MODERN JAVASCRIPT

Modern JavaScript

Copyright © 2017 SitePoint Pty. Ltd.

Cover Design: Natalia Balska

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

Table of Contents

Preface

It's not uncommon these days to see people complaining about just how complex JavaScript development seems to have become. We can have some sympathy with that view when it's coming from someone new to the language. If you're learning JS, it won't take long for you to be exposed to the enormity of the ecosystem and the sheer number of moving pieces you need to understand (at least conceptually) to build a modern web application. Package management, linting, transpilation, module bundling, minification, source maps, frameworks, unit testing, hot reloading... it can't be denied that this is a lot more complex that just including a couple of script tags in your page and FTPing it up to the server.

For a long time, JavaScript was looked upon by many as a joke; a toy language whose only real use was to add non-essential eye-candy, such as mouseover changes, and was often a source of weird errors and broken pages. The language is still not taken seriously by some today, despite having made much progress since those early days. It's not hard to have some sympathy with PHP developers. For better or for worse, JavaScript was (and still is) the only language supported natively by the vast majority of web browsers. The community has worked hard to improve the language itself, and to provide the tooling to help build production-grade apps.

This anthology is a collection of articles, hand-picked from SitePoint's JavaScript channel with the aim of giving you a head start on modern JavaScript development. Of course, with the fast pace of change of both the language and the eco-system, it's important to keep up to date with the latest developments. Don't be left behind, head over to the channel after you've finished this book to stay abreast of new trends and techniques!

James Hibbard & Nilson Jacques, SitePoint JavaScript channel editors.

Conventions Used

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk.</p>
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ↪ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-design-real
↪ -user-testing/?responsive1");
```

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Chapter **1**

The Anatomy of a Modern JavaScript Application

by James Kolce

There's no doubt that the JavaScript ecosystem changes fast. Not only are new tools and frameworks introduced and developed at a rapid rate, the language itself has undergone big changes with the introduction of ES2015 (aka ES6). Understandably, many articles have been written complaining about how difficult it is to learn modern JavaScript development these days.

In this article, I'll introduce you to modern JavaScript. We'll take a look at the most recent developments in the language and get an overview of the tools and techniques currently used to write front-end web applications. If you're just starting out with learning the language, or you've not touched it for a few years

and are wondering what happened to the JavaScript you used to know, this article is for you.

A Note about Node.js

Node.js is a runtime that allows server-side programs to be written in JavaScript. It is possible to have full-stack JavaScript applications, where both the front and back-end of the app is written in the same language. Although this article is focused on client-side development, Node.js still plays an important role.

The arrival of Node.js had a significant impact on the JavaScript ecosystem, introducing the npm package manager and popularizing the CommonJS module format. Developers started to build more innovative tools and develop new approaches to blur the line between the browser, the server, and native applications.

JavaScript ES2015+

In 2015, the sixth version of ECMAScript—the specification that defines the JavaScript language—was released under the name of ES2015 (still often referred to as ES6). This new version included substantial additions to the language making easier and more feasible to build ambitious web applications. But improvements don't stop with ES2015; each year, a new version is released.

Declaring variables

JavaScript now has two additional ways to declare variables: **let** and **const**.

`let` is the successor to `var` - although `var` is still available, `let` limits the scope of variables to the block (rather than the function) they're declared within, which reduces the room for error:

```
/ ES5
for (var i = 1; i < 5; i++) {
  console.log(i);
}
```

```
// <-- logs the numbers 1 to 4
console.log(i);
// <-- 5 (variable i still exists outside the loop)

// ES2015
for (let j = 1; j < 5; j++) {
  console.log(j);
}
console.log(j);
// <-- 'Uncaught ReferenceError: j is not defined'
```

Using const allows you to define variables that cannot be rebound to new values. For primitive values such as strings and numbers, this results in something similar to a constant, as you cannot change the value once it has been declared.

```
const name = 'Bill';
name = 'Steve';
 // <-- 'Uncaught TypeError: Assignment to constant
↪ variable.'

// Gotcha
const person = { name: 'Bill' };
person.name = 'Steve';
// person.name is now Steve.
 // As we're not changing the object that person is bound to,
↪ JavaScript doesn't complain.
```

Arrow functions

Arrow functions provide a cleaner syntax for declaring anonymous functions (lambdas), dropping the function keyword and the return keyword when the body function only has one expression. This can allow you to write functional style code in a nicer way.

```
/ ES5
var add = function(a, b) {
  return a + b;
}

// ES2015
const add = (a, b) => a + b;
```

The other important feature of arrow functions is that they inherit the value of `this` from the context in which they are defined:

```
function Person(){
  this.age = 0;

  // ES5
  setInterval(function() {
    this.age++; // |this| refers to the global object
  }, 1000);

  // ES2015
  setInterval(() => {
    this.age++; // |this| properly refers to the person object
  }, 1000);
}

var p = new Person();
```

Improved Class syntax

If you are a fan of object-oriented programming, you might like the addition of classes to the language on top of the existent mechanism based on prototypes. While it is just syntactic sugar, it provides a cleaner syntax for developers trying to emulate classical object-orientation with prototypes.

```
class Person {
  constructor(name) {
```

```
    this.name = name;
  }

  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
}
```

Promises / Async functions

The asynchronous nature of JavaScript has long represented a challenge; any non-trivial application ran the risk of falling into a callback hell when dealing with things like Ajax request.

Fortunately, ES2015 added native support for <u>promises</u>. Promises represent values that don't exist at the moment of the computation but that may be available later, making the management of asynchronous function calls more manageable without getting into deeply nested callbacks.

ES2017 (due out this year) introduces <u>async functions</u> (sometimes referred to as async/await that make improvements in this area, allowing you to treat asynchronous code as if it were synchronous.

```
async function doAsyncOp () {
  const result = await asynchronousOperation();
  console.log(result);
  return result;
};
```

Modules

Another prominent feature added in ES2015 is a native module format, making the definition and usage of modules a part of the language. Loading modules was previously only available in the form of third-party libraries. We'll look at modules in more depth in the next section.

There are other features we won't talk about here, but we've covered some of the major differences you're likely to notice when looking at modern JavaScript. You can check a complete list with examples on the Learn ES2015 page on the Babel site, which you might find useful to get up to date with the language. Some of those features include template strings, iterators, generators, new data structures such as Map and Set, and more.

More on ES2015

To learn more about ES2015, check out our Premium course: Diving into ES2015

Code linting

Linters are tools that parse your code and compare it against a set of rules, checking for syntax errors, formatting, and good practices. Although the use of a linter is recommended to everyone, it is especially useful if you are getting started. When configured correctly for your code editor/IDE you can get instant feedback to ensure you don't get stuck with syntax errors as you're learning new language features.

You can check out ESLint which is one of the most popular and supports ES2015+.

Modular Code

Modern web applications can have thousands (even hundred of thousands) of lines of code. Working at that size becomes almost impossible without a mechanism to organize everything in smaller components, writing specialized and isolated pieces of code that can be reused as necessary in a controlled way. This is the job of modules.

CommonJS modules

A handful of module formats have emerged over the years, the most popular of which is CommonJS. It's the default module format in Node.js, and can be used in client-side code with the help of module bundlers, which we'll talk about shortly.

It makes use of a `module` object to export functionality from a JavaScript file and a `require()` function to import that functionality where you need it.

