That command installs the package in our local node_modules folder, and updates our package.json file. Also, it makes the mocha command available for us to run in our scripts. Let's add a test folder and a test.js file with a trivial test. Now we can run it and get some pretty formatting by using the mocha command with the following switches. But writing that every time will get tiring fast, so let's move it into our package.json file as our test script. Now we can run our test scripts with all those switches by going npm t or any of the other shortcuts I told you about. In this way, we can call out to external tools and packages and make use of them in our scripts. Let me show you a neat little trick that can be very helpful. When we npm install a tool, it's installed into the node_modules folder. In that node_modules folder, there's a hidden folder called .bin. Things that are installed in the .bin folder can be reached by our script by simply using their name, like mocha as in our example. A very simple way to find out what commands we have available is just to list the content of that .bin folder, like this, for example, ls node_modules/.bin. In our case, that just lists the mocha tool just as expected. As we get more and more packages that we are using while scripting, this trick will become very handy to see what packages we have to our disposal.

Our App: Start and Test Commands

Let's put together what we learned so far and create something a little bit more real. At the end of the course, we have created a pretty advanced build pipeline. Let's continue on the skeleton application npm, and it's generated for us in the last module. For this very simple website, I'm using the simplest web framework I know, Koa. There's a decent course on that here on Pluralsight. I use Koa since it's super small and stays out of our way to a large extent. Install Koa and save it as a dependency for our application, npm install koa --save. Now create index.js with the simplest application possible, fits into it. This simple Hello World application responds on every request of this port by just returning Koa says Hi. Start this application from the Command Prompt by going node and the name of the file index.js. Now we can test it out at localhost:3000. Now let's move that command to the start script. And we start it again with npm start. Most likely in the future, we want to start this application on different ports; let's add some code for that in our application. Simply add something that checks if there's a process port set or if there's an argument or we use port 3000. Now we start the application in development

environment by going node index and the port number, 4000, for example. That works fine. Let's move it to a script. We call the script start:dev. Let's try that out, npm run start:dev. It works fine. Any serious development in JavaScript needs to have tests. I've created a very simple test that verifies that our application works as intended. It's using a tool called supertest to access the web application. Install the tool we need as our development dependency, mocha, should, and supertest. Don't forget the --save. Create a test directory and add a site.spec.js test file into that directory. Add the following simple test. It uses supertest's agent to listen to the application, and it just verifies that we actually get the message that we're sending out. To get this to work with supertest and Koa, we need to export the app object from our application, so it's visible outside the index.js module, like this. Run the test by using mocha test, the directory we want to test, -u bdd -R spec for the pretty reporting. That works. Copy that command to a test script. Now try npm t to run the test command with a shortcut. It works.

Summary

Congratulations! You've just started doing scripting with npm and package.json. Let's summarize. Scripts are stored under the scripts node. You can start them with npm run and the script-name. You can install packages to use from your script by using the npm install command as normal. Remember to store them as development or runtime dependencies using the correct flag. Some scripts, like start and test, are so common that they have shortcuts directly from npm. That's it for now. In the next module we'll continue by using a feature that helps us to organize our scripts a little bit, pre and post hooks. See you there.

Pre and Post Hooks

Introduction

As we start to use more and more scripts, we will soon find the need of structuring the execution of our scripts a bit. Luckily, npm has a few tools built in to do so. In this module, we will specifically check in on the pre and post hooks that help us to organize our scripts a bit.

Pre and Post Hooks

All scripts in our package.json file could have associated pre and post scripts. For example, for a test there could be a pretest script that runs before the test script, and subsequently a posttest that runs after. Let's try it out. Here's a minimal package.json file I've created. It only contains a dummy test script. Let's run it just to make sure that it works, npm tst. Yes, it outputs the text. A hook is just another script with the same name as the main script, prefixed by pre or post. Let's add a pretest script. Now when we run the npm test script, the pretest script is run before it. We can see this in the output. By the way, that is a bit verbose. Let's silence it down a bit by adding that -s flag to npm command, npm tst -s. There, that's the two output statements. Let's add a posttest script, too. Running npm t -s gives us the

expected output. A few points here, first of all, these pre and post scripts are like any other script. We can npm run pretest to only run the pretest script. Secondly, more obvious, but still, the order in the package.json file does, of course, not matter.

Hooks for Custom Tasks

Thirdly, and this is a bit less obviously maybe, pre and post hooks works for our own custom scripts as well. Let's create the new script greet that just says Hello. We can now create hooks before and after the greet script by just adding pregreet and postgreet hooks, like this. Now when we run the greet task, npm run greet, we see that the pre and post scripts are run, too. This is a very powerful feature that we will make use of when we build something a little bit more complex.

