CODER CODER PRESENTS...
GULP
FOR BEGINNERS
UGLIFY
BABEL
Automate
your workflow!

Course Sections

1 GULP AND NODE.JS

Node Package Manager (npm)
Versioning and dependencies in npm
Track packages with package.json

2 SETTING UP GULP

Gulp CLI
Installing Gulp
Configuring the Gulpfile

3 CONCEPTS IN GULP

Vinyl objects, src(), globs, dest(), and pipe()
series(), parallel(), and watch()

4 INITIAL FILE SETUP

5 SASS WORKFLOW

Set up Sass files and packages
Create a Sass task in Gulp
Load the website with a local server
Working with vendor files

6 JAVASCRIPT WORKFLOW

Set up JavaScript files and packages
Create a basic JavaScript Gulp task
Run Babel for ES6 syntax
Bundling modules with Browserify

7 UTILITY GULP FUNCTIONS

Image minification
Cache busting

8 OPTIMIZING GULP

Watch task and Browsersync
Dev and production tasks

Introduction

Hello, and welcome to the Gulp 4 for Beginners course!

Gulp is my build tool of choice, and I'm excited to help you learn how it all works.

When you've finished this course, you should have a better idea of how to build a simple front-end workflow using Gulp. We'll be compiling Sass, bundling JavaScript, running a local server with Browsersync, and optimizing our files.

But before we begin, I want to share some tips that will help you go through this course more effectively.

We are going to go through quite a bit of code as we build out our Gulp workflow. I recommend that you try to manually type things out as you go through each section. It's tempting to simply consume the material and copy and paste code, but you'll learn and remember the information much better if you write out all the code yourself.

I do have the final source code for the Gulp workflow up on GitHub, but I recommend only using it as a reference or as a last resort if something isn't working.

With that said, the only tools you need to go through this course are a computer and a code editor (I use VS Code) so let's get right into it!

CREATING OUR PROJECT

Let's start by creating our project folder that all our files will be in. So open up your file explorer and create a new folder. I'm calling mine "gulp4-for-beginners" but you can call yours anything you want.

Then, open VS Code and open that folder by selecting File > Open Folder, and select the folder you just made. This will be our workspace for the course!

For now we'll leave it empty, but as we continue through the course we'll add files and folders to it.

SECTION 1

Gulp and Node.js

Since Gulp uses Node.js and specifically utilizes Node streams, let's take a look at what Node.js and npm (Node Package Manager) are all about. (If you are already familiar with Node and npm, you can just skip down to the Setting Up Gulp section!)

Node.js is a run-time environment that can execute JavaScript code on the server. It can be used in the server-side of web apps, and you can also run it locally using the command line interface (CLI).

Gulp runs using Node.js, so before installing Gulp itself, you need to make sure you have Node installed on your local computer. First, open up your CLI. On Windows this might be Powershell or the "cmd" window. On Mac you can use the Terminal program.

I usually like to use the integrated terminal window in VS Code. You can open it by clicking on

If you're not sure whether you have Node installed, you can check by typing in `node -v`. If it is installed, it'll return a version number like you see here.

```
PS C:\Users\Jessica\Documents\GitHub\gulp-for-beginners> node -v
v8.16.0
```

If you don't have Node installed, you can download and install it from the NodeJS.org website.

Once you have it installed, you'll now be able to install other tools (like Gulp) on your computer via npm, Node Package Manager. Npm is automatically added when you install Node.

NODE PACKAGE MANAGER (NPM)

Node Package Manager is a giant library of collections of code that individuals and companies have created and published. Each collection is called a package.

And, a lot of the code is open source, so we can use these packages for free. The reason we're interested in npm is because Gulp is one of these packages!

Some packages may be tiny and just contain a few functions. One example of this is camelcase, which converts regular strings into camel case (alternate lowercase and uppercase capitalization in a compound word).

Others packages are large and complex, like Express.js, and React.js. Both are packages that get millions of downloads per week.

So how do we install packages onto our computer? To do that we need to go back to our command line interface.

PACKAGE.JSON FILES

Now, before we actually start installing packages, we need to create a `package.json` file. This is a file that will keep track of all the packages we install in our project, as well as which version each one is.

To create your `package.json` file, open up your code editor and in your terminal window, type in `npm init -y`. The "y" flag will automatically fill in the default values for the metadata.

Now you should see a `package.json` file in the directory. If you open it up and take a look, you can see some basic information:

```
{
  "name": "gulp4-for-beginners",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Again, since we're just using this to install some packages locally, you don't need to worry about filling any of this in,but you can if you want.

Let's try installing the camelCase package that we were talking about earlier. In your terminal window type in `npm install camelcase`. This format of `npm install` followed by the package name, can be used to install any npm package.

When you install the camelCase package, you may see some warnings about not having a description or repository field, but you can ignore those.

You may have noticed that installing the package has also created a `node_modules` folder in your project root. There's also a subfolder named `camelcase`. As we install more packages, they will all get installed to this `node_modules` folder as well.

There was another file created called `package-lock.json`. It's similar to the `package.json` file, but a little different. Because it includes information about the actual version that we installed on the computer.

This gets into an important aspect of using npm packages: versioning.

VERSIONING IN NPM

Understanding how versioning and version dependencies work will help you troubleshoot errors when you're working with lots of packages.

When you're using multiple packages, not every version of every package is compatible with every version of the other packages. And if you have installed incompatible versions of packages in your project, it may cause errors and other issues.

These can be rather frustrating to resolve, because it may involve a lot of Googling and searching through issues in the different GitHub repositories. I wanted to mention this so that if you have weird errors, you can consider compatibility issues as potential cause, and hopefully avoid too much frustration!

Let's take a look at our camelCase package to see how versioning works. If we open up the `package-lock.json` file, we can see under `camelcase` that the version has a specific number. I have `5.3.1`, but you might have a newer version by the time you see this (which is fine).

This format of writing the version number is called semantic versioning or "semver" for short. In it, the author follows a pattern called `Major.Minor.Patch`. This pattern is made up of three numerical digits separated by a dot, is what most software products use.

The number looks sort of like a decimal number, but it has two decimal points. Each digit in the number represents a type of update that the author of the package has released.

The first digit is for tracking the major versions. These are very big changes that will usually not be backwards compatible and may cause errors, due to large overhauls in the code. The first public version will be `1.0.0`, the second major version will be `2.0.0`, and so on.

The second digit is for minor version changes. One example would be moving from `1.2.11` to `1.3.0`. Note that unlike a decimal, the numbers in each digit can be greater than 9. And when you increment major or minor versions, the digits after the one that is changed reset to zero. Minor changes theoretically shouldn't cause any backwards compatibility issues, but sometimes will in practice.

The third digit is for patches, or bug fixes. These are quick updates that shouldn't cause any issues when updating. An example would be moving from `2.14.6` to `2.14.7`.

Let's take a look at the version history for camelCase as an example.

Version History

Version	Released
5.3.1	10 months ago
5.3.0	10 months ago
5.2.0	a year ago
5.1.0	a year ago
5.0.0	2 years ago
4.1.0	3 years ago
4.0.0	3 years ago
3.0.0	4 years ago
2.1.1	4 years ago
2.1.0	4 years ago
2.0.1	4 years ago
2.0.0	4 years ago
1.2.1	4 years ago
1.2.0	4 years ago
1.1.0	5 years ago
1.0.2	5 years ago
1.0.1	5 years ago
1.0.0	5 years ago

We can see that the current version (as of this writing) at the top is `5.3.1`, and at the bottom is the very first major release, which is always `1.0.0`.

If we go up from the bottom we can see how the numbers change with each release. After the initial `1.0.0` release, there were a couple patch updates, `1.0.1` and `1.0.2`. Then there was a minor update, `1.1.0`, which also reset the patch number to `0`.

The same reset happens when a major update gets released. Going from `1.2.1` to the major update `2.0.0`, the patch and minor numbers reset to `0`.

In each case, a major or minor update will automatically reset the smaller digits to `0`. This is because each number is essentially tied to the bigger update.

Hopefully that helps explain how the versioning and numbers work! You can read more about versioning in the npm documentation.

Now that we've gone through semantic versioning, you might be wondering which version of a package should you use? Usually, you can safely assume that you can install the latest current version of each package you need. When you use the `npm install` command, npm will automatically install the latest version, including minor and patch updates.

Just be careful when you're updating existing packages in your project to a new major version, due to breaking changes as we mentioned earlier. You can check for specific breaking changes and issues in the GitHub repository of the package.

USING PACKAGE.JSON AND PACKAGE-LOCK.JSON

There's one other helpful feature of the `package.json` and `package-lock.json` files that I wanted to mention. And that is that you can use them to install the same set of packages in different project folders on your computer, and even on other computers and servers.

So if you wanted to use the same Gulp workflow from this course in a new project folder, all you would have to do is take the `package.json` file and copy it into the new folder. Then if you run the command `npm install` in that new folder, npm will automatically install all the packages from that `package.json` file.

Where the `package-lock.json` file comes in is related to semantic versioning.

For example, in our `package.json` file, the version of camelcase that we installed is listed as `^5.3.1` (or whichever version is current at the time you're reading this).

The caret (`^`) symbol will install the latest minor version in the major version that you designate. So this means that running `npm install` with this `package.json` file will install the newest minor and patch versions of camelCase that are in the major version 5.

However, if you want to install specifically version `5.3.1` and nothing else, you can use the `package-lock.json` file to install that specific version. If you ensure that the `package-lock.json` file also exists in the same folder as the `package.json` file, when you run `npm install`, it will read from both `package.json` and `package-lock.json` files and install the `5.3.1` version of camelCase.

Now that we've gone through the basics of Node and npm, let's move on to installing everything we need for Gulp and Gulp-related packages!

SECTION 2
Setting up Gulp

GULP CLI

To install Gulp, the first package we need is the Gulp CLI or Command Line Interface. The Gulp CLI lets you manage multiple versions of Gulp on your computer, if you need different versions for different projects.

> **Note:** If you have `gulp` globally installed (this was an older way of doing things), you will need to uninstall your global `gulp` first. Then you can install the Gulp CLI.
>
> To check if you have the Gulp CLI, type in `gulp -v` and see if it returns a message saying "CLI version..." If it mentions Gulp but not the CLI, uninstall the global Gulp by typing in `npm rm --global gulp`. Then you can continue installing `gulp-cli`.
>
> You can read more details about that on the Gulp blog (on Medium).

To install the Gulp CLI, on your command line, first type in `npm install --global gulp-cli`. The `--global` flag means that the package will be installed and accessible from any folder on your computer.

Once you have the Gulp CLI installed, you won't have to do it again (yay!)

INSTALLING GULP

Let's start by installing the gulp npm package. In your command line, make sure you're in your project's root directory.

Type in `npm install gulp`, and npm will start installing all the necessary packages (this might take a minute or so).

Once the install is complete, let's check out our `package.json` file:

```
"name": "gulp4-for-beginners",
"version": "1.0.0",
"description": "",
"main": "gulpfile.js",
"dependencies": {
  "gulp": "^4.0.2"
}
```

We can now see the "dependencies" section, with an entry for the Gulp package. We've installed the latest version, 4.0.2 as of this writing. You may have installed a newer patch or minor version, but everything should work just the same.

If you look at the console messages when you installed, you may have noticed that you installed over 300 npm packages. That's a lot! Why is that? You only meant to install Gulp.

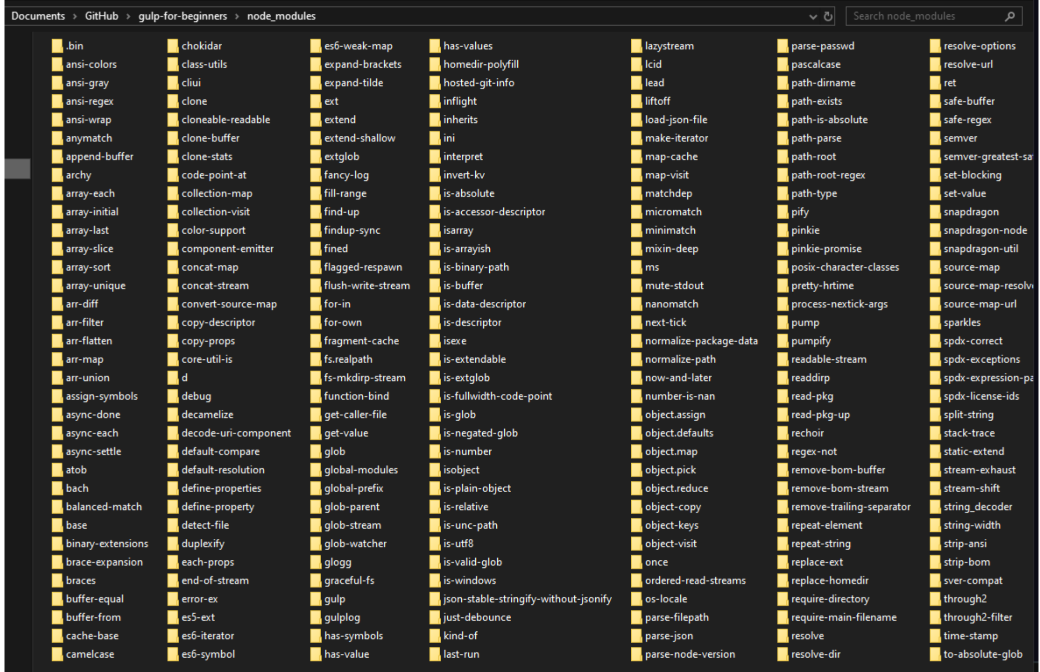
```
PS C:\Users\Jessica\Documents\GitHub\gulp-for-beginners> npm install gulp
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN gulp-for-beginners@1.0.0 No description
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.9 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.9: wanted {"os":"d
arwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

+ gulp@4.0.2
added 316 packages from 217 contributors and audited 6509 packages in 15.485s
found 0 vulnerabilities
```

Let's check out the new `node_modules` folder. Open it up and you'll see that there are a *ton* of subfolders. Each one contains a separate npm package, also called a Node module.

As we mentioned earlier, npm allows you to utilize packages that other people have already created.

The authors of Gulp are using other existing packages to perform certain functions. And the authors of *those* packages are using other packages as well. You can see all the other packages that Gulp itself is dependent on in the `node_modules/gulp/package.json` file.

This is what leads to having so many subfolders in your `node_modules` folders. You don't have to understand what every single one does, but at least knowing **why** there are so many will help.



You can also see all these Gulp dependencies listed if you go back to your project's `package-lock.json` file, in the `dependencies` section. These will match all the subfolders in your `node_modules` folder.

We'll be installing more Gulp-related packages, but first let's go over how Gulp works, and what functions you can run with it.

USING THE GULPFILE TO RUN GULP TASKS

The core of the Gulp functionality is in a configuration file that we will be creating called the gulpfile. The gulpfile is a JavaScript file that loads the npm packages you need as node modules. And it's where we will write our Gulp functions, called Gulp tasks.

Gulp tasks can be public, meaning they can be exported to run on the command line. Or they can be private, only able to be executed within the Gulpfile itself.

You need at least one public task, called the default Gulp task, and you can run it by typing `gulp` into the command line. This default task will then contain private tasks to run all your Gulp functions.

WRITING "HELLO WORLD" IN YOUR GULPFILE

Now, let's get acquainted with some Gulp functionality by creating a Gulpfile with a short "Hello World" function in JavaScript.

In your project root, create a new file called `gulpfile.js`. It's important that this is named exactly that. We'll be referring to this `gulpfile.js` file as simply the "gulpfile" in the future.

So your project folder may look something like this:

```
(project folder)
    |-- node_modules (folder)
    |-- gulpfile.js
    |-- package-lock.json
    |-- package.json
```

Open your gulpfile and create a `helloWorld()` function:

```
function helloWorld(cb){
    console.log('Hellooooo world!');
    cb();
}
```

You might be wondering what the `cb` parameter is all about. It's a callback, and running it signals that the function is complete. This is something required in Gulp 4, because in this new version all tasks are asynchronous JavaScript functions.

Synchronous functions execute one after the other, and each one has to wait until the one before it is finished before it can begin. Asynchronous functions, on the other hand, can start running while still waiting for the result of previous functions.

This makes processes more efficient since they aren't blocked. But in order to know when a function has completed, we need them to return explicit signifiers that they are done.

Most Gulp tasks that we'll be using will return a Node stream to signify that the function is complete. Node streams are a way that Node reads data from the files we will be processing in Gulp.

However, the `helloWorld()` function isn't returning anything. So to explicitly signify that the function is done running, we have to put a callback function at the end of it. Otherwise you will get an error saying:

```
The following tasks did not complete: default. Did you forget to signal async
completion?
```

Once our function is done, the next step is to create a public default task by using an `exports` object:

```
exports.default = helloWorld;
```

This creates the default Gulp task, which we can run by typing `gulp` on the command line. And in that default Gulp task, we will run our `helloWorld` function.

Let's see what happens when we type in `gulp` on the command line:

```
PS C:\Users\Jessica\Documents\GitHub\gulp-for-beginners> gulp
[12:09:21] Using gulpfile ~\Documents\GitHub\gulp-for-beginners\gulpfile.js
[12:09:21] Starting 'default'...
Hellooooo world!
[12:09:21] Finished 'default' after 2.04 ms
```

Obviously this is a very small example. But it's a good way to show the basics of how Gulp works and how we can get it to run any functions that we need.

Ultimately we will have Gulp run several different tasks as part of the default task. And they will follow the same principles as running this `helloWorld` function!

(And by the way, you can delete all the `helloWorld()` code from the gulpfile since we don't really need it anymore.)

IMPORTING NODE MODULES INTO THE GULPFILE

In our gulpfile, we will be using JavaScript to write our Gulp tasks. First, we need to import all the necessary npm packages into our gulpfile so that they are accessible in our code.

In a previous step, we installed the Gulp package by typing in `npm install gulp`. This added Gulp into our `node_modules` folder. Now, to import the Gulp package as a module, at the top of our gulpfile we will add the following line:

```
const { src, dest, watch, series, parallel } = require('gulp');
```

The `require()` function in Node.js will search the `node_modules` folder for a package named "gulp" and then import it into the gulpfile. We can then use methods from the Gulp API like `src()`, `dest()` and `watch()`.

We'll be installing and importing other npm packages for our workflow. But for now, let's take a closer look at the Gulp methods and some other concepts.

SECTION 3

Concepts in Gulp

Gulp has some methods and concepts that are frequently used in creating Gulp tasks. Let's take a look at what each of these does:

VINYL OBJECTS

Vinyl is a virtual file format created by the Gulp team specifically for Gulp. A Vinyl object contains metadata about the file in question, such as the path and the contents of that file.

According to an article written by the Gulp team, Vinyl allows Gulp to use Node streams in object mode, and it can read and return data as Vinyl objects.

Object mode is good because it lets Gulp process multiple files in one stream, which makes it run faster. However, the downside is that this mode and Vinyl objects themselves are incompatible with Node streams in buffer mode, which many other npm plugins use.

This is one reason why we often have to use npm packages specifically designed for Gulp, like `gulp-sass` and `gulp-concat`. But it *is* possible to use non-Gulp plugins by converting file data between the two types of streams. (We'll go through this later when we set up Browserify for bundling JavaScript files.)

Now let's take a look at the methods that Gulp uses to process files:

SRC()

The `src()` method reads files from our file system as Vinyl objects, and returns them as a Node stream. We can use this to read JavaScript, Sass, and other files, and then pass them off to other processes in our pipeline.

For example, writing `src('js/scripts.js')` will create a stream to read the `scripts.js` file in the `js` folder.

You can also read multiple files in the `src()` method by putting them in an array like this: `src(['js/script.js', 'js/script2.js'])`.

GLOBS

Globs are a way of referencing file paths that lets you use wildcards and other characters in addition to regular characters. Here are some of the special characters you can use in globs:

- (*) star -- the star, or asterisk, acts as a wildcard for any string. For example, to reference any JavaScript file, you could write `'*.js'`.
- (**) double star -- the double star can stand in for any type of characters, including blanks. It's most often used to reference multiple nested folders. As a hypothetical example, `'js/**/*.js'` would reference JavaScript files in the `js`, `js/plugins`, `js/modules`, and `js/modules/blocks` folders.
- (!) negative/not -- this is used to exclude specified files or folders when referencing files. For example, `'*.js, !all.js', !js/plugins/*.js'` would refer to all JavaScript files **except** for the `all.js` file and and JavaScript files in the `js/plugins` folder. Make sure to add the exclusion glob at the end, after any other inclusion globs.

Globs are most often used in conjunction with the `src()` method.

DEST()

The `dest()` method is a counterpart to `src()`. It can write processed data from a Vinyl object in the stream into a new file and location.

For example, the method `dest('dist')` will write the Vinyl objects in the stream to the `dist` folder in the file system.

PIPE()

The `pipe()` method is part of Node, not specific only to Gulp. It's used to chain together multiple steps in a process.

Here's a test example that pipes `src()` and `dest()` together in a task:

```
function moveFiles(){
  return src('js/scripts.js')
    .pipe(dest('dist'));
}
```

The `moveFiles()` function creates a stream, reading the `js/scripts.js` file as a Vinyl object, then moving it into the `dist` folder. In real applications we would usually also pipe some additional methods between `src()` and `dest()` to process the file.

SERIES()

The `series()` method is one of the ways that you can execute Gulp functions. Using `series()`, you can run multiple functions one after the other.

Here's some code that runs two functions, `doSomething()` and `doAnotherThing()`, using `series()`:

```
series(doSomething, doAnotherThing);
```

PARALLEL()

The `parallel()` method is similar to `series()`, except that you can use it to run multiple functions at the same time. You can combine `series()` and `parallel()` in order to run some functions simultaneously, and other functions in a specific order.

Here's an example using both `series()` and `parallel()`. It will run the `doSomething()` and `doAnotherThing()` functions simultaneously, then once both are complete it will run the `doLastThing()` function:

```
series(
    parallel(doSomething, doAnotherThing),
    doLastThing
);
```

WATCH()

The `watch()` method is very important in Gulp, as it allows you to continually watch certain files for changes. Then when it detects changes, you can tell it to run a function or even multiple functions.

Here's an example of a watch task that uses `series()` and `parallel()`:

```
function watchTask(){
  watch(['css/*.scss', 'js/*.js'],
    series(
      parallel(doSomething, doAnotherThing),
      doLastThing
    )
  );
}
```

The `watchTask()` first watches `*.scss` files in the `css` folder and `*.js` files in the `js` folder for any changes. If it detects any changes, it will then run the following functions:

First it will run `doSomething()` and `doAnotherThing()` simultaneously using `parallel()`. Then once those tasks are complete, it will run `doLastThing()`.

Just as a note, I personally only use `series()` and run everything one after the other. I've tested out using `parallel()`, but haven't noticed much time saving, so I stick with just using `series()`. However I recommend at least trying out `parallel()` so you know how it works.

Now, let's put together all these Gulp methods and concepts that we've discussed, and create our workflow tasks!

Before we begin, I just want to note that in web development, there is almost always more than one solution to any given problem. I'll walk you through how I like to set up Gulp, but you might have differences in your own situation.

You might name your folders and files something different from me, or even use a different folder structure. And that's totally ok!

In the following section (which is the majority of this course), we'll be building all our Gulp tasks.

First, we'll create private tasks that will process our Sass, JavaScript, and image files, watch for changes, and run cache bust and Browsersync functions.

We'll also create our default public Gulp task, that we can run on the command line. And we'll create a slightly different production task that we'd want to run only when deploying to a production server.

SECTION 4

Initial file setup

To continue setting up our project files and folders, let's create an `app` folder, with `app/scss` and `app/js` subfolders. These folders will store our initial Sass and JavaScript files.

We'll also create a `dist` folder, also in the root, where we will place all our final, compiled files.

Lastly, we'll want to create our `index.html` file, also in the root. In your `index.html` file, we'll add basic markup and references to the final CSS and JavaScript files.

Now, I know I did say "don't just copy and paste" at the beginning of this course... but I'm assuming that you already know basic HTML.

And since this HTML markup is simply to help us test our Gulp workflow, I don't think it's wrong to copy and paste this code into your file 😁:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Gulp Project</title>
    <link rel="stylesheet" href="dist/style.css">
</head>
<body>

    <h1 class="white">Gulp Workflow Project</h1>

    <div class="content">
        <div class="content__block">
            <h3>What we're doing in Gulp</h3>
            <p>
```

```
                Lorem ipsum dolor sit amet consectetur adipisicing elit. Atque
voluptates ut quidem deserunt fugit, a cumque provident mollitia sequi incidunt
dolorem dignissimos, nesciunt reiciendis officiis? Id voluptatibus praesentium est
totam!
            </p>
        </div>
        <div class="content__block">
            <h3>Another thing we're doing in Gulp</h3>
            <p>
                Lorem ipsum dolor sit amet consectetur adipisicing elit. Atque
voluptates ut quidem deserunt fugit, a cumque provident mollitia sequi incidunt
dolorem dignissimos, nesciunt reiciendis officiis? Id voluptatibus praesentium est
totam!
            </p>
            <p>
                Lorem ipsum dolor sit, amet consectetur adipisicing elit. Pariatur
quaerat possimus qui reiciendis illo voluptas a ad fugiat vitae optio! Cupiditate,
fuga deserunt. Commodi impedit pariatur, ut corporis ullam reprehenderit.
            </p>
        </div>
    </div>

    <script src="dist/scripts.js"></script>
</body>
</html>
```

In addition to the markup, notice that we're loading the `style.css` file in the `<head>` tag, and the `script.js` file in the `<body>`, after the rest of the document has been loaded. Both files will be loaded from the `dist` folder. (You don't need to worry about creating those two files now, we will get to them later!)

Our project structure should now look like this:

```
(project folder)
    |-- app (folder)
         |-- scss (folder)
         |-- js (folder)
    |-- dist (folder)
    |-- node_modules (folder)
    |-- gulpfile.js
    |-- index.html
    |-- package-lock.json
    |-- package.json
```

Next, let's build our Sass workflow!

SECTION 5

Sass workflow

For our Sass files, we'll walk through how to do the following:

- Compile multiple Sass files into one final CSS file,
- Create sourcemaps to identify the original Sass file that a style is using in the browser,
- Autoprefix styles that need it for older browser support,
- Combine multiple media queries into one, and
- Minify the CSS file,

We'll also address how to add vendor Sass files to the workflow from your local files or importing from your `node_modules` folder.

SET UP SASS FILES

When working with Sass files, the first thing we need is to create our main Sass file. In the `app/scss` folder, make a new file called `style.scss`.

Then, let's create a few partial Sass files in the same location: `_variables.scss`, `_global.scss`, and `_homepage.scss`. These partial files get their name because they each contain just a part of your overall styles. It's a great feature of Sass that lets you organize your styles in these separate files. All the partial Sass files will ultimately get imported into the main Sass file.

One note, make sure that you begin each file name of the partials with an underscore. The underscore allows us to just use the file name without the `.scss` extension when importing.

Going back to the main `style.scss` file, we can now import our partials. And we only need to write the filename without the underscore or file extension. So it should look like this:

```
@import "variables";
@import "global";
@import "homepage";
```

Now let's add some styles that we will use to test our workflow.

In `_variables.scss`, we'll create some Sass variables to store colors. This will help us test that the Sass files are getting compiled to CSS correctly:

```
$white: #ffffff;
$lightGray: #cecece;
$medGray: #606060;
$darkGray: #303030;
$black: #000000;
```

Note that we want to import our variables before the other partials in our main Sass file. That way, we can use the variables in our other Sass files.

Then, in `_global.scss`, we'll add the following:

```
html {
    box-sizing: border-box;
    font-size: 100%;
}

*, *:before, *:after {
    box-sizing: inherit;
}

body {
    background-color: $darkGray;
    font-family: Arial, Helvetica, sans-serif;
    margin: 20px;
}

h1, h2, h3 {
    margin-top: 0px;
}

h1 {
    font-size: 4rem;

    @media (min-width: 64em){
        font-size: 5rem;
    }
}

h2 {
    font-size: 3rem;

    @media (min-width: 64em){
        font-size: 4rem;
    }
}

h3 {
    font-size: 2.5rem;

    @media (min-width: 64em){
        font-size: 3rem;
    }
}

.white {
    color: $white;
}
```

```
.dkGray {
    color: $darkGray;
}
```

In `_homepage.scss`, we'll add some basic styles. We'll include some Sass nesting and CSS properties (`transition`, `transform`) that need prefixes for some browsers. This will let us test Autoprefixer, a tool that will add `-webkit` and other browser prefixes to those properties.

And we have some media queries to test a PostCSS plugin called PostCSS Combine Media Query (We'll get more into those plugins in the next section.)

```
.content {
    display: grid;
    grid-template-columns: 1fr;
    grid-gap: 20px;
    margin: 40px 0;

    @media (min-width: 64em){
        grid-template-columns: repeat(2, 1fr);
    }

    &__block {
        background-color: $white;
        color: $darkGray;
        padding: 20px;
        border-radius: 10px;
        transition: all 200ms ease-in-out;

        &:hover {
            transform: translateY(-20px);
        }
    }
}
```

We want the content blocks to use CSS grid, so we've added grid properties to the `.content` parent element. Then we're use Sass nesting for the `.content__block` element styles.

These styles are pretty bare bones, but they're mainly for testing that the workflow is doing everything that it's supposed to.

INSTALL SASS NPM PACKAGES

Once we have our Sass files set up, we'll install npm packages for our Sass workflow and import them into our gulpfile.

Let's open our command line, and type in the following:

```
npm install gulp-sass gulp-postcss autoprefixer postcss-combine-media-query cssnano
```

It might take a few minutes to install all those packages. Also, if you happen to get a security error after installing, you can run `npm audit fix` to take care of security vulnerabilities in any of the packages.

Once everything is finished, let's go into our gulpfile and import those packages as modules. Under the line that mentions `require('gulp')`, we will load each package as a constant like this:

```
const sass = require('gulp-sass');
const postcss = require('gulp-postcss');
const autoprefixer = require('autoprefixer');
const combinemq = require('postcss-combine-media-query');
const cssnano = require('cssnano');
```

And don't worry-- we'll be going over what each of these packages does in the next section.

Now we are ready to write the functions for our Sass task!

CREATE A SASS TASK IN GULP

Now that we have our packages set up for our Sass workflow, let's write a function that will process our Sass files.

Below all the `require()` statements, we'll make a new function called `scssTask()`. This function will be where we write all the functionality to process our Sass files into CSS.

Here's what our Sass task function looks like:

```
function scssTask(){
    return src('app/scss/style.scss', { sourcemaps: true })
        .pipe(sass())
        .pipe(postcss([autoprefixer(), combinemq(), cssnano()]))
        .pipe(dest('dist', { sourcemaps: '.' })
    );
}
```

Now let's take it apart and go through what each line does.

First, we begin with a `return` statement and the `src()` method. We want to read our main Sass file, so the method will have one parameter, which is the location of the main Sass file, `app/scss/style.scss`.

We're also using an option in the `src()` method to turn on sourcemaps. This will generate a sourcemaps file to go along with our final CSS file. The sourcemap lets us trace a style rule in the browser back to the original `*.scss` file, which helps a lot with debugging.

After the `src()` method, we are going to add on other methods from the plugins we installed. In order to this, we have to use the `pipe()` method to chain them one after the other.

The first one we'll use after `src()` is the `gulp-sass` plugin. We can then call the `sass` constant we created at the top and run it as a function. This will compile our Sass files to CSS files.

Next, we'll pipe on the remaining plugins, which are all PostCSS plugins.

PostCSS is a tool that comes with a collection of plugins to process CSS files. To run those plugins, you can add them as parameters to the `postcss()` method. Autoprefixer, PostCSS Combine Media Query, and CSS Nano are the PostCSS plugins that we are using.

Autoprefixer adds browser prefixes to the CSS properties that need it. Prefixed properties, such as `transform`, are newer CSS properties that don't have complete support across the board, especially in older browsers. Each browser has its own prefix which allows it to decide how to support those properties. The most common prefixes are `-webkit-` for Chrome and Safari, `-moz-` for Firefox, and `-ms-` for Internet Explorer.

Over time, the properties that become fully supported will no longer require the prefix, such as `border-radius`. You can check what properties still need prefixes at ShouldIPrefix.com.

PostCSS Combine Media Query is a PostCSS plugin that will combine styles in duplicate media queries under one media query.

CSS Nano is used to minify, or optimize your CSS files, by removing extra whitespace and using abbreviated values when possible (such as `#fff` instead of `#ffffff` for the hex color white).

To run all these plugins, we need to run an initial `postcss()` method and then run each plugin as a parameter in it.

Finally, we'll save our final CSS file using the `dest()` method. The parameter for `dest()` is the file location, the `dist` folder. We also need to finish the sourcemaps process by using the option for `sourcemaps`. The value of `sourcemaps` is `'.'` to save the sourcemaps file in the same location as the final CSS file, in the `dist` folder.

AUTOPREFIXER AND BROWSERSLIST

One other thing we need to configure is related to the Autoprefixer plugin that we're using. We can specify which specific browser versions we want to support. The reason for this is that the older the browsers we support, the more properties that will require prefixes.

To configure what versions we want Autoprefixer to target, we will use a Node.js config tool called Browserslist. To use Browserslist, we'll have to create a new file called `.browserslistrc` in our project's root directory.

Open up that `.browserslistrc` file, and in it add the following config settings that Browserslist recommends:

```
# Browsers that we support

last 1 version
> 1%
IE 10 # sorry
```

Now Autoprefixer will know exactly which versions to target.

> Note: You can also set the browserslist config in your [package.json](), as well as in an option of the script itself. However, we'll be reusing these settings when we run Babel in our JavaScript task later on, so saving them in the `.browserslistrc` file allows both plugins to use the same settings.