```
// lib/math.js
function sum(x, y) {
  return x + y;
}

const pi = 3.141593

module.exports = {
  sum: sum,
  pi: pi
};


// app.js
const math = require("lib/math");

console.log("2π = " + math.sum(math.pi, math.pi));
```

ES2015 modules

ES2015 introduces a way to define and consume components right into the language, which was previously possible only with third-party libraries. You can have separate files with the functionality you want, and export just certain parts to make them available to your application.

Broswer Support

Note: Native browser support for ES2015 modules is still under development, so you currently need some additional tools to be able to use them.

Here's an example:

```
/ lib/math.js
```

```
export function sum(x, y) {
  return x + y;
}
export let pi = 3.141593;
```

Here we have a module that *exports* a function and a variable. We can include that file in another one and use those exported functions:

```
/ app.js

import * as math from "lib/math";

console.log("2π = " + math.sum(math.pi, math.pi));
```

Or we can also be specific and import only what we need:

```
/ otherApp.js

import {sum, pi} from "lib/math";

console.log("2π = " + sum(pi, pi));
```

These examples have been extracted from the Babel website. For an in-depth look, check out Understanding ES6 Modules.

Package Management

Other languages have long had their own package repositories and managers to make it easier to find and install third-party libraries and components. Node.js comes with its own package manager and repository, npm. Although there are other package managers available, npm has become the de facto JavaScript package manager and is said to be the largest package registry in the world.

In the npm repository you can find third-party modules that you can easily download and use in your projects with a single `npm install <package>` command. The packages are downloaded into a local `node_modules` directory, which contains all the packages and their dependencies.

The packages that you download can be registered as dependencies of your project in a package.json file, along with information about your project or module (which can itself be published as a package on npm).

You can define separate dependencies for both development and production. While the production dependencies are needed for the package to work, the development dependencies are only necessary for the developers of the package.

Example package.json file

```
{
  "name": "demo",
  "version": "1.0.0",
  "description": "Demo package.json",
  "main": "main.js",
  "dependencies": {
    "mkdirp": "^0.5.1",
    "underscore": "^1.8.3"
  },
  "devDependencies": {},
  "scripts": {
 "test": "echo \"Error: no test specified\" && exit
↳ 1"
  },
  "author": "Sitepoint",
  "license": "ISC"
}
```

Build Tools

The code that we write when developing modern web applications almost never is the same code that will go to production. We write code in a modern version of

JavaScript that may not be supported by the browser, we make heavy use of third-party packages that are in a node_modules folder along with their own dependencies, we can have processes like static analysis tools or minifiers, etc. Build tooling exists to help transform all this into something that can be deployed efficiently and that is understood by most web browsers.

Module bundling

When writing clean, reusable code with ES2015/CommonJS modules, we need some way to load these modules (at least until browsers support ES2015 module loading natively). Including a bunch of script tags in your HTML isn't really a viable option as it would quickly become unwieldy for any serious application, and all those separate HTTP requests would hurt performance.

We can include all the modules where we need them using the `import` statement from ES2015 (or `require`, for CommonJS) and use a module bundler to combine everything together into one or more files (bundles). It's this bundled file that we are going to upload to our server and include in our HTML. It will include all your imported modules and their necessary dependencies.

There are a currently a couple of popular options for this, the most popular ones are Webpack, Browserify and Rollup.js. You can choose one or another depending on your needs.

> ### learning More About Module Bundling
>
> If you want to learn more about module bundling and how it fits into the bigger picture of app development, I recommend reading Understanding JavaScript Modules: Bundling & Transpiling.

Transpilation

While support for ES2015 is pretty good among modern browsers, your target audience may include legacy browsers and devices with partial or no support.

In order to make our modern JavaScript work, we need to translate the code we write to its equivalent in an earlier version (usually ES5). The standard tool for this task is Babel; a compiler that translates your code into compatible code for

most browsers. In this way you don't have to wait for vendors to implement everything, you can just use all the modern JS features.

There are a couple of features that need more than a syntax translation; Babel includes a <u>Polyfill</u> that emulates some of the machinery required for some complex features, like promises.

Build systems & task runners

Module bundling and transpilation are just two of the build processes that we may need in our projects. Others include code minification (to reduce files sizes), tools for analysis, and perhaps tasks that don't have anything to do with JavaScript, like image optimizations or CSS/HTML pre-processing.

The management of tasks can become a laborious thing to do, and we need a way to handle it in an automated way, being able to execute everything with simpler commands. The two most popular tools for this are <u>Grunt.js</u> and <u>Gulp.js</u>, they provide a way to organize your tasks into groups in an ordered way.

For example, you can have a command like `gulp build` which may run a code linter, the transpilation process with Babel, and module bundling with Browserify. Instead of having to remember three commands and their associated arguments in order, we just execute one that will handle the whole process automatically.

Wherever you find yourself manually organizing processing steps for your project, think if it can be automatized with a task runner.

Application Architecture

Web applications have different requirements than websites. For example, while page reloads may be acceptable for a blog, that is certainly not the case for an application like Google Docs. Your application should behave as close as possible to a desktop one, otherwise, the usability would be compromised.

Old-style web applications were usually done by sending multiple pages from a web server, and when a lot of dynamism was needed, content was loaded via

Ajax by replacing chunks of HTML according to user actions. Although it was a big step forward to a more dynamic web, it certainly had its complications; sending HTML fragments or even whole pages on each user action represented a waste of resources, especially time from the user's perspective. The usability still didn't match the responsiveness of desktop applications.

Looking to improve things, we created to new methods to build web applications, from the way we present them to the user to the way we communicate the client with the server. Although the amount of JavaScript required for an application also increased drastically, the result is now applications that behave very closely to native ones; without page reloading or extensive waiting periods each time we click a button.

Single Page Applications (SPAs)

The most common high-level architecture for web applications is called SPA, which stands for *Single Page Application.* SPAs are big blobs of JavaScript that contain everything the application needs to work properly. The UI is rendered entirely client-side, so no reloading is required. The only thing that changes is the data inside the application, which is usually handled with a remote API via Ajax or another asynchronous method of communication.

One downside to this approach is that the application takes longer to load for the first time. Once it has been loaded, however, transitions between views (pages) are generally a lot quicker, since it is only pure data being sent between client and server.

Universal / Isomorphic Applications

Although SPAs provide a great user experience, depending on your needs, they might not be the optimal solution. Especially if you need quicker initial response times or optimal indexing by search engines.

There is a fairly recent approach to solving these problems called Isomorphic (or Universal) JavaScript applications. In this type of architecture, most of the code can be executed both on the server and the client. You can choose what you want to render on the server for a faster initial page load, and after that, the client takes

over the rendering while the user is interacting with the app. Because pages are initially rendered on the server, search engines can index them properly.

Deployment

With modern JavaScript applications, the code that you write is not the same as the code that you deploy for production; you only deploy the result of your build process. The workflow to accomplish this can vary depending on the size of your project, the number of developers working on it, and sometimes the tools/libraries that you are using.

For example, if you are working alone on a simple project, each time you are ready for deployment you can just run the build process and upload the resulting files to a web server. Keep in mind that you only need to upload the resulting files from the build process (transpilation, module bundling, minification, etc.), which can be just one `.js` file containing your entire application and dependencies.

You can have a directory structure like this:

```
├── dist
│   ├── app.js
│   └── index.html
├── node_modules
├── src
│   ├── lib
│   │   ├── login.js
│   │   └── user.js
│   ├── app.js
│   └── index.html
├── gulpfile.js
├── package.json
└── README
```

Having all of your application files in an `src` directory, written in ES2015, and importing packages installed with npm and your own modules from a `lib` directory.

Then you can run Gulp, which will execute the instructions from a `gulpfile.js`to build your project: bundling all modules into one file (including the ones installed with npm), transpiling ES2015+ to ES5, minifying the resulted file, etc. Then you can configure it to output the result in a convenient `dist` directory.

Files That Don't Need Processing

If you have files that don't need any processing, you can just copy them from `src` to the `dist` directory. You can configure a task for that in your build system.

Now you can just upload the files from the `dist` directory to a web server, without having to worry about the rest of the files, which are only useful for development.

Team development

If you are working with other developers, it is likely that you are also using a shared code repository, like GitHub, to store the project. In this case, you can run the building process right before making commits and store the result with the other files in the Git repository, to later be downloaded onto a production server.

However, storing built files into the repository is prone to errors if several developers are working together and you might want to keep everything clean from build artifacts. Fortunately, there is a better way to deal with that problem: you can put a service like Jenkins, Travis CI, CircleCI, etc. in the middle of the process, so it can automatically build your project after each commit is pushed to the repository. Developers only have to worry about pushing code changes without building the project first each time, also the repository is kept clean of automatically generated files, and at the end, you still have the built files available for deployment.

Conclusion

The transition from simple web pages to modern JavaScript applications can seem daunting if you have been away from web development during recent years,

but I hope this article was useful as a starting point. I've linked to more in-depth articles on each topic where possible, so you can explore further.

And remember that if at some point, after looking all the options available, everything seems overwhelming and messy; just keep in mind the <u>KISS principle</u>, and use only what you think you need and not everything you have available. At the end of the day, solving problems is what matters, not using the latest of everything.

Chapter

2

An Introduction to Gulp.js

by Craig Buckler

Peer reviewed by <u>Giulio Mainardi</u> and <u>Tim Severien</u>.

Developers spend precious little time coding. Even if we ignore irritating meetings, much of the job involves basic tasks which can sap your working day:

- generating HTML from templates and content files
- compressing new and modified images
- compiling Sass to CSS code
- removing `console` and `debugger` statements from scripts
- transpiling ES6 to cross-browser-compatible ES5 code
- code linting and validation
- concatenating and minifying CSS and JavaScript files

■ deploying files to development, staging and production servers

Tasks must be repeated every time you make a change. You may start with good intentions but the most infallible developer will forget to compress an image or two. Over time, pre-production tasks become increasingly arduous and time-consuming; you'll dread the inevitable content and template changes. It's mind-numbing, repetitive work. Would it be better to spend your time on more profitable jobs?

If so, you need a *task runner* or *build process*.

That Sounds Scarily Complicated!

Creating a build process will take time. It's more complex than performing each task manually but, over the long-term, you will save hours of effort, reduce human error and save your sanity. Adopt a pragmatic approach:

■ automate the most frustrating tasks first
■ try not to over-complicate your build process; an hour or two is more than enough for the initial set-up
■ choose task runner software and stick with it for a while. Don't switch to another option on a whim.

Some of the tools and concepts may be new to you but take a deep breath and concentrate on one thing at a time.

Task Runners: the Options

Build tools such as GNU Make have been available for decades but web-specific task runners are a relatively new phenomenon. The first to achieve critical mass was Grunt - a Node.js task runner which used plug-ins controlled (originally) by a JSON configuration file. Grunt was hugely successful but there were a number of issues:

1. Grunt required plug-ins for basic functionality such as file watching.
2. Grunt plug-ins often performed multiple tasks which made customisation more awkward.

3. JSON configuration could become unwieldy for all but the most basic tasks.
4. Tasks could run slowly because Grunt saved files between every processing step.

Many issues were addressed in later editions but <u>Gulp</u> had already arrived and offered a number of improvements:

1. Features such as file watching were built-in.
2. Gulp plug-ins were *(mostly)* designed to do a single job.
3. Gulp used JavaScript configuration code which was less verbose, easier to read, simpler to modify, and provided better flexibility.
4. Gulp was faster because it uses <u>Node.js streams</u> to pass data through a series of piped plug-ins. Files were only written at the end of the task.

Of course, Gulp itself isn't perfect and new task runners such as <u>Broccoli.js</u>, <u>Brunch</u> and <u>webpack</u> have also been competing for developer attention. More recently, <u>npm itself has been touted as a simpler option</u>. All have their pros and cons, but <u>Gulp remains the favorite and is currently used by more than 40% of web developers</u>.

Gulp requires Node.js but, while some JavaScript knowledge is beneficial, developers from all web programming faiths will find it useful.

What About Gulp 4?

This tutorial describes how to use Gulp 3 - the most recent release version at the time of writing. Gulp 4 has been in development for some time but remains a beta product. It's possible to <u>use or switch to Gulp 4</u> but I recommend sticking with version 3 until the final release.

Step 1: Install Node.js

Node.js can be downloaded for Windows, Mac and Linux from <u>nodejs.org/ download/</u>. There are various options for installing from binaries, package managers and docker images - full instructions are available.

Note for Windows Users

Node.js and Gulp run on Windows but some plug-ins may not install or run if they depend on native Linux binaries such as image compression libraries. One option for Windows 10 users is the new bash command-line; this solves many issues but is a beta product and could introduce alternative problems.

Once installed, open a command prompt and enter:

```
node -v
```

to reveal the version number. You're about to make heavy use of npm - the Node.js package manager which is used to install modules. Examine its version number:

```
npm -v
```

Note for Linux users: Node.js modules can be installed globally so they are available throughout your system. However, most users will not have permission to write to the global directories unless npm commands are prefixed with sudo. There are a number of options to fix npm permissions and tools such as nvm can help but I often change the default directory, e.g. on Ubuntu/Debian-based platforms:

```
cd ~
mkdir .node_modules_global
npm config set prefix=$HOME/.node_modules_global
npm install npm -g
```

Then add the following line to the end of ~/.bashrc:

```
export PATH="$HOME/.node_modules_global/bin:$PATH"
```

and update with:

```
source ~/.bashrc
```

Step 2: Install Gulp Globally

Install Gulp command-line interface globally so the `gulp` command can be run from any project folder:

```
npm install gulp-cli -g
```

Verify Gulp has installed with:

```
gulp -v
```

Step 3: Configure Your Project

Note for Node.js projects: you can skip this step if you already have a `package.json` configuration file.

Presume you have a new or pre-existing project in the folder `project1`. Navigate to this folder and initialize it with npm:

```
cd project1
npm init
```

You will be asked a series of questions - enter a value or hit **Return** to accept defaults. A `package.json` file will be created on completion which stores your `npm` configuration settings.

Note for Git users: Node.js installs modules to a `node_modules` folder. You should add this to your `.gitignore` file to ensure they are not committed to your

repository. When deploying the project to another PC, you can run `npm install` to restore them.

For the remainder of this article we'll presume your project folder contains the following sub-folders:

`src` folder: pre-processed source files

This contains further sub-folders:

- `html` - HTML source files and templates
- `images` — the original uncompressed images
- `js` — multiple pre-processed script files
- `scss` — multiple pre-processed Sass `.scss` files

`build` folder: compiled/processed files

Gulp will create files and create sub-folders as necessary:

- `html` - compiled static HTML files
- `images` — compressed images
- `js` — a single concatenated and minified JavaScript file
- `css` — a single compiled and minified CSS file

Your project will almost certainly be different but this structure is used for the examples below.

Tip: If you're on a Unix-based system and you just want to follow along with the tutorial, you can recreate the folder structure with the following command:

```
 mkdir -p src/{html,images,js,scss}
↪ build/{html,images,js,css}
```

Step 4: Install Gulp Locally

You can now install Gulp in your project folder using the command:

```
npm install gulp --save-dev
```

This installs Gulp as a development dependency and the `"devDependencies"` section of `package.json` is updated accordingly. We will presume Gulp and all plug-ins are development dependencies for the remainder of this tutorial.

Alternative Deployment Options

Development dependencies are not installed when the `NODE_ENV` environment variable is set to `production` on your operating system. You would normally do this on your live server with the Mac/Linux command:

```
export NODE_ENV=production
```

Or on Windows:

```
set NODE_ENV=production
```

This tutorial presumes your assets will be compiled to the `build` folder and committed to your Git repository or uploaded directly to the server. However, it may be preferable to build assets on the live server if you want to change the way they are created, e.g. HTML, CSS and JavaScript files are minified on production but not development environments. In that case, use the `--save` option for Gulp and all plug-ins, i.e.

```
npm install gulp --save
```

This sets Gulp as an application dependency in the `"dependencies"` section of `package.json`. It will be installed when you enter `npm install` and can be run wherever the project is deployed. You can remove the `build` folder from your repository since the files can be created on any platform when required.

Step 4: Create a Gulp Configuration File

Create a new `gulpfile.js` configuration file in the root of your project folder. Add some basic code to get started:

```
// Gulp.js configuration
var
  // modules
  gulp = require('gulp'),

  // development mode?
  devBuild = (process.env.NODE_ENV !== 'production'),

  // folders
  folder = {
    src: 'src/',
    build: 'build/'
  }
;
```

This references the Gulp module, sets a `devBuild` variable to `true` when running in development (or non-production mode) and defines the source and build folder locations.

ES6 note: ES5-compatible JavaScript code is provided in this tutorial. This will work for all versions of Gulp and Node.js with or without the `--harmony` flag. Most ES6 features are supported in Node 6 and above so feel free to use arrow functions, `let`, `const`, etc. if you're using a recent version.

`gulpfile.js` won't do anything yet because you need to...

Step 5: Create Gulp Tasks

On it's own, Gulp does nothing. You must:

1. install Gulp plug-ins, and

2.   write tasks which utilize those plug-ins to do something useful.

It's possible to write your own plug-ins but, since almost 3,000 are available, it's unlikely you'll ever need to. You can search using Gulp's own directory at gulpjs.com/plugins/, on npmjs.com, or search "gulp *something*" to harness the mighty power of Google.

Gulp provides three primary task methods:

- `gulp.task` - defines a new task with a name, optional array of dependencies and a function.
- `gulp.src` - sets the folder where source files are located.
- `gulp.dest` - sets the destination folder where build files will be placed.

Any number of plug-in calls are set with `pipe` between the `.src` and `.dest`.

Image Task

This is best demonstrated with an example so let's create a basic task which compresses images and copies them to the appropriate `build` folder. Since this process could take time, we'll only compress new and modified files. Two plug-ins can help us: gulp-newer and gulp-imagemin. Install them from the command-line:

```
npm install gulp-newer gulp-imagemin --save-dev
```

We can now reference both modules the top of `gulpfile.js`:

```
// Gulp.js configuration

var
  // modules
  gulp = require('gulp'),
  newer = require('gulp-newer'),
  imagemin = require('gulp-imagemin'),
```

We can now define the image processing task itself as a function at the end of `gulpfile.js`:

```
// image processing
gulp.task('images', function() {
  var out = folder.build + 'images/';
  return gulp.src(folder.src + 'images/**/*')
    .pipe(newer(out))
    .pipe(imagemin({ optimizationLevel: 5 }))
    .pipe(gulp.dest(out));
});
```

All tasks are syntactically similar. This code:

1. Creates a new task named `images`.
2. Defines a function with a return value which...
3. Defines an `out` folder where build files will be located.
4. Sets the Gulp `src` source folder. The `/**/*` ensures that images in sub-folders are also processed.
5. Pipes all files to the `gulp-newer` module. Source files that are newer than corresponding destination files are passed through. Everything else is removed.
6. The remaining new or changed files are piped through `gulp-imagemin` which sets an optional `optimizationLevel` argument.
7. The compressed images are output to the Gulp `dest` folder set by `out`.

Save `gulpfile.js` and place a few images in your project's `src/images` folder before running the task from the command line:

```
gulp images
```

All images are compressed accordingly and you will see output such as:

```
Using file gulpfile.js
Running 'imagemin'...
```

```
Finished 'imagemin' in 5.71 ms
gulp-imagemin: image1.png (saved 48.7 kB)
gulp-imagemin: image2.jpg (saved 36.2 kB)
gulp-imagemin: image3.svg (saved 12.8 kB)
```

Try running `gulp images` again and nothing should happen because no newer images exist.

HTML Task

We can now create a similar task which copies files from the source HTML folder. We can safely minify our HTML code to remove unnecessary whitespace and attributes using the gulp-htmlclean plug-in:

```
npm install gulp-htmlclean --save-dev
```

which is then referenced at the top of `gulpfile.js`:

```
var
  // modules
  gulp = require('gulp'),
  newer = require('gulp-newer'),
  imagemin = require('gulp-imagemin'),
  htmlclean = require('gulp-htmlclean'),
```

We can now create an `html` task at the end of `gulpfile.js`:

```
// HTML processing
gulp.task('html', ['images'], function() {
  var
    out = folder.build + 'html/',
    page = gulp.src(folder.src + 'html/**/*')
      .pipe(newer(out));
```