Our Example: Linting

Let's get real and put this in use in demonstration application and add a task for linting. There is a tool that helps you write better JavaScript. It's called JSLint, and has been known to make developers cry. JavaScript Linter is very strict about what is good JavaScript, but the rules are really good and the best time to start to follow them is right in the beginning of your development task. I view this as a compilation check step. Linting can be done by a Node package called jshint. There are many others, too. So, let's install first, saving the package as a development time dependency; npm install jshint --save-dev. We can now create a script that runs linting on all our JS files in the current application directory and recursively below. It is such a script. Let's run it, npm run lint. Told you! Loads of warnings and an exit code of 1. The exit code will be useful for us later. The main issue for jshint here is that we are using generators and yield, so let's tell jshint about that. This can be done in a separate file called .jshintrc, or a configuration right in the package.json file. Let's do that for now. This tells jshint that we are using ECMAScript 6 generators and that we allow the generators to not have a yield. We re-run npm run lint. Huh, only one error. We're missing semicolon in our test file. Let's catch it early and fix that. Now that's fixed and linting runs without problem. We now have our code linted and clean. Now, all we have to do is to remember to run npm run lint very, very often; for example, before every test run. This is, in other words, a perfect fit for the pretest hook. Let's hook it up. Notice that we are using npm to start another script in the package.json file. And then run the npm test command, and sure enough, it first lints our code and then runs the tests. Let's add a linting error by removing a semicolon again. Now when we run the tests, we get the linting error and the tests are not run. This is because jshint exits with an error code of 1 if a lint error happens. This is great news. Yay, an error! We are now halted with a small error before we can proceed to run our tests. I'd rather have small errors often than big errors seldom.

Summary

By using pre and post hooks, we can structure the execution of our scripts to always do things in a certain order. The pre and post hooks are just ordinary scripts in the package.json file that just have those prefixes. They run before and after the script with the same name. Premyscript runs before myscript, postmyscript runs after myscript. You can execute scripts from within another script in the package.json file by using npm run and the name of the script. That's it for now. Now you know the basics of scripting with npm, so in the next module, we will be a little more task-oriented and start to build scripts that will be useful during the development of our application.

Scripts for Development and Test

Introduction

Starting from this module and a couple of modules forward, we will change gears a little bit. So far, we learned about the basics and syntax and functionality of scripting with npm. We will now use those acquired skills to support us in different stages of the development cycle. In this module, we will create a few scripts that can be useful when we develop and test our application. I'll make sure to sneak in some tips and tricks in the progress, too.

Compile Coffee-script

There's a number of different languages that compiles from its own syntax into common JavaScript. Two examples are CoffeeScript and TypeScript. Let's use CoffeeScript as our first example. I've created a src folder with some CoffeeScript code in the coffeescripts folder. It's super simple stuff, but shows the principle. Here's the CoffeeScript code. As you can see, it's a simple function just returning a string with the variables injected. In order to use that code from our application, we need to compile the CoffeeScript to JavaScript. Let's write a test to test the functionality once it's completed. My plan is to compile the CoffeeScript code in the src coffeescripts folder to the lib folder and use it from the test from the lib folder. The test just used the fill function, passing the strings mug and coffee, which should equal the total string, Filling the mug with coffee. When we run the test, npm t -s, it fails, of course, since no such file exists yet. Let's fix that. First we need to install the CoffeeScript compiler. That's easily done with npm install coffee-script --save-dev. This is a typical development dependency, so don't forget the --save-dev flag. That now exposes a new command for us, coffee. Remember that we can list that with ls node_modules/.bin. Let's create a task to compile CoffeeScripts in the coffeescripts folder to JavaScript in the lib folder. We're passing the --compile to perform the compilation, and --output to indicate where we want the files, and the location of the coffeescripts files is at the end of the command. I've chosen to prefix the script with compile and colon, since that makes a nice grouping of scripts should we have many compiled scripts, for example. Let's try that new script, npm run compile:coffee. That worked just fine, and if we check the lib folder, a coffeeCode.js file has been created for us. Let's run our test to see if it works, npm t -s. Ah, that gives us error in linting. Remember that our lint script runs pretest, before our test. The linting on the generated code fails. Let's exclude the

lib folder from linting. Here's how that new script looks. Okay, now when we run npm t -s, the test runs just fine.

Compile TypeScript

Let's do some TypeScript too, just for good measure. This will be very similar, so I'll speed it up a little bit. Write some simple proof of concept code in TypeScript, locate it in the src typescripts tsCode.ts file. Here we see it's just a simple class using a greet function, passing a parameter name. And don't forget to export the greeter, so we can reach it from other code. Now we write a test to use the compiled JavaScript. And we're simply using the greet function, and make sure that we get the expected output. Npm t -s runs the test, but fails, since there's no compiled code. So we install the TypeScript compiler with npm install typescript --save-dev. Write a new compile:ts task that compiles the TypeScript to the lib folder. Here's how that task looks. We set that output directory with a --outDir, and we say that we want the module to be commonjs. And the code is found in the file that we indicated. Now when we run npm run compile:ts, it compiles, and npm t -s runs the test and it works just fine.