CHECK THAT THE SASS TASK IS WORKING

Now that we've created our complete Sass task, let's test it out and make sure everything is working correctly.

To do that, we need to export our default Gulp task so we can run it on the command line. To do that, in our `gulpfile.js` at the bottom, add the following:

```
exports.default = scssTask;
```

This line will export the `default` Gulp task, and when we run it, we want to run our `scssTask()` function.

Going to our command line, type in `gulp` and check that everything runs with no errors.

We also want to check that the files we want have been generated correctly. First check the `dist` folder and make sure it contains a `style.css` file and a `style.css.map` sourcemaps file now.

Then, open the `style.css` file and check that it has very long lines. Most of the CSS should be all on one line. This means it has been minified correctly. Also in the `style.css` file, check that there exists a `-webkit-transform` property, that should have been added by Autoprefixer.

And lastly, check that there is only one `@media` at-rule that contains the responsive styles for our `h1`, `h2`, and `h3` tags.

If your Gulp ran with no errors, and all the files and styles look good, then that means your Sass task is working!

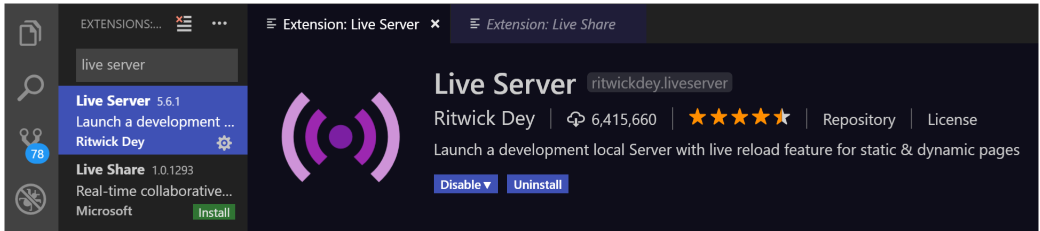
LOAD THE WEBSITE WITH A LOCAL SERVER

Now that we've confirmed the code seems correct, we'll want to load our local website to check that the website itself looks good. There are a couple ways you can do this.

If you are using VS Code, you can install the Live Server extension which will start and run a localhost server for you. It's super handy!

To install it, in VS Code go to the Extensions panel. You can click the extensions symbol in the left sidebar, or navigate to `File > Preferences > Extensions`.

In the Extensions menu, type in "live server" into the search bar and you should see something like this:



Look for the "Install" button on the right side and click to install the extension. (Mine says "Uninstall" since I already have it installed)

Once the extension is installed, in your VS Code look in the blue bar at the bottom of the editor and you should see something saying "Go Live":

If you don't see it, try closing and reopening VS Code. Once you do see it, click "Go Live" and VS Code will start a local server and automatically open a new browser tab loading the localhost server at `http://127.0.0.1:5500/`

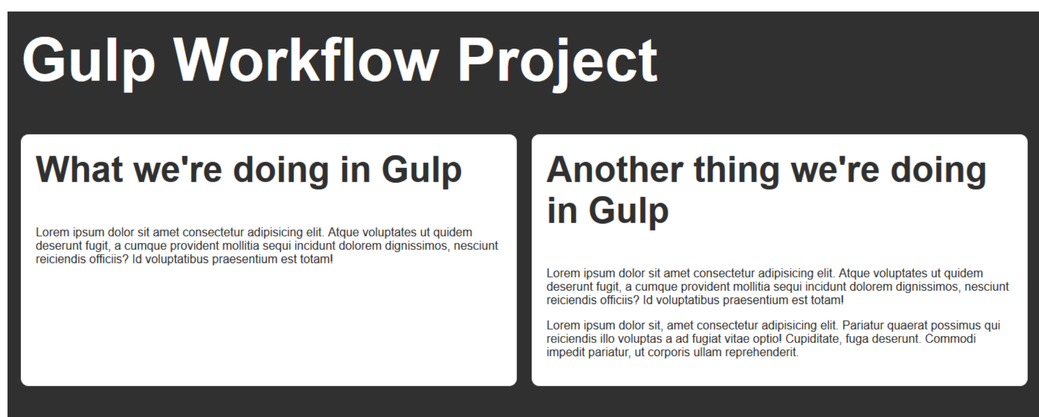
If you aren't using VS Code, you can run a local server by installing the `http-server` npm package.

In your command line, run `npm install http-server -g` to install HTTP Server globally on your computer. Then, making sure you are in your project root file, run `http-server`. You should see a message like this:

```
PS C:\Users\Jessica\Documents\GitHub\gulp-for-beginners> http-server
Starting up http-server, serving ./
Available on:
  http://192.168.1.103:8080
  http://127.0.0.1:8080
Hit CTRL-C to stop the server
```

And you'll be able to load the local website by loading one of those URLs listed or `http://localhost:8080/`.

Whichever method you choose to use, once you have loaded the local website it will hopefully look something like this:



Later on, we'll be using Browsersync to load and update a local server. But this more manual way is just as good, although it won't refresh automatically like Browsersync will.

WORKING WITH VENDOR SASS & CSS FILES

We've gone over how to include local Sass files into your workflow, but what about vendor or third party files?

Since this is a very common scenario, we're going to look at two different ways that you might want to include vendor files:

- Manually copying them over to your project files, and
- Installing them via an npm package.

We'll be adding the free Font Awesome library to our project in order to illustrate this approach.

In a nutshell, how Font Awesome works is that it's a custom web font that works with CSS styles. You can load an icon by using either an <i> or a <span> element with a specific Font Awesome class for that icon.

Then in the CSS styles, each icon will load a pseudo element with a `content` value that is mapped to a symbol in the Font Awesome font. This will then load the icon image that you see.

There are two components of Font Awesome that we will need to load:

- The font file for their custom font, and
- The Sass files for their styles.

> Note: This example will load Font Awesome files locally, not using their CDN (Content Delivery Network). CDNs can be quicker to set up since you're not hosting the files yourself, but if you want to customize their styles or are concerned with downtime, hosting their files locally is always an option.

ADDING FONT AWESOME MANUALLY

We're going to first add Font Awesome manually to our project. Meaning, we will get their font files and Sass files, add them into our own project files, and incorporate them into our Gulp workflow.

To do this, we will first download the zip file from the website, and then add the files that we need into our project files. Font Awesome does come with Sass files, so we'll be able to add them to our Gulp workflow the same way as our local Sass files.

According to their directions for setting up with Sass, we need to copy the `scss` and `webfonts` Font Awesome folders into our project.

I like to keep my plugins all together, so we'll create a new `plugins` subfolder in the root, create a `fontawesome` subfolder, and add those two Font Awesome folders there.

Our project structure will then look like this:

```
(project folder)
    |-- app (folder)
    |-- dist (folder)
    |-- node_modules (folder)
    |-- plugins (folder)
        |-- fontawesome (folder)
            |-- scss (folder)
            |-- webfonts (folder)
    |-- gulpfile.js
    |-- index.html
    |-- package-lock.json
    |-- package.json
```

Then, according to the directions, we need to create a Sass variable in our Sass files called `$fa-font-path` that points to the `webfonts` folder. In our `scss/_variables.scss` file I've added it here like this:

```
$fa-font-path: "../../plugins/fontawesome/webfonts";
```

And lastly, we need to import the Font Awesome Sass files in our main `style.scss` file. For basic usage, we want to import the main file (`fontawesome.scss`) and a theme file (`solid.scss`).

In our main `style.scss` file, we will import them like this:

```
@import "variables";
@import "global";
@import "homepage";

// Font Awesome
@import "../../plugins/fontawesome/scss/fontawesome.scss";
@import "../../plugins/fontawesome/scss/solid.scss";
```

You can import your Sass files in a different order, depending on your preferences. Just make sure to import the Font Awesome Sass files after you import the variables Sass file.

This is because the `$fa-font-path` variable needs to be defined in our variables file before it is called in the Font Awesome Sass files. Otherwise you will get an error when you compile your Sass files.

In addition, you need to make sure that the paths to the Font Awesome Sass files in your main Sass file are correct. Let's trace those paths to double-check that we have them right.

Our main Sass file is in the `app/scss` folder, and the Font Awesome files are in the `plugins/fontawesome/scss` folder. Here's a quick tree structure of where our main Sass file and the Font Awesome Sass files are in relation to one another:

```
(project folder)
    |-- app (folder)
         |-- scss (folder)
            |-- style.scss
    |-- plugins (folder)
        |-- fontawesome (folder)
            |-- scss (folder)
                |-- fontawesome.scss
                |-- solid.scss
```

All the import statements are in the `app/scss/style.scss` file, so we need to navigate from there up two levels through the `scss` and `app` folder to reach the root directory. Then we can navigate to the `plugins/fontawesome/scss` folder.

To navigate up the tree to parent folders, we can use the `../` symbols. The forward slash (`/`) will navigate up one level, and the two dots (`..`) signify the folder itself.

So, starting from the `style.scss` file in the `app/scss` folder, the first `../` signifier will get us one level up out of the `scss` folder and into the `app` folder. Then the second `../` will get us one more level up and into the project root folder.

From the root we can then navigate to the `plugins/fontawesome/scss` folder and refer to the actual file names we're that loading.

Now that we've added our files and references, let's test it out!

First, let's go back into our index.html and add a couple of Font Awesome icons into our markup. You can browse through the free icons, and click on the one you'd like to add.

I'm going to pick the "open box" icon here, and copy the markup: <i class="fas fa-box-open"></i>.

Then I'll paste it into my index.html in one of the <h3> tags like this:

```
<h3><i class="fas fa-box-open"></i> What we're doing in Gulp</h3>
```

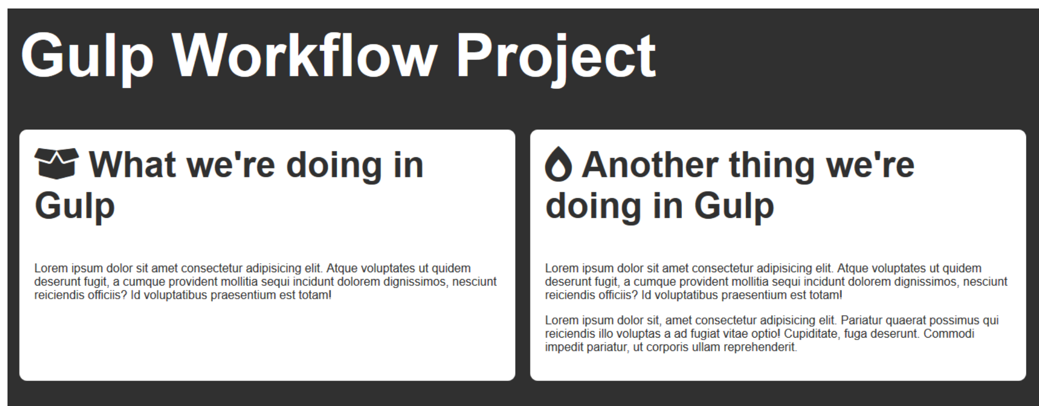
And just for fun, I'll get another icon and paste it in the other <h3>:

```
<h3><i class="fas fa-burn"></i> Another thing we're doing in Gulp</h3>
```

Next, we want to make sure that there are no errors when we run our Gulp task. Since we added the Font Awesome files into our main Sass file, we don't need to make any changes in our gulpfile.