```
  // minify production code
  if (!devBuild) {
    page = page.pipe(htmlclean());
  }

  return page.pipe(gulp.dest(out));
});
```

This reuses `gulp-newer` and introduces a couple of concepts:

1. The `[images]` argument states that our `images` task must be run before processing the HTML (the HTML is likely to reference images). Any number of dependent tasks can be listed in this array and all will complete before the task function runs.
2. We only pipe the HTML through `gulp-htmlclean` if `NODE_ENV` is set to `production`. Therefore, the HTML remains uncompressed during development which may be useful for debugging.

Save `gulpfile.js` and run `gulp html` from the command line. Both the `html` and `images` tasks will run.

JavaScript Task

Too easy for you? Let's process all our JavaScript files by building a basic module bundler. It will:

1. ensure dependencies are loaded first using the gulp-deporder plug-in. This analyses comments at the top of each script to ensure correct ordering e.g. `// requires: defaults.js lib.js`.
2. concatenate all script files into a single `main.js` file using gulp-concat, and
3. remove all `console` and `debugging` statements with gulp-strip-debug and minimize code with gulp-uglify. This step will only occur when running in production mode.

Install the plug-ins:

```
 npm install gulp-deporder gulp-concat gulp-strip-debug
↪ gulp-uglify --save-dev
```

Reference them at the top of `gulpfile.js`:

```
var
  ...
  concat = require('gulp-concat'),
  deporder = require('gulp-deporder'),
  stripdebug = require('gulp-strip-debug'),
  uglify = require('gulp-uglify'),
```

Then add a new `js` task:

```
// JavaScript processing
gulp.task('js', function() {

  var jsbuild = gulp.src(folder.src + 'js/**/*')
    .pipe(deporder())
    .pipe(concat('main.js'));

  if (!devBuild) {
    jsbuild = jsbuild
      .pipe(stripdebug())
      .pipe(uglify());
  }

  return jsbuild.pipe(gulp.dest(folder.build + 'js/'));

});
```

Save then run `gulp js` to watch the magic happen!

CSS Task

Finally, let's create a CSS task which compiles Sass `.scss` files to a single `.css` file using gulp-sass. This is a Gulp plug-in for node-sass which binds to the super-fast LibSass C/C++ port of the Sass engine *(you won't need to install Ruby).* We'll presume your primary Sass file `scss/main.scss` is responsible for loading all partials.

Our task will also utilize the fabulous PostCSS via the gulp-postcss plug-in. PostCSS requires its own set of plug-ins and we'll install:

- postcss-assets to manage assets. This allows us to use properties such as `background: resolve('image.png');` to resolve file paths or `background: inline('image.png');` to inline data-encoded images.
- autoprefixer to automatically add vendor prefixes to CSS properties.
- css-mqpacker to pack multiple references to the same CSS media query into a single rule.
- cssnano to minify the CSS code when running in production mode.

First, install all the modules:

```
 npm install gulp-sass gulp-postcss postcss-assets
↳ autoprefixer css-mqpacker cssnano --save-dev
```

and reference them at the top of `gulpfile.js`:

```
var
  ...
  sass = require('gulp-sass'),
  postcss = require('gulp-postcss'),
  assets = require('postcss-assets'),
  autoprefixer = require('autoprefixer'),
  mqpacker = require('css-mqpacker'),
  cssnano = require('cssnano'),
```

We can now create a new `css` task at the end of `gulpfile.js`. Note the `images` task is set as a dependency because the `postcss-assets` plug-in can reference images during the build process. In addition, most plug-ins can be passed arguments - refer to their documentation for more information:

```
// CSS processing
gulp.task('css', ['images'], function() {

  var postCssOpts = [
  assets({ loadPaths: ['images/'] }),
  autoprefixer({ browsers: ['last 2 versions', '> 2%'] }),
  mqpacker
  ];

  if (!devBuild) {
    postCssOpts.push(cssnano);
  }

  return gulp.src(folder.src + 'scss/main.scss')
    .pipe(sass({
      outputStyle: 'nested',
      imagePath: 'images/',
      precision: 3,
      errLogToConsole: true
    }))
    .pipe(postcss(postCssOpts))
    .pipe(gulp.dest(folder.build + 'css/'));

});
```

Save the file and run the task from the command line:

```
gulp css
```

Step 6: Automate Tasks

We've been running one task at a time. We can run them all in one command by adding a new `run` task to `gulpfile.js`:

```
// run all tasks
gulp.task('run', ['html', 'css', 'js']);
```

Save and enter `gulp run` at the command line to execute all tasks. Note I omitted the `images` task because it's already set as a dependency for the `html` and `css` tasks.

Is this still too much hard work? Gulp offers another method - `gulp.watch` - which can monitor your source files and run an appropriate task whenever a file is changed. The method is passed a folder and a list of tasks to execute when a change occurs. Let's create a new `watch` task at the end of `gulpfile.js`:

```
// watch for changes
gulp.task('watch', function() {

  // image changes
  gulp.watch(folder.src + 'images/**/*', ['images']);

  // html changes
  gulp.watch(folder.src + 'html/**/*', ['html']);

  // javascript changes
  gulp.watch(folder.src + 'js/**/*', ['js']);

  // css changes
  gulp.watch(folder.src + 'scss/**/*', ['css']);

});
```

Rather than running `gulp watch` immediately, let's add a default task:

```
// default task
gulp.task('default', ['run', 'watch']);
```

Save `gulpfile.js` and enter `gulp` at the command line. Your images, HTML, CSS and JavaScript will all be processed then Gulp will remain active watching for updates and re-running tasks as necessary. Hit Ctrl/Cmd + C to abort monitoring and return to the command line.

Step 7: Profit!

Other plug-ins you may find useful:

- gulp-load-plugins - load all Gulp plug-in modules without `require` declarations
- gulp-preprocess - a simple HTML and JavaScript preprocessor
- gulp-less - the Less CSS pre-processor plug-in
- gulp-stylus - the Stylus CSS pre-processor plug-in
- gulp-sequence - run a series of gulp tasks in a specific order
- gulp-plumber - error handling which prevents Gulp stopping on failures
- gulp-size - displays file sizes and savings
- gulp-nodemon - uses nodemon to automatically restart Node.js applications when changes occur.
- gulp-util - utility functions including logging and color-coding

One useful method in `gulp-util` is `.noop()` which passes data straight through without performing any action. This could be used for cleaner development/ production processing code, e.g.

```
var gutil = require('gulp-util');

// HTML processing
gulp.task('html', ['images'], function() {
  var out = folder.src + 'html/**/*';

  return gulp.src(folder.src + 'html/**/*')
    .pipe(newer(out))
```

```
    .pipe(devBuild ? gutil.noop() : htmlclean())
    .pipe(gulp.dest(out));