Refactoring

Okay, that works just fine, but now we have to remember to run two scripts for all our compilation. We can do better than that. Let's create a new task, compile, that runs all of our compiling. In our case, the two scripts we created after each other, npm run compile:coffee && npm run compile:ts. As we saw before, you can run other scripts in your package.json file by simply using the npm run command. You can combine several commands at the Command Prompt by using &&. This means that if the first command is run and exit with something else than 0, the second command is not run. Should the first fail, we stop, just what we want. We don't want to continue our compilation if one compilation step fails. You can combine commands also with a semicolon, that means that you're ignoring the exit code and run both commands no matter what. Sometimes that is what we want. Great, now we have a nice little script that does all our compilation for us. Let's test it with npm run compile. It works. But how can we make sure that the changes we make are actually compiled so we don't accidentally use old stuff? Let's, just to be sure, clean our output lib folder before we run the compilation. We create a pre-compile task in which we clean out the lib folder. Here I could use a Linux OSX command like rm -rf lib, but that doesn't really work on Windows, so instead, let's use node and a package called rimraf that does the same thing. Install it with npm install, and add the following script. Call it clean and just point rimraf to the lib folder. And then we call that script from the precompile hook, like this. Now we have a really sweet thing going here. We run the compile task, which in turn runs the precompile task, which runs the clean script. We can, of course, just run the clean script separately if we want to. Should we really want to, we could call the compile script before our tests in our pretest script to ensure that we are running our test against the compiled linted code, like this, npm run compile && npm run lint. Now when we run the test, the cleaning, compiling, and linting is run before our testing. Sweet!

Less Compilation

Let's turn our heads to the client a little bit. Full disclosure, I'm no front-end guy, so I don't know a lot about this, but front-end development of today have many processing compile-like steps in development, too, just like what we did on the server-side. Let's take two examples and you can most likely elaborate on these to suit your own needs. For our examples, let's compile Less to CSS and then do some bundling and minification. First, Less compilation. I find CSS perplexing, to say the least, but there's help. One of the tools that can help us with the horrors that many of us face when using CSS is called Less. Nice to have things such as variables, inheritance, and nesting, yeah, CSS is really lacking some basic features. But browsers still just understand CSS, so we need to compile the less code to CSS. Well, you could actually use your browser to read the less code directly, but that's another course. I've created a client/less/demo.less file with some dummy less code. You can check it out on your own time; I will not go through it here. The idea is that we write our code in the nice less language and compile it to CSS in our sites assets directory. Let's call it public/css/site.css. To do this compilation, we're using yet another Node package, simply called less. Let's install that, too, and save it to our development dependencies. In doing so, we now have a new command line interface to our help, as we can see with the ls node_modules/.bin command. Note that the less client is called lessc. To perform the compilation from and to the correct files, we can use this command. Let's move into our package.json. I prefixed these client compilation steps with build just to separate them from the compiling happening on the server. Let's try that script out, npm run build:less, and now if we check the output folder, a css file has been created.

Bundling with Browserify

The JavaScript that is running on the client of course needs to be sent to the client to run. We want to send a few small files down, hence the practice of bundling and minifying has become staple practice in the front-end development of today. Bundling basically means that you're taking all our client-side JavaScript files and put them together into one file. Minifying means, well, you get that, of course. Make it smaller, uglier to read even, but smaller. There's a plethora of tools and ways to do this, and I hope you forgive me for showing only one example here. I've created an example and it will be in the client/js folder. I have two folders here in the folder hierarchy, and we want these to be bundled into one and put into the public/js folder. Let's use a tool called Browserify to bundle our client files. Browserify is really cool, because it gives you the opportunity to use require statements to include functionalities from other files, just like we do in Node, but on the client side. Browserify will use these require statements when it figures out how to bundle the files together. Here's my super simple example. I have an app.js, which uses a nested folder lib.js, and that's just a simple Hello World. Let's try to bundle those files into one. We can install Browserify with, of course, npm install and --save-dev. After that, we can create a script called build:bundle. Let's make it look like this. This simply points Browserify to our app.js file and lets Browserify figure out how to use the require statements to bundle it into one file. The -o

switch tells Browserify to write the result to the specified location. I've already created a public/js folder. Let's try it, npm run build:bundle. Yes! It works. We can even try this with Node now, since I wrote an example so simple. Let's go to the public/js folder and just run node bundle.js, which is the file that you would include on your page; it works.

Minifying with Uglify

Yes, it works, but it's not minified. In this simplified case, this will not make much difference, but if you have many thousand lines of client JavaScript, then this next step will have a big impact for your page load time. Let's use a tool called uglify-js to minify the bundled JavaScript. There's a twist there, though, but we'll get to that later. First, install it with npm install as usual. Using the uglify-js tool is now relatively straightforward. Let's call the command build:uglify. But wait a second here, it's the bundle.js that we want to minify. And that will be recreated with our Browserify script. It's time to get a little bit more advanced in our script. We've already seen how to use the && to run scripts in sequence, but this time we want to take the output of the bundle script and send or pipe the result to the uglify task. Luckily, this is easy. Let's update the build.bundle task to do two things. This is a little bit complicated, so let's go a little bit slower. As before, we are calling our Browserify task with our app.js. This will create a bundle file just as before, but this time we have not indicated the output file, we have not used the -o switch. Doing so means that Browserify will send the output to the standard output, the terminal window, or the task that we are piping the output to. The vertical bar means take the output of the command on the left and pass it as a first argument to the command on the right. This means that we get the bundled result as an input to uglify.js. We also pass two flags, -mc, to mangle and compress the file with uglify.js. Finally, we're writing the output of that command using the redirection operator, or the greater than character. This creates or overwrites a file with the contents sent to the operator. It works in the same way on Windows. Running npm run build:bundle will produce a bundled, minified, uglified bundle.js file in the public js directory. We can again test it with node public/js/bundle.js. It works, although it's small and unreadable; just what we wanted, in fact.