Type in gulp in your command line, and if there are no errors, load your local website in the browser and check!

You should be able to see the icons that we added, like this:



If everything loads correctly with no errors, then everything is working!

If you get this error in Gulp: `File to import not found or unreadable...`:

Your `@import` statements for the Font Awesome Sass files in your main Sass file are probably incorrect. Double-check your file path and make sure it matches what's in your project file structure.

If you get no errors in Gulp, but the icons don't load in your website:

You may have an incorrect path for the `webfonts` folder where you defined the `$fa-font-path` variable. You can confirm if this is the case by opening up your Developer Inspection tools in your browser and checking if you have any browser console errors.

They might look something like this (I'm using Firefox):

```
downloadable font: download failed (font-family: "Font Awesome 5 Free"
style:normal weight:900 stretch:100 src index:1): status=2147746065
source: http://localhost:8080/node_moduleszzz/@fortawesome/fontawesome-
free/webfonts/fa-solid-900.woff2
```

If this is the case, go back to your Sass file where you defined that `$fa-font-path` variable and double-check that the path matches your file structure.

ADDING FONT AWESOME VIA NPM

Now let's load Font Awesome by installing the npm package. This is a pretty common way of installing plugins, so I think it's a good skill for you to know.

According to the instructions in the Font Awesome documentation, to install it, open your command line and run `npm install @fortawesome/fontawesome-free`. You'll then see a new `@fortawesome/fontawesome-free` subfolder in your `node_modules` folder.

Note that Font Awesome Free package is listed under the Fort Awesome scope. This scope is the company, Fort Awesome, that has published multiple packages. So if you're a company or even an individual who has multiple related packages, a scope is a way to namespace your npm packages and so keep them all together.

To import the Font Awesome files from our `node_modules` folder, we're going to take advantage of a cool feature in Sass called `includePaths`.

How this works is, the `sass()` method from `gulp-sass` that we are using in our Gulp workflow has an option called `includePaths`. This option allows you to load paths from other locations and make them accessible to the `sass()` function.

What we want to do here is to use the `includePaths` option to load the `node_modules/@fortawesome` file path.

To do this, in your gulpfile, go to the `scssTask()` function and look for the line that says `.pipe(sass())`. Then add the `includePaths` option and load the Fort Awesome path like this:

```
.pipe(sass({
    includePaths: 'node_modules/@fortawesome'
}))
```

This makes the `node_modules/@fortawesome` folder accessible to Sass `@import` statements. So now in our main Sass file, we can use more simplified import statements:

And update the main Sass file so the Font Awesome Sass files are loaded from that location:

```
@import "fontawesome-free/scss/fontawesome.scss";
@import "fontawesome-free/scss/solid.scss";
```

Unfortunately, Sass variables can't utilize `includePaths`, so in your variables Sass file you'll have to load the complete path for `$fa-font-path` like this:

```
$fa-font-path: "../../node_modules/@fortawesome/fontawesome-free/webfonts";
```

Once you have your paths set, run `gulp` to make sure you don't have any errors, and then load the local website to check that the icons are loading.

So that's it for how to add local and vendor Sass files into your Gulp workflow!

Let's move on to the next section, and set up our JavaScript workflow.

SECTION 6

JavaScript workflow

In this section, we're going to be building out our JavaScript workflow.

For JavaScript files, we'll walk through how to do the following:

- Concatenate multiple JavaScript files into one big JavaScript file,
- Transpile ES6 syntax into an older ES5 format to support older browsers,
- Bundle and import Node resources called by `require()` functions, and
- Uglify (minify) our final JavaScript file.

BASIC JAVASCRIPT WORKFLOW

Let's first set up some initial JavaScript files that we will then add into our Gulp workflow.

In Gulp, we will combine those multiple files into one final JavaScript file, and then uglify it.

These are the basic steps in our JavaScript workflow, and we'll add on the other steps we mentioned, later on to fill it out.

In our `app/js` folder, let's some new files called `script.js` and `script2.js`.

In the `script.js` file, we're going to add in some JavaScript code:

```
var testLocal = "Here's a local script file!";
console.log(testLocal);
```

This is pretty simple for right now, because we just want to test that our JavaScript code will run in the browser.

Then in the `script2.js` file, add in this code:

```
console.log("Here's another script file!");
```

This is just so we have multiple JavaScript files that we will load in Gulp.

Now that we have our initial JavaScript files created, let's install some npm packages we need for this basic JavaScript task:

In the command line, run the following:

```
npm install gulp-concat gulp-uglify
```

And in our gulpfile, we'll import those packages as modules by creating new constants:

```
const concat = require('gulp-concat');
const uglify = require('gulp-uglify');
```

CREATE A JAVASCRIPT GULP TASK

Now, just like we did with our Sass workflow, we'll create a function for our JavaScript workflow.

In your gulpfile, after the `scssTask()` function, create a new function called `jsTask()` with the following code:

```
function jsTask(){
    return src(['app/js/script.js', 'app/js/script2.js'], { sourcemaps: true })
        .pipe(concat('scripts.js'))
        .pipe(uglify())
        .pipe(dest('dist', { sourcemaps: '.' })
    );
}
```

You might have seen some familiar methods here-- like `src()`, `pipe()`, and `dest()`-- the same ones that we used in our Sass task.

The `jsTask()` function is using `src()` to read the two JavaScript files. Notice that the `src()` method contains square brackets (`[]`) to indicate an array containing the two files.

We're also going to turn on the sourcemaps option to create an external sourcemaps file for JavaScript.

Then we're using `pipe()` to add on a new method, `concat()`. This is loading the `gulp-concat` module, which will concatenate the files in the stream. This means that multiple files will be combined into one long JavaScript file, one after the other.

In our case, `script.js` will be added first, then `script2.js` will be added after that. And the parameter in `concat()`, `scripts.js` is what the final combined file will be called.

Once the file is concatenated, we will use the `uglify()` method to uglify (or minify) the JavaScript file, using `gulp-uglify`.

And the last step piped on will use the `dest()` method to save that final JavaScript file and the sourcemaps file both in the `dist` folder.

CHECK THAT THE JAVASCRIPT TASK IS WORKING

Let's now add the `jsTask()` to our default Gulp task, so that it will be executed when typing in `gulp` on the command line.

At the bottom of our gulpfile, where it says `exports.default`, we want to run the `jsTask()` function after the `scssTask()` function. But since we have multiple functions that we're running, we have to use either the `series()` or `parallel()` method to tell Gulp which order to run the functions in. If you don't, Gulp will throw an error.

We'll use a `series()` method, and our default task will look like this:

```
exports.default = series(scssTask, jsTask);
```

Now when running the default Gulp task, it will run the `scssTask`, then the `jsTask`.

To test it out, type in `gulp` in your command line, and if there are no errors, you should see a message like this:

```
PS C:\Users\Jessica\Documents\GitHub\gulp-for-beginners> gulp
[12:48:35] Using gulpfile ~\Documents\GitHub\gulp-for-beginners\gulpfile.js
[12:48:35] Starting 'default'...
[12:48:35] Starting 'scssTask'...
[12:48:36] Finished 'scssTask' after 1.3 s
[12:48:36] Starting 'jsTask'...
[12:48:36] Finished 'jsTask' after 46 ms
[12:48:36] Finished 'default' after 1.35 s
```

Then let's check the `dist` folder to make sure that the final `scripts.js` file and the `scripts.js.map` sourcemaps file both exist, along with the final CSS files.

We also want to open the `scripts.js` file to make sure that the code from both `script.js` and `script2.js` exist in it, and that the code is uglified and compressed onto one line.

If you have no errors, and the files are all there, then the JavaScript task is working!

Lastly, we want to load the local website and check our console to make sure that both our `console.log` statements have printed out there. If they are showing up, then that means everything is working the way we wanted to.

> **Note:** At this point, we need our JavaScript to use ES5 syntax, because the `gulp-uglify` plugin is not ES6-compatible. It will throw an error if you add ES6 syntax. But don't worry, we will take care of handling ES6 syntax with Babel in the next section.

This basic workflow is good for regular vanilla JavaScript files. But there are some common cases where you will need some extra functionality. Let's take a look at some of these now.

RUN BABEL FOR ES6 SYNTAX

In this section, we're going to use the Babel plugin to convert ES6 syntax to ES5.

ES5 or ECMAScript 5, is an older version of JavaScript, and ES6 is the newer version, released in 2015. Not all browsers support the newer ES6 syntax, specifically Internet Explorer. So if you need to support older browsers, you will need to transpile (or convert) your JavaScript files that are using ES6 down to ES5.

One simple example of ES6 vs ES5 syntax is in creating variables. In ES6 you can use `constant` and `let` to create variables. However in ES5 you can't use those, and all variables need to be created as `var`.

To set up Babel, we need to install the `gulp-babel` plugin which allows Babel to work with Gulp. We also need to install some important components of the `@babel` scope:

- `@babel/core` for core functionality, and
- `@babel/preset-env` which works with the environments targeted in our `.browserslistrc` file.

If you remember from the Sass section, we created the `.browserslistrc` file when working with Autoprefixer. Babel can also use this file when targeting browser versions.

To install these packages, run this in the command line:

```
npm install gulp-babel @babel/core @babel/preset-env
```

Then in our gulpfile we will import Babel as a constant:

```
const babel = require('gulp-babel');
```

We don't need to create constants for the other two Babel packages we installed, as `gulp-babel` will automatically look for them.

Now we can add the `babel()` method to our `jsTask()` function. We're going to run Babel after we concatenate our JavaScript files:

```
function jsTask(){
    return src(['app/js/script.js', 'app/js/script2.js'], { sourcemaps: true })
        .pipe(concat('scripts.js'))
        .pipe(babel({ presets: ['@babel/preset-env'] }))
        .pipe(uglify())
        .pipe(dest('dist', { sourcemaps: '.' })
    );
}
```

In the `babel()` method, we're using the `presets` option to load the environment settings. Gulp will then search for the `.browserslistrc` file and if it exists, use the settings in it.

Before we test our default `gulp` task, let's update our `script.js` file to use some ES6 syntax. In the first line change `var testLocal` to `const testLocal`:

```
const testLocal = "Here's a local script file!";
console.log(testLocal);
```

Now when we run our default `gulp` task, Gulp should concatenate the JavaScript files, then transpile our ES6 syntax to ES5, and save the final file in the `dist` folder as `scripts.js`.

If all goes well, you should have no errors. And when we open up the final `dist/scripts.js` file, we can see at the beginning of the file that the code for `const testLocal` has been converted back to the ES5-compatible `var testLocal` syntax.

BUNDLING MODULES WITH BROWSERIFY

In this section, we're going to use Browserify to bundle any required Node modules in with our JavaScript files.

At the beginning of this course we talked about using npm packages, and how you can install and store them in your `node_modules` folder. This is what we've done with all the packages we need for Gulp. And in our gulpfile, we can import those packages by using the `require()` function.

However, while we can use `require()` in Node applications like in Gulp, you can't directly use it in client-side JavaScript for the browser. If you do attempt to use `require()` in your JavaScript code, the browser will just throw an error, saying `require is not defined`.

The good news though, is that you can use bundler tools like Browserify to bundle imported npm modules in with your script. That way it's accessible to the browser.

To illustrate this, we're going to use the Camelcase package mentioned earlier in this course. If you don't have it installed yet, you can do so by running `npm install camelcase` on the command line. And again, Camelcase is a simple package that will take any string and capitalize it according to the camel case pattern.