});
```

Gulp can also call other Node.js modules - they don't necessarily need to be plug-ins, e.g.

- browser-sync - automatically reload assets or refresh your browser when changes occur
- del - delete files and folders (perhaps clean your `build` folder at the start of every run).

Invest a little time and Gulp could save many hours of development frustration. The advantages:

- plug-ins are plentiful
- configuration using pipes is readable and easy to follow
- `gulpfile.js` can be adapted and reused in other projects
- your total page weight can be reduced to improve performance
- you can simplify your deployment.

Useful links:

- Gulp home page
- Gulp plug-ins
- npm home page

Applying the processes above to a simple website reduced the total weight by more than 50%. You can test your own results using page weight analysis tools or a service such as New Relic which provides a range of sophisticated application performance monitoring tools.

<div align="right">

Chapter **3**

</div>

The Basics of DOM Manipulation in Vanilla JavaScript (No jQuery)

by Sebastian Seitz

Peer reviewed by <u>Vildan Softic</u> and <u>Joan Yin</u>.

Whenever we need to perform DOM manipulation, we're all quick to reach for jQuery. However, the vanilla JavaScrpt DOM API is actually quite capable in its own right, and since IE < 11 has been <u>officially abandoned</u>, it can now be used without any worries.

In this article, I'll demonstrate how to accomplish some of the most common DOM manipulation tasks with plain JavaScript, namely:

■   querying and modifying the DOM,

- modifying classes and attributes,
- listening to events, and
- animation.

I'll finish off by showing you how to create your own super slim DOM-library that you can drop into any project. Along the way, you'll learn that DOM manipulation with vanilla JS is not rocket science and that many jQuery methods in fact have direct equivalents in the native API.

So let's get to it ...

DOM Manipulation: Querying the DOM

This Isn't An Exhaustive Guide to the DOM API

I won't explain the Vanilla DOM API in full detail, but only scratch the surface. In the usage examples, you may encounter methods I haven't introduced explicitly. In this case just refer to the excellent Mozilla Developer Network for details.

The DOM can be queried using the `.querySelector()` method, which takes an arbitrary CSS selector as an argument:

```
const myElement = document.querySelector('#foo >
↪ div.bar')
```

This will return the first match (depth first). Conversely, we can check if an element matches a selector:

```
myElement.matches('div.bar') === true
```

If we want to get all occurrences, we can use:

```
const myElements = document.querySelectorAll('.bar')
```

If we already have a reference to a parent element, we can just query that element's children instead of the whole `document`. Having narrowed down the context like this, we can simplify selectors and increase performance.

```
 const myChildElemet =
↪ myElement.querySelector('input[type="submit"]')

// Instead of
 // document.querySelector('#foo > div.bar
↪ input[type="submit"]')
```

Then why use those other, less convenient methods like `.getElementsByTagName()` at all? Well, one important difference is that the result of `.querySelector()` is *not live*, so when we dynamically add an element (see section 3 for details) that matches a selector, the collection won't update.

```
const elements1 = document.querySelectorAll('div')
const elements2 = document.getElementsByTagName('div')
const newElement = document.createElement('div')

document.body.appendChild(newElement)
elements1.length === elements2.length // false
```

Another consideration is that such a live collection doesn't need to have all of the information up front, whereas `.querySelectorAll()` immediately gathers everything in a static list, making it less performant.

Working with Nodelists

Now there are two common gotchas regarding `.querySelectorAll()`. The first one is that we can't call Node methods on the result and propagate them to its elements (like you might be used from jQuery objects). Rather we have to explicitly iterate over those elements. And this is the other gotcha: the return value is a NodeList, not an Array. This means the usual Array methods are not available directly. There are a few corresponding NodeList implementations such

as `.forEach`, which however are still not supported by any IE. So we have to convert the list to an array first, or "borrow" those methods from the Array prototype.

```
// Using Array.from()
Array.from(myElements).forEach(doSomethingWithEachElement)

// Or prior to ES6
 Array.prototype.forEach.call(myElements,
↪ doSomethingWithEachElement)

// Shorthand:
[].forEach.call(myElements, doSomethingWithEachElement)
```

Each element also has a couple of rather self-explanatory read-only properties referencing the "family", all of which are live:

```
myElement.children
myElement.firstElementChild
myElement.lastElementChild
myElement.previousElementSibling
myElement.nextElementSibling
```

As the Element interface inherits from the Node interface, the following properties are also available:

```
myElement.childNodes
myElement.firstChild
myElement.lastChild
myElement.previousSibling
myElement.nextSibling
myElement.parentNode
myElement.parentElement
```

Where the former only reference elements, the latter (except for `.parentElement`) can be any kind of node, e.g. text nodes. We can then check the <u>type</u> of a given node like e.g.

```
 myElement.firstChild.nodeType === 3 // this would be a text
↪ node
```

As with any object, we can check a node's prototype chain using the `instanceof` operator:

```
myElement.firstChild.nodeType instanceof Text
```

Modifying Classes and Attributes

Modifying classes of elements is as easy as:

```
myElement.classList.add('foo')
myElement.classList.remove('bar')
myElement.classList.toggle('baz')
```

You can read a more in-depth disussion of how to modify classes in this <u>quick tip by Yaphi Berhanu</u>. Element properties can be accessed like any other object's properties

```
// Get an attribute value
const value = myElement.value

// Set an attribute as an element property
myElement.value = 'foo'

// Set multiple properties using Object.assign()
Object.assign(myElement, {
  value: 'foo',
```

```
  id: 'bar'
})

// Remove an attribute
myElement.value = null
```

Note that there are also the methods `.getAttribute()`, `.setAttribute()` and `.removeAttribute()`. These directly modify the *HTML attributes* (as opposed to the DOM properties) of an element, thus causing a browser redraw (you can observe the changes by inspecting the element with your browser's dev tools). Not only is such a browser redraw more expensive than just setting DOM properties, but these methods also can have <u>unexpected results</u>.

As a rule of thumb, only use them for attributes that don't have a corresponding DOM property (such as `colspan`), or if you really want to "persist" those changes to the HTML (e.g. to keep them when cloning an element or modifying its parent's `.innerHTML` — see <u>section 3</u>).

Adding CSS styles

CSS rules can be applied like any other property; note though that the properties are camel-cased in JavaScript:

```
myElement.style.marginLeft = '2em'
```

If we want certain values, we can obtain these via the `.style` property. However, this will only give us styles that have been explicitly applied. To get the computed values, we can use, `.window.getComputedStyle()`. It takes the element and returns a <u>CSSStyleDeclaration</u> containing all styles from the element itself as well as those inherited from its parents:

```
window.getComputedStyle(myElement).getPropertyValue('margin-l
↳ eft')
```

Modifying the DOM

We can move elements around like so:

```
// Append element1 as the last child of element2
element1.appendChild(element2)

 // Insert element2 as child of element 1, right before
↪ element3
element1.insertBefore(element2, element3)
```

If we don't want to move the element, but insert a copy, we can clone it like so:

```
// Create a clone
const myElementClone = myElement.cloneNode()
myParentElement.appendChild(myElementClone)
```

The `.cloneNode()` method optionally takes a boolean as argument; if set to true, a *deep* copy will be created, meaning its children are also cloned.

Of course, we can just as well create entirely new elements or text nodes:

```
const myNewElement = document.createElement('div')
const myNewTextNode = document.createTextNode('some text')
```

which we can then insert as shown above. If we want to remove an element, we can't do so directly, but we *can* remove children from a parent element, like so:

```
myParentElement.removeChild(myElement)
```

This gives us a nice little work around, meaning can actually remove an element indirectly, by referencing its parent element:

```
myElement.parentNode.removeChild(myElement)
```

Element properties

Every element also has the properties `.innerHTML` and `.textContent` (as well as `.innerText`, which is similar to `.textContent`, but has some <u>important differences</u>). These hold the HTML and plain text content respectively. They are writable properties, meaning we can modify elements and their contents directly:

```
// Replace the inner HTML
myElement.innerHTML = `
  <div>
    <h2>New content</h2>
    <p>beep boop beep boop</p>
  </div>
`