Putting It All Together

Let's create the build task that does all of our client-side building for us, npm run build:less and npm run build:bundle. One could argue that the build:bundle script now is doing two things. I'll leave that as an exercise for you to refactor into separate tasks should you see it necessary. One thing that we probably should do is to clean the output folders on the client-side before the build. Just as before, I'm reusing the rimraf package and just clean the output folders. We run the clean script from the prebuild script before building. Whew, that's plenty, but now when we run npm run build, build runs the prebuild script, prebuild runs the build:clean script, build:clean cleans the output, build is then executed, build:less compiles our less files, and if that works, build:bundle is executed. Let's try to run that command as well. It works.

Summary

This module is growing out of hand. I have more things on the development test, but I'm bumping it to the next module. In this module, we learned about a lot of tools and created scripts to compile our Coffee or TypeScript code to JavaScript, clear out compilation destination using rimraf, building less code to CSS, building and minifying JavaScript for our clients. On top of that, we learned some basic npm tips and tricks. You can chain commands, running commands in sequence with &&. If the first fails, the other will not be run. We can pipe the output of one command into the next with the pipe vertical bar sign. We can write the output of one command to a file using the redirection operator. Whew, that was a lot of things in a small module. I'm a little bit tired, are you?

One More Thing...

In fact, let's take a breather before we continue to the next module. Now we have created a bunch of scripts in our package.json file. How on earth can we remember them all? I've got good news. You don't have to. If you run npm run without any more arguments, we will actually get a listing of all our scripts right in our terminal window. This is very helpful to see what scripts we have to our disposal. This is an often missed opportunity to learn about the scripts in the package.json file. Okay, feel better? Ready for the next one? Let's go.

Scripts for Development: Watching

Introduction

As JavaScript is a dynamic language, there are a number of checks that we want to have run continuously to make sure that we can proceed our development with confidence. For example, our linting and testing are two such things for the server-side. On the client, there's an additional layer of complexity. When we change a parameter in a .less file, for example, it has to be compiled to a css file, and then the browser has to be reloaded for us to see how that change looks. Let's in this module take a look on how we can continuously watch our changes and have scripts executed as the consequences of those changes, leaving us to focus on our code rather than to start scripts.

Server Side Watching

For some of the tools that we already are using, there's a watch option, or flag that we can use. One example is Mocha, the test runner. Let's write a new script to watch our test folder for changes and run the test for each change. The trick is in the -w or --watch in the end. This will watch the test folder. Let's make some room for our windows here so that we can see the code and the terminal at the same time.

If we run npm run watch:test, we see that it waits for changes, and if I make a change, the tests are rerun. That was fun, let's do it again. What? It crashes and burns with an EADDRINUSE, address is in use error. To fix this, that only happens when we watch and start our tests over and over, we will have to make sure that our web server does not start up and listen when it's under test, because the starting of the web server is done as part of the test now. Here is the code to do so. We only start to listen when there's no parent, as the case when we run it outside of testing. In testing, the test will start our application. We can shorten this up to one line, too. Okay, that was a detour. Back to our current task, watching tests. Now it works, as we can see when running npm run watch:test. We can make changes and run it over and over. You probably noticed that we copied the entire test script saved on different flags. Let's improve on that a little bit. Now we are calling into the test script. Did you notice the last part, the -- and then a space? It allows us to pass argument through to the underlying command. In this case, we pass an extra -w for watching, and also overrides the reporter with a -R to use the min reporter that makes better sense when you're watching your folders. Let's try it again, npm run watch:test. Oh, great, I almost forgot about that; now that we call the test script, we will also rerun our cleaning and linting on every change. Awesome.

Watching the Unwatchable

Some other tools might not have a built-in watch option. Jshint is one example. What we can do there instead is to use a tool called a watcher, simply a tool that is specialized just to watch folders. One that I've used is called watch. Those Node guys and their funny names. Very inventive, guys, thank you. Let's install it and play with it a bit. The command could not be easier to use. We simply give the command a command to run and the files to watch. Let's create a watch:lint script. This says watch the current directory with a dot and run npm run lint on the changes. We try it out by starting it with npm run watch:lint, and then create a linting error by removing a semicolon in the index.js file. Yes, it crashed, just what we wanted. We see that it picks up the missing semicolon.

Using Nodemon

Finally for the server-side, let's use a common tool to watch our server, nodemon. Nodemon watches our application files and folders and, when a change is detected, restarts the application. This means that we don't have to restart our application manually all the time. We install it with npm install as normal, and add a script like this. The command watches all files in our application directory, ignoring the client and public folders. We start the index.js file. When any changes are made to the watch files, the server is restarted. Let's try it out, npm run watch:server, our application is started, and now when we make the change, the server is restarted by nodemon. Great.