Now, let's add the `require()` method into our JavaScript. Open up the `script.js` file, and import the camelCase module like this:

```
const camelCase = require('camelcase');
```

(Notice that this is the same way we've imported all our Gulp modules in our gulpfile.)

Under where you imported camelCase, let's add some code that will use the `camelCase()` function:

```
const testVar = "testing camelcase in js";
console.log(camelCase(testVar));
```

Now that we have our JavaScript file with the `require()` function, let's set up Browserify.

We'll be installing Browserify and some Vinyl-related packages to make it compatible with Gulp. As we mentioned earlier, Vinyl objects are how Gulp manages the files that it is processing in the stream.

- **browserify** -- bundles the files
- **vinyl-source-stream** -- converts the Browserify text output into a Vinyl object to make it compatible with Gulp
- **vinyl-buffer** -- buffers the stream so more Gulp methods can be used afterwards

The reason we need these Vinyl packages is because Browserify itself is not a Gulp-specific package. Previously in the course we had mentioned how Gulp uses Node streams in object mode, and other plugins like Browserify use Node streams in buffer mode. And those two modes are not directly compatible.

So we need to make the files generated by Browserify in the stream compatible with Gulp using those two Vinyl packages. You can think about them like an adapter that lets you connect two different types of cables.

To install these packages, run this in your command line:

```
npm install browserify vinyl-source-stream vinyl-buffer
```

Once they're installed, we can import them as modules in your gulpfile:

```
const browserify = require('browserify');
const source = require('vinyl-source-stream');
const buffer = require('vinyl-buffer');
```

Now there is one catch when using Browserify, and that's that the `browserify()` method will only take a single file in its parameter. Since you may very well have multiple files that need bundling, we will need to concatenate all our JavaScript files before running them through Browserify.

I'm going to actually create a new separate function in Gulp just to concatenate. Then we'll take that one concatenated JavaScript file and run it through `browserify()`, also in its own function. And lastly, we'll run all of our other JavaScript tasks left in the `jsTask()` function.

Let's set this up. First, in our `jsTask()` function, we're going to remove the `concat()` method and put it in its own function, called `concatJsTask()`. It should look like this:

```
function concatJsTask(){
    return src('app/js/*.js')
        .pipe(concat('scripts.js'))
        .pipe(dest('app/js'));
}
```

So we are concatenating all the JavaScript files in our `app/js` folder into a combined file called `scripts.js`, but we're going to keep that combined file in the `app/js` folder. (Not in the `dist` folder like before.)

Then to bundle, we'll create another new function called `browserifyTask()` and in it we will run the following:

```
function browserifyTask(){
    return browserify('app/js/scripts.js')
        .bundle()
        .pipe(source('scripts.js'))
        .pipe(buffer())
        .pipe(dest('app/js'));
}
```

In our `browserifyTask()`, we are running Browserify on our concatenated `scripts.js` file. Then we are starting the bundling, setting the `scripts.js` file as the final bundled filename, buffering it, then saving the bundled file again in the `app/js` folder.

Note that this file will overwrite the `scripts.js` file that we created in the previous `concatJsTask()`. I'm purposely overwriting it so that we have fewer files to worry about.

Lastly, we will run the rest of our JavaScript functionality in the `jsTask()` function, which will now look like this:

```
function jsTask(){
    return src('app/js/scripts.js', { sourcemaps: true })
        .pipe(babel({ presets: ['@babel/preset-env'] }))
        .pipe(uglify())
        .pipe(dest('dist', { sourcemaps: '.' }));
}
```

So we are taking the `scripts.js` file generated by Browserify, and run it through our normal JavaScript workflow. We're generating a sourcemaps, running it through Babel, uglifying it, and finally saving the final `scripts.js` file in the `dist` folder.

Now we need to update our default Gulp task so that it runs the `concatJsTask()`, `browserifyTask()`, and the `jsTask()` functions, in that order:

```
exports.default = series(
    scssTask,
    concatJsTask,
    browserifyTask,
    jsTask
);
```

Now, let's check that you can run `gulp` with no errors. Then we want to check the `dist` folder and make sure that the `scripts.js` and `scripts.js.map` files have been generated.

Once we've confirmed that they are there, we need to open up the `dist/scripts.js` file and make sure that it contains the `camelCase` code that has been bundled from the camelCase module.

You should see much more code than you had before, as well as a mention of `var o=e("camelcase");`. All this extra code is the imported code from camelCase.

And lastly, we need to open the local website and check our browser console. If camelCase was successfully imported, we should see that console message that we created using camelCase.

Once you have all that working, you've successfully set up Browserify!

CLEANING UNNEEDED FILES

There's one more thing that we want to add to our JavaScript workflow. If you remember, the `concatJsTask()` and `browserifyTask()` generate the `scripts.js` file in the `app/js` folder.

We don't need that file once Gulp is finished running, and keeping it around could cause potential confusion. So it's better to clean (or delete) that file after Gulp is done.

To do this, we'll use an npm package called Del. You can install it by running `npm install del` on the command line. Then import it into your gulpfile by adding the line:

```
const del = require('del');
```

Then we will create another function in the gulpfile called `cleanTask()` like this:

```
function cleanTask(){
    return del(['app/js/scripts.js']);
}
```

The function only does one thing, deletes the `scripts.js` file from the `app/js` folder! It's a small but still important function.

And of course, we'll add `cleanTask` to the end of the default task in Gulp like this:

```
exports.default = series(
    scssTask,
    concatJsTask,
    browserifyTask,
    jsTask,
    cleanTask
);
```

Now, after we run `gulp` we should be able to see that there is no more `scripts.js` file in the `app/js` folder.

And that's it for the JavaScript section of this tutorial! Let's move on to some other processes we want to set up in Gulp.

SECTION 7

Utility Gulp functions

In this section, we're going to cover some utility functions that I've found pretty helpful in my Gulp workflow.

IMAGE MINIFICATION

The first one deals with minifying images. We've worked on optimizing our CSS and JavaScript files. And we want to do the same thing for our image files, whether they are JPG, PNG, GIF, or SVG images.

First, let's create a new folder called `img` in our project root to store image files. After creating it, your project structure should look like this:

```
(project folder)
    |-- app (folder)
    |-- dist (folder)
    |-- img (folder)
    |-- node_modules (folder)
    |-- plugins (folder)
    |-- gulpfile.js
    |-- index.html
    |-- package-lock.json
    |-- package.json
```

Then we'll download an image to test this out. I'm using a JPG from Unsplash by JR Korpa, but you can use any image you like! Once you've downloaded the image, put it in the `img` folder.

Next, go into your code editor and install the `gulp-imagemin` package by running `npm install gulp-imagemin` in your command line.

Then in your gulpfile, import the package as a module:

```
const imagemin = require('gulp-imagemin');
```

And we'll create a new function to process our images, using the recommended options :

```
function imageminTask(){
    return src('img/*')
        .pipe(imagemin([
                imagemin.gifsicle({interlaced: true}),
                imagemin.mozjpeg({quality: 75, progressive: true}),
                imagemin.optipng({optimizationLevel: 5}),
                imagemin.svgo({
                    plugins: [
                        {removeViewBox: true},
                        {cleanupIDs: false}
                    ]
                })
            ],
            { verbose: true }
        ))
        .pipe(dest('img'));
}
```

In our `imageminTask()`, we are reading any files in the `img` folder by using the wildcard (`*`) signifier. Then we are piping on the `imagemin()` method along with the options for each file type.

We also are using the `verbose: true` option so Gulp will display information about the compressed files. And finally, the minified files will be saved in the same folder.

Then we need to add the `imageminTask()` to our default Gulp task:

```
exports.default = series(
    scssTask,
    concatJsTask,
    browserifyTask,
    jsTask,
    cleanTask,
    imageminTask
);
```

Now that we've created our `imageminTask()` function, let's add the image itself to our website. Let's load the image as a background image for our `<body>` tag.

Open up the `app/scss/_global.scss` file, and update the styles so it looks like this:

```
body {
    background-color: $darkGray;
    background-image: url('/img/jr-korpa-stwHyPWNtbI-unsplash.jpg');
    background-size: cover;
    background-repeat: no-repeat;
    font-family: Arial, sans-serif;
    margin: 20px;
}
```

We'll leave the `background-color` in place as a default. But we're also adding the `background-image` and some other background-related properties so the image will fill the entire space.

Let's run Gulp and check if the minification works! In the command line, run `gulp` and check the output.

If everything worked without an error, you should see something like this:

```
[13:31:42] Starting 'imageminTask'...
[13:31:42] gulp-imagemin: ✓ jr-korpa-stwHyPWNtbI-unsplash.jpg (saved 44.5 kB - 5.3%)
[13:31:42] gulp-imagemin: Minified 1 image (saved 44.5 kB - 5.3%)
```

And with the `verbose` setting we turned on, we can see the `gulp-imagemin` messages saying how much was compressed!

One neat feature of imagemin is that once you've minified an image once, it will remember. So the next time you run Gulp, it will skip that file as you can see here:

```
[13:32:35] Starting 'imageminTask'...
[13:32:35] gulp-imagemin: ✓ jr-korpa-stwHyPWNtbI-unsplash.jpg (already optimized)
[13:32:35] gulp-imagemin: Minified 0 images
[13:32:35] Finished 'imageminTask' after 398 ms
```

Lastly, load your website in the browser and check that the background image is loaded:

CACHE BUSTING

In this section, we're going to add cachebusting to our Gulp workflow.

First off, what is caching exactly?

Well, when you load a website in your browser, the browser will save (or cache) copies of all the website files on to your local computer. This includes the HTML, CSS and JavaScript files, as well as other assets like images.

So the next time you go back to the same website, the browser won't have to re-download all those files from the server, but can instead load your local copies of the files. This makes loading the website much faster the second time around.

This is great for speed, but the downside is that if the website has changed since the last time you loaded it, sometimes your browser will load the old locally cached files instead of loading the new versions from the server.

Cachebusting forces the browser to reload specific files from the server. And in our case, we want to force reloading of the CSS and JavaScript files in our website.

So how exactly can we make the browser refresh those files? To do this, let's open up our `index.html` file. The CSS and JavaScript files are loaded in tags like this:

```
<link rel="stylesheet" href="dist/style.css">

<script src="dist/scripts.js"></script>
```

One way to force refreshing is to add a query string to the end of the file reference. A query string is a string that begins with a question mark (`?`) and often followed by sets of keys and values.

For example, the query string we're going to add to our references will look something like this: `?cb=123`. This query string has one key/value pair. The key is `cb` and the value of that key is `123`.

> **Note:** You can name the query string components anything you want. We're using `cb` as a reminder that it is related to cache busting. You can also use multiple components in one query string, as long as you separate them with an ampersand (`&`) symbol like this: `scripts.js?cb=123&version=test`. For our cache busting purposes, however, we only need one component.

So how does adding this string work to cache bust the files?

It all comes down to how your browser views files. Let's say your browser has downloaded and cached your original `style.css` file.

Then if you add the query string so that your CSS file changes to read as `style.css?cb=123`, your browser thinks that it's a completely different file and will download it from scratch. The query string doesn't affect the file itself at all, it's simply a tool that tricks your browser into thinking it's a new file that needs to be cached.

The catch here is that the `123` number needs to also change every time we run Gulp. That way the browser will continue to refresh the files as we make code changes.