// Remove all child nodes
myElement.innerHTML = null

// Append to the inner HTML
myElement.innerHTML += `
  <a href="foo.html">continue reading...</a>
  <hr/>
`
```

Appending markup to the HTML as shown above is usually a bad idea though, as we'd lose any previously made property changes on the affected elements (unless we persisted those changes as HTML attributes as shown in <u>section 2</u>) and bound event listeners. Setting the `.innerHTML` is good for completely throwing away markup and replacing it with something else, e.g. server-rendered markup. So appending elements would better be done like so:

```
const link = document.createElement('a')
const text = document.createTextNode('continue reading...')
```

```
const hr = document.createElement('hr')

link.href = 'foo.html'
link.appendChild(text)
myElement.appendChild(link)
myElement.appendChild(hr)
```

With this approach however we'd cause two browser redraws — one for each
appended element — whereas changing the `.innerHTML` only causes one. As a
way around this performance issue we can first assemble all nodes in a
DocumentFragment, and then just append that single fragment:

```
const fragment = document.createDocumentFragment()

fragment.appendChild(text)
fragment.appendChild(hr)
myElement.appendChild(fragment)
```

Listening to events

This is possibly the best known way to bind an event listener:

```
myElement.onclick = function onclick (event) {
    console.log(event.type + ' got fired')
}
```

But this should generally be avoided. Here, `.onclick` is a property of the element,
meaning that you can change it, but you cannot use it to add additional listeners
— by reassigning a new function you'll overwrite the reference to the old one.

Instead, we can use the much mightier `.addEventListener()` method to add as
many events of as many types as we like. It takes three arguments: the event type
(such as `click`), a function that gets called whenever the event occurs on the

element (this function gets passed an event object), and an optional config object which will be explained further below.

```
myElement.addEventListener('click', function (event) {
  console.log(event.type + ' got fired')
})

myElement.addEventListener('click', function (event) {
  console.log(event.type + ' got fired again')
})
```

Within the listener function, `event.target` refers to the element on which the event was triggered (as does `this`, unless of course we're using an arrow function). Thus you can easily access its properties like so:

```
// The `forms` property of the document is an array holding
// references to all forms
const myForm = document.forms[0]
const myInputElements = myForm.querySelectorAll('input')

Array.from(myInputElements).forEach(el => {
  el.addEventListener('change', function (event) {
    console.log(event.target.value)
  })
})
```

Preventing default actions

Note that `event` is always available within the listener function, but it is good practive to explitly pass it in anyway when needed (and we can name it as we like then, of course). Without elaborating on the Event interface itself, one particularly noteworthy method is `.preventDefault()`, which will, well, prevent the browser's default behavior, such as following a link. Another common use-case would be to conditionally prevent the submission of a form if the client-side form-validation fails.

```
myForm.addEventListener('submit', function (event) {
  const name = this.querySelector('#name')

  if (name.value === 'Donald Duck') {
    alert('You gotta be kidding!')
    event.preventDefault()
  }
})
```

Another important event method is `.stopPropagation()`, which will prevent the event from bubbling up the DOM. This means that if we have a propagation-stopping click listener (say) on an element, and another click listener on one of its parents, a click event that gets triggered on the child element won't get triggered on the parent — otherwise, it would get triggered on both.

Now `.addEventListener()` takes an optional config object as a 3rd argument, which can have any of the following boolean properties (all of which default to `false`):

- `capture`: The event will be triggered on the element before any other element beneath it in the DOM (event capturing and bubbling is an article in its own right, for more details have a look [here](#))
- `once`: As you might guess, this indicates that the event will get triggered only once
- `passive`: This means that `event.preventDefault()` will be ignored (and usually yield a warning in the console)

The most common option is `.capture`; in fact, it is so common that there's a shorthand for this: instead of specifying it in the config object, you can just pass in a boolean here:

```
myElement.addEventListener(type, listener, true)
```

Event listeners can be removed using `.removeEventListener()`, which takes the event type and a reference to the callback function to be removed; for example, the `once` option could also be implemented like

```
 myElement.addEventListener('change', function listener
↪ (event) {
   console.log(event.type + ' got triggered on ' + this)
   this.removeEventListener('change', listener)
})
```

Event delegation

Another useful pattern is *event delegation*: say we have a form and want to add a `change` event listener to all of its `input` children. One way to do so would be iterating over them using `myForm.querySelectorAll('input')` as shown above. However, this is unnecessary when we can just as well add it to the form itself and check the contents of `event.target`.

```
myForm.addEventListener('change', function (event) {
   const target = event.target
   if (target.matches('input')) {
     console.log(target.value)
   }
})
```

Another advantage of this pattern is that it automatically accounts for dynamically inserted children as well, without having to bind new listeners to each.

Animation

Usually, the cleanest way to perform animations is to apply CSS classes with a `transition` property, or use CSS `@keyframes`. But if you need more flexibility (e.g. for a game), this can be done with JavaScript as well.

The naive approach would be to have a `window.setTimeout()` function call itself until the desired animation is completed. However, this inefficiently forces rapid document reflows; and this *layout thrashing* can quickly lead to stuttering, expecially on mobile devices. Intead, we can sync the updates using

`window.requestAnimationFrame()` to schedule all current changes to the next browser repaint frame. It takes a callback as argument which receives the current (high res) timestamp:

```
const start = window.performance.now()
const duration = 2000

window.requestAnimationFrame(function fadeIn (now)) {
  const progress = now - start
  myElement.style.opacity = progress / duration

  if (progress < duration) {
    window.requestAnimationFrame(fadeIn)
  }
}
```

This way we can achieve very smooth animations. For a more detailed discussion, please have a look at <u>this</u> article by Mark Brown.

Writing your own helper methods

True, always having to iterate over elements to do something with them may be rather cumbersome compared to jQuery's concise and chainable `$('.foo').css({color: 'red'})` syntax. So why not simply write our own shorthand methods for things like this?

```
const $ = function $ (selector, context = document) {
 const elements =
↪ Array.from(context.querySelectorAll(selector))

  return {
    elements,

    html (newHtml) {
      this.elements.forEach(element => {
        element.innerHTML = newHtml
      })
```

```
      return this
    },

    css (newCss) {
      this.elements.forEach(element => {
        Object.assign(element.style, newCss)
      })

      return this
    },

    on (event, handler, options) {
      this.elements.forEach(element => {
        element.addEventListener(event, handler, options)
      })

      return this
    }

    // etc.
  }
}
```

Thus we have a super slim DOM-library with only the methods that we really need, and without all the backwards-compatibility weight. Usually we'd have those methods in our collection's prototype though. Here's a (somewhat more elaborate) gist with some ideas how to implement such helpers. Alternatively, we might keep it as simple as

```
 const $ = (selector, context = document) =>
↪ context.querySelector(selector)
 const $$ = (selector, context = document) =>
↪ context.querySelectorAll(selector)

const html = (nodeList, newHtml) => {
  Array.from(nodeList).forEach(element => {
    element.innerHTML = newHtml
```