Client Side Watching

There's two parts to watching on the client-side, watching the files to react to changes as on the server, but also update and reload the browser. Let's do what we have done so far, one step at a time. We start with the watching of the files. Many of the tools that we have used so far for client-side development have built-in watching, like CoffeeScript and TypeScript, for example. Here's two tasks that use watches for those scripts. Nothing particularly strange here, just adding the -w flag to both of our commands, and again, if your tool supports that, I recommend this way to go. Browserify that we used for bundling, they have a separate tool altogether. It's called Watchify. After installing it with npm install, we can write a simple bundle minify watching task, like this. The watchify command takes the same parameters as the browserify command did, so that's pretty easy. I've added two flags for watchify. First, I delay execution with -d, so it doesn't run my processor at 100%, and secondly, with v I have verbose login to better show what's happening. Note that this command will not minify our scripts, but that's okay, since we're only developing. Minifying is part of the build step. Do you agree? If not, I will soon show you how you very easily can fix that. We try it out, npm run watch:bundle. Now we make a small change in the client/js/nested/lib, which should trigger bundling. Yes, it does. Of course, we could have used a watcher like we did for the server-side also here on the client should our tool not have a watch option built in. For example, we could rewrite the watch:bundle script without Watchify, like this. Since this now is running our basic build:bundle, we get minification, too.

Live Reloading of Browser

There's a little part missing of our scheme here, the browser. Because just because we regenerated all our files on every change doesn't mean that the browser is updated. To get the browser updated, we have to reload the page. The tool that I've been using for this is called live-reload. It's a nifty little tool that does exactly that, reloads the browser. As with almost every tool I've suggested so far, there's, of course, alternatives. For this course, I'm using live-reload. First, install it, npm install, and save it to your development dependencies. Live-reload has two parts, first a little web server that runs our files, and secondly, a small client script that the live-reload server communicates with. When that client-script is present, the live-reload server triggers a reload on every change. To test this, we need to serve up an HTML page that looks something like this. The only line of interest here is the scripts tag, one that serves the live-reload client. I've also, at the end of the page, added a little script that prints the current time and date so we can see when the page was rendered. Sadly, we need to do a small detour now, because the only problem with this is that we need to serve the HTML from our web server in the index.js file. Koa, cool as it is, is not really the point of this course, so I will keep this exceedingly simple and just hard code html into the page. Don't ever do this unless you're doing it as part of a course, not focusing on Koa. Our new server looks like this. Basically, we just check if the URL ends in /client. If so, we serve the hard-coded html. If not, we serve the text in the body as before. A little bug has slipped by me that I found in this demo. Of course, we need to set the port that we are listening to. Now we can npm start the application to try it out. Let's open a browser on localhost:3000 as before to see that it works. Yes, it does. We get the text as before. And if we had to /client, we get the nice page with the timestamp. Okay, our application works as intended. With that in place, we can now create a script to start the live reload server like this, live-reload --port 9091 public, or in English, start a live-reload server on port 9091

and watch for changes in the public folder. We can try this with npm run watch:browser, but we also need to start our server. Let's create an overarching watch script and start them both. Notice that single ampersand between the commands. That simply means that we're starting the commands concurrently, it starts the command in the background, as a new process. It's very handy to do this for watching tasks. In a Windows environment, this would be accomplished using the START /b command, for example, like this. Okay, we better verify this too, and this will be a little complicated. First, we start the watch task. We open two browsers on the URL localhost:3000 client to show our amazing web page. (Loading) Now, we trigger a change in our generated client by opening a new Command Prompt and run npm run build, the task that compiles, bundles, and minifies all files in the public folder. Watch the clients reload automatically. Whoa, I was a bit worried for that demo, I'm telling you. It works.

Putting It Together

Let's write the one watch task to watch them all. We have already started, and I'll leave it up to you to make this as complicated as you see fit. Basically, when we change a file, client or server, we want to compile, lint, test, build, and minify our code to our heart's content. Luckily for us, we've already done the hard part, written the tasks. Our watch task now simply has to stitch it all together. Here is how it looks now. Pretty good. We are reloading the browser and restarting the server when we make changes. Let's add two more things, running our tests on changes, and also rebundling our client when we make client changes. That's actually not at all hard now. Oh, that looks like a lot. Let's try it. It works. Let's verify. We change our server to fail our test, yes. Changing our file in the client, and it rebundles. Yes. Open a browser, and it reloads, yes. And again, notice that we are using the single ampersand between the commands to run them as separate processes concurrently. And we can shut it all off with Ctrl+C.

Summary

In this module, we have created a lot of scripts for watching that is useful during development and testing. They monitor our files and regenerates tests, compiles, bundle, or what have you, as we change the file. Again, there are special watcher tools like watch, that we used, for example. I recommend that you look into the tool if the tool has a separate watch option. Often that works better in my experience. We also took a look at tools that can help us to reload the browser through the live-reload package. This is very handy for client development where we do not only need to regenerate the files, but also refresh our browser to see the change. We learned a few new tricks. First, to run tasks concurrently by using the single ampersand. On the Windows platform, this could be accomplished with the START /B command. Also in the beginning of the module, we learned that you can pass arguments into the underlying command using the -- space. One example was that we could make our test script use the watch flag by doing this, npm test -- -w. Now the -w flag is passed through to the command in the test script. There the -w flag is used for Mocha, which simply starts Mocha with watching. A final word, this package.json file is getting very long now. It's just for demonstration purposes. In real life, you would grow this little by little, of course, and don't let it get this unwieldy. I trust you to improve on my work. In this next

module, we will take a look at how you can version your source code and push to repositories. See you there.