So to make this work in our Gulp workflow, we need to accomplish a few things:

First, we need to update our CSS and JavaScript file references in our `index.html` file to contain the basic query string format.

So they should look like this:

```
<link rel="stylesheet" href="dist/style.css?cb=123">

<script src="dist/scripts.js?cb=123"></script>
```

The `cb` values are initially going to be hardcoded as `123` just to start off.

The second thing we need to do is to change those values to a different number every time Gulp runs. This will ensure that the browser will be forced to refresh our CSS and JavaScript files every time.

Lastly, once we have that number, we need Gulp to replace the `123` number in the `index.html` file with the new cache bust number. To do this, we will need to install an npm package called `gulp-replace`.

In the command line, run `npm install gulp-replace` and import it as a constant in the gulpfile:

```
const replace = require('gulp-replace');
```

Then we'll create a new function and also load our constant for the current Date Time:

```
function cacheBustTask(){
    let cbNumber = new Date().getTime();
    return src('index.html')
        .pipe(replace(/cb=\d+/g, 'cb=' + cbNumber))
        .pipe(dest('.'));
}
```

First off, inside the `cacheBustTask()` function, we are generating our unique number for the querystrings.

To generate the number, we will get the current Date Time using the JavaScript method `getTime()`. This will return the current date in milliseconds since midnight on January 1, 1971. It's usually a long thirteen-digit number. Then we are storing the number as a `let`.

You can use other methods of generating a number, random string, or a hash. I personally like the Date Time approach since we know the number will always be unique and won't repeat.

The `cacheBustTask()` function uses `src()` to read the `index.html` file, then pipes on the `replace()` method, and then uses `dest()` to put the file back in the same directory.

Let's take a closer look at the `replace()` method since it's a bit complex. It has two parameters separated by a comma: `/cb=\d+/g` and `'cb=' + cbNumber`.

The first parameter is a regular expression, also called "regex." Regular expressions are a way of writing patterns when searching for text matches in programming. The symbols may look intimidating at first, but let's break the string down into parts.

In the expression we're using, the forward slashes `/` at the beginning and end signifies that it is a regex string.

Then the `cb=` section means that we want to find text containing that exact string, `cb=`.

The `\d` will match any decimal digit (0-9), and the `+` will match if there are one or more digits. This means that `\d+` will match both `1` and `12345`. If we didn't have the `+`, it would only match for one-digit numbers.

Lastly, the `g` after the last `/` is a flag that when you turn it on, will run the search repeatedly over the entire text. Without the global flag, the regex search will stop after it gets one match.

Since we are using the cachebust string twice, once for the CSS and once for the JavaScript reference, we want to use a global search.

Putting it all together, the `replace()` method will look for strings starting with `cb=` and followed by a number that has one or more digits.

Then it will replace it with the string `cb=` and the new Date Time number.

> **Note:** Regular expressions can be quite confusing; there's a lot of syntax involved. But there are some great online tools that can help you create and debug your regular expressions, such as regex101.com. So you don't have to necessarily memorize all the regex syntax yourself!

Let's add the `cacheBustTask()` function to our default Gulp task:

```
exports.default = series(
    scssTask,
    concatJsTask,
    browserifyTask,
    jsTask,
    cleanTask,
    imageminTask,
    cacheBustTask
);
```

When we run `gulp`, after it complete, check the `index.html` file. If everything worked with no errors, we should see that the CSS and JavaScript references now have a different number from the original `123`!

```
<link rel="stylesheet" href="dist/style.css?cb=1574132114479">

<script src="dist/scripts.js?cb=1574132114479"></script>
```

(Your numbers will be slightly different from what we have here, but they should be long numbers of some sort!)

You can try running `gulp` again and checking to see that the numbers in the query strings change each time.

If everything seems good and the query string number keeps changing, we should be all set with our cache bust task!

SECTION 8
Optimizing Gulp

In this section, we're going to optimize our Gulp workflow in a few different ways.

First, we're going to set up Browsersync and a Gulp watch task to make local testing easier. Then we will export a production task for the functions that we only need to run once, right before deploying. These improvements will streamline our workflow and make it more efficient.

BROWSERSYNC

Let's start by setting up Browsersync.

Browsersync is a tool that will start a local server based on your website files, and automatically load it in your browser.

What we're going to do is have Gulp start the Browsersync server, and reload it whenever we make changes to our files. This will make things really convenient for testing, because you won't have to keep reloading the browser manually to see your changes.

To install Browsersync, in your command line run `npm install browser-sync`.

Then in your gulpfile, import the package as a constant:

```
const browsersync = require('browser-sync').create();
```

And instead of simply importing the `browser-sync` package, we're going to also run the Browsersync `create()` method. This will create the local server, and then we will run additional methods on top of it.

We need Browsersync to do two things: run the local server, and then reload the server anytime we make changes. To accomplish this, we will create two separate functions in our gulpfile related to Browsersync:

```
function browserSyncServe(cb){
    browsersync.init({
        server: {
            baseDir: '.'
        }
    });
    cb();
}

function browserSyncReload(cb){
    browsersync.reload();
    cb();
}
```

The `browserSyncServe()` function initializes the Browsersync local server with the `init()` method. And in the options we are setting the base directory (`baseDir`) of the local server to our root project directory using the `'.'` notation. This means it will detect and load the local site based off of our `index.html` file.

The second function, `browserSyncReload()`, can be only be run when the Browsersync server has already been initialized. This function will reload all the local website files.

Also note that both `browserSyncServe()` and `browserSyncReload()` are using the callback function `cb()`. As we mentioned at the beginning of this course, this is necessary to explicitly signify that the function is complete, since they are not returning a Gulp stream.

Now that we have our Browsersync functions set up, let's integrate them into our existing Gulp workflow.

We want to initialize the Browsersync server when we run Gulp, so let's add the `browserSyncServe()` function to our default Gulp task like this:

```
exports.default = series(
    scssTask,
    browserifyTask,
    jsTask,
    cleanTask,
    imageminTask,
    cacheBustTask,
    browserSyncServe
);
```

Now let's test and make sure the `browserSyncServe()` function is working by running `gulp` in our command line. If all goes well, you should see a similar message to this:

```
[13:40:05] Starting 'browserSyncServe'...
[13:40:05] Finished 'browserSyncServe' after 17 ms
[13:40:05] Finished 'default' after 2.44 s
[Browsersync] Access URLs:
 ------------------------------------
       Local: http://localhost:3000
    External: http://192.168.1.103:3000
 ------------------------------------
          UI: http://localhost:3001
 UI External: http://localhost:3001
 ------------------------------------
[Browsersync] Serving files from: .
```

And a new tab should pop up in your browser with the local site, from the URL `http://localhost:3000`.

So that takes care of setting up and initiating the local server. But what about the `browserSyncReload()` function?

When we run the default Gulp task, it only executes one time and then it's done. Ideally we would want to reload the browser every time we make a code change.

And in fact, we'd also want to rerun many of our other Gulp functions at the same time, like our Sass and JavaScript processes.

So to rerun all these functions, we'll need to take advantage of one of the built-in Gulp methods, and create a `watch()` task.

WATCH TASK

As we mentioned in the beginning of the course, the Gulp `watch()` method will watch files and then run tasks when it detects changes in those files.

Now there are a few different types of files that we want to watch. We want to watch and detect changes in our `index.html` file, our Sass and JavaScript files, and any image files in the `img` folder.

Let's create a `watchTask()` function and set up separate `watch()` methods to handle each case:

```
function watchTask(){
    watch('index.html', browserSyncReload);
    watch(
        [
            'app/scss/**/*.scss',
            'app/js/**/*.js',
            '!app/js/scripts.js'
        ],
        series(
            scssTask,
            concatJsTask,
            browserifyTask,
            jsTask,
            cleanTask,
            cacheBustTask,
            browserSyncReload
        )
    );
    watch('img/*', series(imageminTask, browserSyncReload));
}
```

In each case, the `watch()` method takes two parameters:

- The files Gulp will watch, and
- What functions to run after any changes are detected.

The first `watch()` method is pretty short. In it, Gulp will watch the `index.html` file, and run the `browserSyncReload()` function when any changes are made in the HTML.

The second `watch()` method handles our Sass and JavaScript files, and is more complicated. It watches any Sass files in the `app/scss` folder, and any JavaScript files in the `app/js` folder. Notice that we can use the wildcard (*) symbol to designate any file with a `*.scss` or `*.js` file extension.

Lastly, we also want to tell Gulp to *not* watch the temporary `app/js/scripts.js` file, using the exclamation point to signify that it should be ignored.

We need to do this because the `scripts.js` file is only deleted towards the end of our Gulp functions. If we don't explicitly ignore it, the moment it's generated by the `concatJsTask()` and `browserifyTask()`, it will trigger the `watchTask()` and then rerun all of our JavaScript functions, which will then trigger the `watchTask()` yet again.

This will keep happening over and over again in an infinite loop. Obviously we don't want this to happen, so we're going to ignore the file.

Once any CSS or JavaScript code changes trigger the second `watch()` method, it will then run a series of functions, many of them the same as in our default Gulp task:

- `scssTask()` to compile our Sass files,
- `concatJsTask()` to concatenate our JavaScript files,
- `browserifyTask()` to bundle our Node modules,
- `jsTask()` to run our other JavaScript processes,
- `cleanTask()` to delete the temporary `scripts.js` file,
- `cacheBustTask()` to refresh our cachebust querystring, and
- `browserSyncReload()` to reload our local server.

The `watchTask()` is running almost all the same functions that our default Gulp task is running. However, instead of `browserSyncServe()`, the `watchTask()` is running `browserSyncReload()` to reload the server.

And it is not running the `imageminTask()` function. We're separating that out into its own `watch()` method, where it will watch for any changes in the `img` folder and then only run `imageminTask()`.

One thing to double-check with your Gulp watch task is that most or all of the functions run in your default Gulp task are covered in your watch task.

I only mention this because there have been times when I've forgotten to include a certain function in my watch task. And you don't realize it until something isn't getting updated, and you have to retrace all your steps back to figure out why. So just make sure to do a quick check that everything you need is getting executed in your watch task.

Lastly, we need to update our default Gulp task to run the `watchTask()` after it executes all the other functions:

```
exports.default = series(
    scssTask,
    concatJsTask,
    browserifyTask,
    jsTask,
    cleanTask,
    imageminTask,
    cacheBustTask,
    browserSyncServe,
    watchTask
);
```

Now let's run `gulp` on the command line to test out our default Gulp task.

> **Note:** you may have to stop any current Browsersync server from running before rerunning Gulp. Otherwise changes to your gulpfile will not go through. You can stop it by killing the terminal window or pressing Ctrl-C to terminate the job.

If all goes well, you should see it run the workflow functions and also initiate the Browsersync server like before.

We also need to test that our `watchTask()` is working, by making changes to our `index.html` file.

Let's try adding the word "TEST" to the <h1> tag:

```
<h1 class="white">TEST Gulp Workflow Project</h1>
```

When you save the file, you should see some additional output in the terminal like this:

```
[Browsersync] Reloading Browsers...
```

Making code changes to the `index.html` file should only reload the Browsersync browser, not run any Gulp tasks.

And checking the local website in your browser, you should see the updated <h1> content:

Now let's test that the watch task is indeed running by making changes to our Sass and JavaScript files.