Versioning, Pushing, and Deploying

Introduction

The last two modules focused on developing and testing, but an application that is not used by users is truly useless, I'm sure everyone agrees with. Therefore, in this module, we will take a look at writing a few scripts to support us in versioning, pushing to our repository, and deploying to our platform of choice. As we said, our scripts node is getting pretty long in the package.json by now. Remember that we can get a quick log of the available commands with npm run. This module focuses on the individual scripts, and in the next, we will put many things that we learned so far during the course together into a big deploy to production script. So, if you're missing a step here, like compiling, testing, bundling, before we're pushing to production, don't worry, that's in the next module. Let's hop to it.

Incrementing Version Number

One of the first and most important steps when we make a release of our application is to get a new version number. Interestingly enough, this is something that npm itself can help us with, with the npm version command. Default with no parameters, npm version gives us some information about the current application. The npm version command has a few switches, and let's spend just a few minutes to dissect a version number and a common way to structure it called semantic versioning. A version number is made up of 4 segments, and we're only focusing on the 3 first, 1.0.411, for example. This entire number uniquely identifies a particular version of a software. With semantic versioning, we assigned the three first positions with a meaning, like this, Major, Minor, Patch. Well, that really doesn't say much about what the meaning of those positions are, so I'll use in my head instead, Breaking, Feature, and Bug Fix. Basically, increment the first position, Major, if you have a breaking change. Increment the second position, Minor, when you release new features. Increment the third position, Patch, for minor changes or bug fixes. I personally prefer small changes often rather than big seldom, so I tend to mainly use the two last positions. With the help of npm version --help, we can see that the help text of npm is again very helpful. It has switches to set the values for us, npm version major increments the major part, npm version minor increments the minor part, and of course, npm version patch increments the patch part. Creating scripts for this is now trivial. Right now our version number in the package.json file is 1.0.0. Let's run npm version patch and see what happens. Well, it returned a string at the prompt, v1.0.1, so that's promising, but also, the number in the package.json file was incremented, which is even better, of course. Our application is now at version 1.0.1. It would be great to note that in our source code repository, too. Let's get to that. Notice that this command increments the patch number. I could also set a specific version number as well should I need it, but I seldom do.

Versioning Code in git

Until now, I have not used source control at all. Shame on me, but it was really to not complicate things for us too much. We need to change that. Let's use git. This is, of course, not a course on Git, but basically we want to add a gitignore in the root, and we use a gitignore from the excellent site gitignore.io. Then, we can initialize a source code repository with git init and a dot. We add all the files we've been working on so far with git add --all, and finally we commit our changes, git commit -n and a suitable text. Running git status now shows that our file is safe in git. Normally we would tag this position doing a release in the git repository by using the git-tag command and passing the new version number. Running git-tag now shows that there are no tags in the repository. We don't have to use git to set the tag, we can use npm version instead. Let's add, if you don't have it already, some source control repository information to our package.json file. Basically this says that we are using git, the repository is in our current directory, the URL property could point to a remote, like github, for example. Now let's save and commit those changes, and then increment the version again, npm run version:patch. As before, our package.json is updated with a new version number, 1.0.2, but now when we run git-tag, we see that npm has set a tag, a marker in our git repository history for that version. What is really cool with this is that the change that was made to package.json incremented the version number, is committed into that tag. We can see this with git show and the version number. So basically using npm version increments both the package.json and sets a tag in our git repository. Very handy.

Push Code to Repository

Now our application is versioned and the version is set in our source control repository, it's time to push the code to our upstream repository, should we have one where we share code with others. This means that this step of the course is, of course, optional. Our repository is so far local, as git remote shows us. There are no remote repositories. Adding an origin or upstream remote to git is pretty easy as far as git goes. Git remote add, the name of the remote, origin in this case, and then the URL to the git repository. Where the remote origin points, of course, is up to you. I have created the repository at GitHub and added that to my remotes. You can list the remotes with git remote -v to see the URLs of the remote. In my case, that lists the original repository only for now. Normally when you have set the tag in git, you will push the code to a remote called origin, for example, with a git command that looks something like this. The --tags switch means that git will include the tag information in the information we are pushing. So let's use that and create a script like this. At this point in my recording, I ran into a little problem, since I tried this many times before. Git complains when you use the same tag over and over, which is a really good thing, of course. So I had to set the version number to something that was new. This is the reason this amazing app is now on 2.0.3 or something like that. Also when I did that, I had another problem that you might run into. I ran npm run version major and got this error. Basically this means that git is trying to set the tag when you have unsaved changes. I forgot to add and commit my changes

in the package.json file when I added the new script. So, commit and add those changes, and now finally, let's try to run the push script, npm run push:origin. Yep, it works.

Deploy the Application

The next step is to actually deploy to our test staging or production environment. This is probably part of the course where there are most options, different setups, opinions, and tools. I'm not even going to try to cover any substantial part of all of that. Instead, if your tool has a command line interface, like Azure, Heroku, and Amazon, for example, then it should be pretty easy just to create a script from those commands and put it into the package.json file. That being said, I want this course to be more or less complete, so let me show you how I have done this for my provider of choice, Heroku. First, I need the client tools to be able to access Heroku easily. They are called the Heroku toolbelt and can be downloaded from toolbelt.heroku.com. Then, I create an application in Heroku for my local application with a simple command, heroku create and the name of the application. The name is optional; Heroku will make one up for you if you leave it out. Among many things, this will create another git remote for us called heroku, as shown when I run git remote -v again. Deploying to a new version of Heroku is now as simple as just pushing to that remote. Let's create a script for that. There's a little update we need to do to make our package.json production worthy. We should add the version of node and npm that we are using. And we, of course, need to add and commit that into git. And now we can try it out by running npm run push:heroku. When we run this, we get a lot of logging information about what Heroku is doing on its part, but after awhile our application is deployed. Just to show some other examples, here is how it would look if you're deploying to Amazon S3 using the S3 client tools, and publishing to Azure, you can also push to a git remote as for Heroku.