In the `_global.scss` file, find the `body` selector and comment out the `background-image` rule so that the background will simply be dark gray:

```
body {
    background-color: $darkGray;
    //background-image: url('/img/jr-korpa-stwHyPWNtbI-unsplash.jpg');
    background-size: cover;
    background-repeat: no-repeat;
    font-family: Arial, Helvetica, sans-serif;
    margin: 20px;
}
```

Monitoring the terminal output, when you save your changes to the global Sass file, it should rerun all the Gulp functions and also reload the local server:

```
[13:57:33] Starting 'scssTask'...
[13:57:34] Finished 'scssTask' after 738 ms
[13:57:34] Starting 'browserifyTask'...
[13:57:34] Finished 'browserifyTask' after 19 ms
[13:57:34] Starting 'jsTask'...
[13:57:34] Finished 'jsTask' after 121 ms
[13:57:34] Starting 'cleanTask'...
[13:57:34] Finished 'cleanTask' after 1.83 ms
[13:57:34] Starting 'imageminTask'...
[13:57:34] gulp-imagemin: ✓ jr-korpa-stwHyPWNtbI-unsplash.jpg (already optimized)
[13:57:34] gulp-imagemin: Minified 0 images
[13:57:34] Finished 'imageminTask' after 320 ms
[13:57:34] Starting 'cacheBustTask'...
[13:57:34] Finished 'cacheBustTask' after 5.58 ms
[13:57:34] Starting 'browserSyncReload'...
[13:57:34] Finished 'browserSyncReload' after 583 μs
[13:57:35] Starting 'browserSyncReload'...
[13:57:35] Finished 'browserSyncReload' after 697 μs
[Browsersync] Reloading Browsers...
```

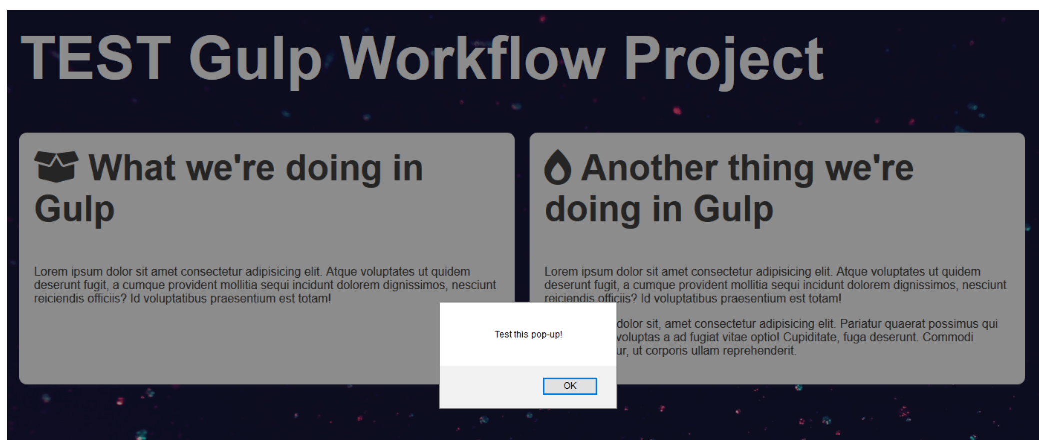
When you check the local website, it should have automatically reloaded, with the background now dark gray instead of having the background image

Let's uncomment that line in the global Sass file and bring the background image back. You should see the Gulp tasks rerun in your terminal, and the local website reloaded with the background image reinstated.

Now let's test our JavaScript changes by making a change in our `script.js` file. In the `app/js/script.js` file, we'll add a short `alert()` message at the bottom of the file like this:

```
alert('Test this pop-up!');
```

When we save the file, we should see the Gulp tasks rerunning in the terminal, and the website should now load a pop-up window like this:



You can comment out or delete the alert message once you've confirmed everything is working.

Lastly, we need to check that the image watch task is working. In our `img` folder I'm just going to copy and paste our image to create a duplicate. This should trigger the image watch task to run.

If you've been able to successfully trigger the `watchTask()` function by changing each type of file, you should be all set!

CREATING A PRODUCTION TASK

The last optimization that we will be setting up is creating a separate production Gulp task.

You usually have a production task when you have automatic deployments set up on your production server. Because you often don't want to run everything in your default Gulp task on production.

And conversely, there are some processes that we've set up that are more essential for the production environment, and not as necessary when we're just doing dev locally.

What we're going to do in this section is to configure our default Gulp task and our new production task with the specific Gulp functions that we want to run on each. This will help keep things more efficient in our workflow.

To get this set up, let's first export a new production task called "prod" in our gulpfile. We want it to run everything from our default task except the Browsersync and watch tasks. Because we don't want to load a local server or watch files on our production server.

The new prod task should look like this:

```
exports.prod = series(
    scssTask,
    concatJsTask,
    browserifyTask,
    jsTask,
    cleanTask,
    imageminTask,
    cacheBustTask
);
```

Again, note that it's identical to the default task except that it doesn't have the Browsersync and watch tasks.

Since we've called the task "prod" in the `exports.prod` section, we can run it on the command line by typing in `gulp prod`. For tasks other than the default task, you can name them basically anything you want.

Next, we're going to create dev and prod versions of both our `scssTask()` and the `jsTask()` functions. And these will replace the original functions.

We'll run the dev functions in our default and watch Gulp tasks, and run the prod functions in our production Gulp task. The difference is that the dev functions won't run the optimizing types of functions to save some time.

Here's what we have in our Sass workflow, with the `scssDevTask()` and `scssProdTask()`:

```
function scssDevTask(){
    return src('app/scss/style.scss', { sourcemaps: true })
        .pipe(sass({
            includePaths: 'node_modules/@fortawesome'
        }))
        .pipe(postcss([autoprefixer()]))
        .pipe(dest('dist', { sourcemaps: '.' }));
}

function scssProdTask(){
    return src('app/scss/style.scss', { sourcemaps: true })
        .pipe(sass({
            includePaths: 'node_modules/@fortawesome'
        }))
        .pipe(postcss([autoprefixer(), combinemq(), cssnano()]))
        .pipe(dest('dist', { sourcemaps: '.' }));
}
```

The `scssProdTask()` has everything we were using in our original `scssTask()`. The `scssDevTask()` is running most of those functions, with the exception of the media query combining and the CSS minifying.

For our JavaScript workflow, we are going to keep the original `concatJsTask()` and `browserifyTask()` functions, but will replace the original `jsTask()` function with the `jsDevTask()` and `jsProdTask()` functions:

```
function jsDevTask(){
    return src(['app/js/scripts.js'], { sourcemaps: true })
        .pipe(babel({ presets: ['@babel/preset-env']}))
        .pipe(dest('dist', { sourcemaps: '.' }));
}

function jsProdTask(){
    return src(['app/js/scripts.js'], { sourcemaps: true })
        .pipe(babel({ presets: ['@babel/preset-env']}))
        .pipe(uglify())
        .pipe(dest('dist', { sourcemaps: '.' }));
}
```

They are identical except for the fact that `jsProdTask()` minifies the JavaScript files using the `uglify()` method, and the `jsDevTask()` doesn't.

Now that we have our dev and prod functions set up, let's update our gulpfile so that the default and watch tasks run the dev functions, and the prod task runs the prod functions.

Here's what that final code will look like:

```
function watchTask(){
    watch('index.html', browserSyncReload);
    watch([
            'app/scss/**/*.scss',
            'app/js/**/*.js',
            '!app/js/scripts.js'
        ],
        series(
            scssDevTask,
            concatJsTask,
            browserifyTask,
            jsDevTask,
            cleanTask,
            cacheBustTask,
            browserSyncReload
        )
    );
    watch('img/*', series(imageminTask, browserSyncReload));
}

exports.default = series(
    scssDevTask,
    concatJsTask,
    browserifyTask,
    jsDevTask,
    cleanTask,
    imageminTask,
    cacheBustTask,
    browserSyncServe,
    watchTask
);

exports.prod = series(
    scssProdTask,
    concatJsTask,
    browserifyTask,
    jsProdTask,
    cleanTask,
    imageminTask,
```

```
    cacheBustTask
);
```

Now when we are working on our website, we can follow this workflow:

When doing normal development locally, we will run the default Gulp task by running `gulp` in the command line. This will run our default task once, and then continually run the watch task to check for file changes.

When we are finished with development and ready to deploy files to our production server, we'll want to first exit the running Gulp task. Like last time, you'll need to kill the terminal window or stop the batch job. When everything is stopped, then you can run the production task by typing in `gulp prod`.

In our terminal you should see all the Gulp tasks run and the prod task completing:

```
PS C:\Users\Jessica\Documents\GitHub\gulp-for-beginners> gulp prod
[14:02:22] Using gulpfile ~\Documents\GitHub\gulp-for-beginners\gulpfile.js
[14:02:22] Starting 'prod'...
[14:02:22] Starting 'scssProdTask'...
[14:02:23] Finished 'scssProdTask' after 1.28 s
[14:02:23] Starting 'browserifyTask'...
[14:02:23] Finished 'browserifyTask' after 35 ms
[14:02:23] Starting 'jsProdTask'...
[14:02:24] Finished 'jsProdTask' after 582 ms
[14:02:24] Starting 'cleanTask'...
[14:02:24] Finished 'cleanTask' after 7.29 ms
[14:02:24] Starting 'imageminTask'...
[14:02:24] gulp-imagemin: ✓ jr-korpa-stwHyPWNtbI-unsplash.jpg (already optimized)
[14:02:24] gulp-imagemin: Minified 0 images
[14:02:24] Finished 'imageminTask' after 426 ms
[14:02:24] Starting 'cacheBustTask'...
[14:02:24] Finished 'cacheBustTask' after 3.7 ms
[14:02:24] Finished 'prod' after 2.34 s
```

A note on the production task

You might be wondering why the production task still needs to run all the Gulp tasks that the default task already is running. It is true that if you're working locally, you will already be running the default and watch tasks.

However, in a true production setup, you may have multiple devs working in the same codebase, often at the same time. And each dev is working locally and committing their changes to Git or other source control.

And if you are compiling CSS and JavaScript files locally, it can often cause Git merge conflicts with the changes from the other devs. Because the final compiled files are all

going to be slightly different from one another, and it's really hard to resolve merge conflicts in a compiled file because it's compressed.

To solve for this, usually you would ignore any of the compiled files in Git. So each dev is working locally and they each have their own version of the compiled files, but none of those compiled files are getting committed to the central Git repository.

Then, when code changes are deployed to the production server, you would then run your Gulp prod task after the new files are added. This will create another set of compiled files, just on the production server.

If you're not working with a full production setup, then you may not need to worry about having a separate production Gulp task. But it can definitely come in handy when working on a project with multiple other developers.

In Closing

And that's it for setting up your Gulp workflow!

Keep in mind that you are free to make any tweaks that make it work better for you. I've shown you an initial setup that covers most scenarios, but you might have other preferences or needs. So it's completely ok to not include all the functions and processes here if you don't need them.

I sincerely hope that this course has helped you use Gulp to build websites more efficiently. If you have any questions or feedback, please feel free to send me an email at jessica@coder-coder.com.

In addition, you can find me online in the following places:

- Instagram
- YouTube
- Twitter
- Coder-Coder.com

Lastly, thank you so much for your support by purchasing this course. You've helped enable me to continue creating more content that helps people learn web development online.

Cheers, and best of luck in your coding journey!

~ Jessica