Launch the Application

I've grown into the custom of making little scripts that start the application once it's deployed to its new environment, just to make sure that it shows up. A silly little thing, maybe; it's saved me from big embarrassment a few times. For example now, because opening stupidlittlewebsite.heroku.com will show that the application actually doesn't work. This has to do with a faulty if statement that we used for our port. Heroku sets the port in production, it might be new every time, via an environment variable, and our code used that, but my clumsy if statement overwrote it. I didn't catch that until now. Luckily I opened the site and checked. Here is what we need to do to change into the correct code. In my case, being on an OSX system, launching the application once it's deployed, is very easy to do. We can use the open command and supply the URL to the site, like this. I love it, and it works. Let's create a script for that right away. Launch:prod. On a Windows system, we will have to use the start command and the URL instead. Using Heroku, this is even easier and even platform safe, since the Heroku toolbelt exposes an open command, that is the same for both Windows and OSX and Linux and what have you, so we can just simplify this to Heroku open. Also, this means that we don't have to hard code the URL

into the configuration, which is very cool. Let's try that, npm run launch:prod. There. Our little website lying on the internet. They grow up so quickly.

Summary

This module was another one with a lot of new scripts in our package.json file. You probably used most of the commands or similar before; now you can reach all with npm run. In the next module, we'll take a look at putting it all together to one big deploy to production script. That will be really exciting. See you there.

Deploy Script and Additional Tricks

Introduction

Okay, let's wrap this course in one big deploy to production script. Here is what we want it to do. First, lint, compile, and test our server-side JavaScript. Secondly, bundle and minify our client-side JavaScript. Compile our less code to CSS. Bump the version number to the next patch number, push to a remote GitHub repository, deploy to Heroku, Well, let's open it to verify that our application works, too, just for good measure. All of the parts are written. We're just going to put it all together. As it turns out, this will be pretty easy.

Deploy

First, let's create a new script and call it deploy:prod. We might have deploy scripts for other environments, too, later. First on our list is lint, compile, and test our server-side code. Looking through the many, many scripts, we actually see that the test and pretest in combination will accomplish exactly this, meaning that if we just call npm run test, it would accomplish our goal. However, this spec reporter might be a little bit verbose for a deployment script, so let's write a specific script for testing in deploy. We call it test:deploy, and npm t -- -R dot. We simply override the reporter switch using dots instead of the verbose spec reporter. Now, we can start writing our deploy to production script, and right now it's just npm run test:deploy. Try it out, npm run deply:prod, and it works. It is a little noisy, though. Let's add the -s switch to the command and call it with npm run deploy:prod -s. It silenced out the logging a bit for us. We can strike one task off the list, onto the next. Now we're going to bundle and minify client-side JavaScript. This is a tall order, but luckily we have already created some great scripts for this. Let's review them. Basically when we call the build script, it will clear the folders, compile to less in CSS, then bundle and minify the client-side JavaScript. Awesome. We append that to our deploy:prod script, npm run build. I'm using the double ampersand to concatenate the task, since I don't want to continue if the first step fails. You could argue that this first step could run in parallel and then you could use the single ampersand to speed up build time. For now, I'm using double ampersand. Let's test it out, npm run

deploy:prod. And it works. There's some warnings from the bundling and minification that we probably should look into, but I'll leave that up to you and continue. Next, less to CSS. Ha! Already done. Next! We are now entering the scripts from the last module. We're going to bump the version to the next patch number. This is easy. Let's just hook the scripts up to our deploy to production script, get a new version number, we'll start with the npm version command, and we call that script version:patch. Now this script always increments the patch part, but that will have to work for now. You might have other more advanced things passing in parameters, et cetera. I have some tips for that later on in this module. That makes the entire deploy to production script look like this. Testing that, npm run deploy:prod, and it fails. Oh, this is the same error as before. We have now made changes in our package.json file, and since this part is updating our git repository, we get a fail because the git repository has uncommitted files. To get this to work, we have to commit the changes in the package.json file in which we are developing now. Luckily for us, the build halted on error. When we have committed the package.json, it works as expected. Great. Next task is up. Next is push to a remote git repository and then deploy to Heroku. Both of these are easy. Just append the two scripts, push:origin, and push:heroku to the deploy:prod script. Looking at that script, it's pretty long now. Let's refactor it a bit. Let's create a script called push, and call the two scripts for pushing from there. Now we can shorten our deploy:prod script to this. A little bit better. Only one thing left, next! Finally, let's add a little launch task at the end that saved us in the last module. There. Commit those changes in the package.json file, and then test run the whole thing with npm run deploy:prod. (Loading) Oh dear, that's a lot of logging, but after awhile, yes, it works!

Reactoring and Extras

So, it works. We know what that means. We are now allowed to clean up a bit. Now we have something that we can safely see if we have broken or not. First, that script is a bit hard to read due to the long string and sadly the lack of multiline strings in JSON. However, this can be solved easily by just breaking the commands out to a separate file. Here's a new script that uses a separate file for a deployment script. In the deploy:prod .sh, I just copied a script and put every command on a separate line. There, much easier to read, but also now we have code in more than one place. It's a trade-off that you will have to make for yourself. I'm leaving both versions in the package.json for your consideration. Also note if you're on Windows, use a .bat file instead. One thing that we can easily add is a timer for our build using the bash command time. Just wrap the entire command with a time command. Or even better, create a new deploy:prod:time task and simply call the deploy:prod task from there. And now we can get a very simple timing for our entire deployment script. Sadly, there's no command like time on Windows to my knowledge. Sometimes I found it useful, especially during development of scripts, just to echo out what the script is doing. This is easy to do with the pre and post hooks like we described earlier. Here's an example. In this way, I am not polluting the actual script with logging. These messages will be shown, even though you silence the output with -s.

Options, Options, Options

Our deployment script is done. Yay for us! I wanted to mention a few alternatives for how you can handle options and arguments to commands and scripts that we haven't used yet. We didn't use this in our package.json file scripts, but you might want to later. For some of our scripts, we are passing a lot of arguments. A common example is to the mocha command, our test command. Right now our test script looks like this, which is not too bad. Let's look at a couple of alternatives to move those arguments and option switches out of our way. First, those parameters can be moved to a separate file with options. For mocha, that file is called mocha.opts, and should go in the test directory. It's just a plain text file where we list the parameters on each line; for example, like this. This now makes our script trivial, simply mocha. You can start it just like normal, npm run test:options. In this case, this is not necessary to have a separate file, but I wanted to show you the possibilities. As I mentioned before, we can override switches using the -- space flag to npm. For example, we could change the reporter that is set to dot in the mocha.opts file to nyan, like this, in our script, Or when we call the script itself from the Command Prompt. Another options alternative is built into package.json itself, the config value. I'm not sold about this, but let's check it out anyway. You can define a new config node in package.json, like this. We can now use the values defined in here in our scripts, like this. Notice the $npm_package_config_reporter. Now when we run this, npm run test:configoptions, that weird landing strip reporter is used for Mocha. But why do that, you might ask? Well, this option can be overridden, in two ways. First, in another command like this. The --stupidlittlewebsite refers to the name property of our package.json file. We can also override it with a command file for the user or globally on the system by setting the configuration variable, like this. A file is created with the reporter configuration set to tap, a test reporter, for the stupid little website project. This file is called .npmrc. We can see that new configuration is picked up by running npm run test:configoptions. It's now using tap as a reporter. I presume this can be useful for build machine, et cetera; I'm not sold. This seems a little bit complicated to me. But now you know about it, too.

Summary

In this module, we put many of our scripts together in a big deployment script, deploy to production. It was pretty simple, since we just could stitch the different scripts together. In doing so, we saw some opportunities to clean up a bit, and also the package.json file we have right now is crazy long; maybe it should not be this long in real life. We also learned about extracting scripts to separate files and moving options and parameters to separate files to further clean up the package.json and increase readability and understanding. We're almost done. Let's summarize.

npm as Build Script Summary

Introduction

This course set out with the assumption that we could npm and the package.json file as our build tool. This is the way that Node and npm themselves are built, and take away the dependencies on third-party

products, such as Gulp and Grunt and all of their packages. To our help, we have the entire plethora of packages and tools that npm offers, the largest Open Source library in the world. Through the course, we have created a monstrously long package.json file with a lot of example scripts and ways that you can use the scripting capabilities of npm to your advantage.

Summary Concepts

Let's look through that package.json file and I'll point out some concepts as we go. First of all, use npm init to create the file. The scripts are found in the scripts node, of course. Scripts can be executed with npm run and the script's name. Some scripts are so common that they have npm shortcuts, npm test and start, for example. It's easy to store any command as a script in the package.json file. Anything you can write and execute at a Command Prompt you can also execute with npm as a script. To extend the functionality of our scripts, we can install Node packages, and then call to them like this, for example. The commands that we have available are easy to list by just going ls node_modules/.bin. There are also pre and post hooks that will run automatically before and after the script with the same name; prebuild will be run before build, postbuild will be run after build. It's easy to call other scripts from our own script, npm run and the name of the script. It's also easy to change commands. && means stop an error and & means run in parallel. Many tools, like Mocha and CoffeeScript, for example, have options to watch a folder or files for changes and rerun on changes. There's also tools that only watch folders for changes and execute a command, like watch that we used, or nodemon. Should a command become very long, we can have the options in a separate file, like we did for Mocha, or move the entire command to a separate file, like we did for our deployment production. We can get a quick overview of the scripts in our package.json file by running npm run without parameters.

Outro

I think npm is a very valuable solution for build scripts. It's easy to get started, the package.json is already present, there's no new tools and API to learn, it's easy to move or call your existing automations within npm, and it's just JavaScript that you already know if you're developing on Node, for example. I hope that you have enjoyed viewing this course as much as I did creating it. I learned many new things in researching and recording this course. My hope is that you picked up many new things, too. Thank you for your attention. Happy scripting!