```
   })
}

// And so on...
```

Demo

To round this article off, here's a <u>CodePen which demonstrates many of the</u> <u>concepts explained above to implement a simple lightbox technique</u>.

Conclusion

I hope I could show that DOM manipulation with plain JavaScript is not rocket science and that in fact, many jQuery methods have direct equivalents in the native DOM API. This means that for some everyday use cases (such as a navigation menu or a modal popup), the additional overhead of a DOM library may be out of place.

And while it's true that some parts of the native API are verbose or inconvenient (such as having to iterate over node lists manually all the time), we can quite easily write our own small helper functions to abstract away such repetitive tasks.

Chapter

4

A Beginner's Guide to Webpack 2 and Module Bundling

by Mark Brown

Webpack has become one of the most important tools for modern web development. Primarily it's a module bundler for your JavaScript but it can be taught to transform all of your front-end assets like HTML and CSS, even images. It can give you more control over the number of HTTP requests your app is making and allows you to use other flavors of those assets (Jade, Sass & ES6 for example). Webpack also allows you to easily consume packages from npm.

This article is aimed at those who are new to webpack and will cover initial setup and configuration, modules, loaders, plugins, code splitting and hot module replacement. If you find video tutorials helpful I can highly recommend Glen

Maddern's <u>Webpack from First Principles</u> as a starting point to understand what it is that makes webpack special.

To follow along at home you'll need to have <u>Node.js installed</u>. You can also <u>download the demo app from our Github repo</u>.

Setup

Let's initialize a new project with npm and install webpack:

```
mkdir webpack-demo
cd webpack-demo
npm init -y
npm install webpack@beta --save-dev
mkdir src
touch index.html src/app.js webpack.config.js
```

Edit these files:

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello webpack</title>
  </head>
  <body>
    <div id="root"></div>
    <script src="dist/bundle.js"></script>
  </body>
</html>
```

```
// src/app.js
const root = document.querySelector('#root')
root.innerHTML = `<p>Hello webpack.</p>`
```

```
// webpack.config.js
const webpack = require('webpack')
const path = require('path')

const config = {
  context: path.resolve(__dirname, 'src'),
  entry: './app.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  },
  module: {
    rules: [{
      test: /\.js$/,
      include: path.resolve(__dirname, 'src'),
      use: [{
        loader: 'babel-loader',
        options: {
          presets: [
            ['es2015', { modules: false }]
          ]
        }
      }]
    }]
  }
}

module.exports = config
```

The config above is a common starting point, it instructs webpack to compile our entry point `src/app.js` into our output `/dist/bundle.js` and all `.js` files will be transpiled from ES2015 to ES5 with Babel.

To get this running we're going to need to install three packages, `babel-core`, the webpack *loader* `babel-loader` and the *preset* `babel-preset-es2015` for the flavor of JavaScript we want to write. `{ modules: false }` enables <u>Tree Shaking</u> to remove unused exports from your bundle to bring down the file size.

```
 npm install babel-core babel-loader babel-preset-es2015
↳ --save-dev
```

Lastly, replace the scripts section of `package.json` with the following:

```
"scripts": {
  "start": "webpack --watch",
  "build": "webpack -p"
},
```

Running `npm start` from the command line will start webpack in *watch mode* which will recompile our bundle whenever a `.js` file is changed in our `src` directory. The output in the console tells us about the bundles being being created, it's important to keep an eye on the number of bundles and the size.

```
Webpack is watching the files…

Hash: 639c19100c96f21917ab
Version: webpack 2.2.0
Time: 514ms
      Asset       Size  Chunks                    Chunk Names
  bundle.js    2.61 kB       0  [emitted]  main
      [0] ./app.js 99 bytes {0} [built]
```

You should now be able to load `index.html` in your browser and be greeted with "Hello webpack.".

```
open index.html
```

Open up `dist/bundle.js` to see what webpack has done, at the top is webpack's module bootstrapping code and right at the bottom is our module. You may not be colored impressed just yet but if you've come this far you can now start authoring ES6 modules and webpack will be able to produce a bundle for production that will work in all browsers.

Stop webpack with `Ctrl + C` and run `npm run build` to compile our bundle in *production mode.*

Notice that the bundle size has come down from **2.61 kB** to **585 bytes**. Take another look at `dist/bundle.js` and you'll see a big ugly mess of code, our bundle has been minified with UglifyJS, the code will run exactly the same but it's done with the fewest characters needed.

Modules

Out of the box webpack knows how to consume JavaScript modules in a variety of formats, the most notable two are:

- ES2015 `import` statements
- CommonJS `require()` statements

We can test this out by installing <u>lodash</u> and importing it from `app.js`

```
npm install lodash --save
```

```
// src/app.js
import {groupBy} from 'lodash/collection'

const people = [{
  manager: 'Jen',
  name: 'Bob'
}, {
  manager: 'Jen',
  name: 'Sue'
}, {
  manager: 'Bob',
  name: 'Shirley'
}, {
  manager: 'Bob',
  name: 'Terrence'
}]
const managerGroups = groupBy(people, 'manager')
```

```
const root = document.querySelector('#root')
 root.innerHTML = `<pre>${JSON.stringify(managerGroups,
↪ null, 2)}</pre>`
```

Run npm start to start webpack and refresh index.html, you should see an array of people grouped by manager.

Let's move the array of people into its own module people.js

```
// src/people.js
const people = [{
  manager: 'Jen',
  name: 'Bob'
}, {
  manager: 'Jen',
  name: 'Sue'
}, {
  manager: 'Bob',
  name: 'Shirley'
}, {
  manager: 'Bob',
  name: 'Terrence'
}]

export default people
```

We can simply import it from app.js with a *relative* path.

```
// src/app.js
import {groupBy} from 'lodash/collection'
import people from './people'

const managerGroups = groupBy(people, 'manager')

const root = document.querySelector('#root')
 root.innerHTML = `<pre>${JSON.stringify(managerGroups,
```

```
↪ null, 2)}</pre>`
```

Note: Imports without a relative path like `'lodash/collection'` are modules from npm installed to `/node_modules`, your own modules will always need a relative path like `'./people'`, this is how you can tell them apart.

Loaders

We've already been introduced to `babel-loader`, one of many <u>loaders</u> that you can configure to tell webpack what to do when it encounters imports for different file types. You can chain loaders together into a series of transforms, a good way to see how this works is by importing Sass from our JavaScript.

Sass

This transformation involves three separate loaders and the `node-sass` library:

```
 npm install css-loader style-loader sass-loader node-sass
↪ --save-dev
```

Add a new rule to our config file for `.scss` files.

```
// webpack.config.js
rules: [{
  test: /\.scss$/,
  use: [
    'style-loader',
    'css-loader',
    'sass-loader'
  ]
}, {
  // ...
}]
```

Note: Whenever you change any of loading rules in `webpack.config.js` you'll need to restart the build with `Ctrl + C` and `npm start`.

The array of loaders are processed in reverse order:

- `sass-loader` transforms Sass into CSS.
- `css-loader` parses the CSS into JavaScript and resolves any dependencies.
- `style-loader` outputs our CSS into a `<style>` tag in the document.

You can think of these as function calls, the output of one loader feeds as input into the next.

```
styleLoader(cssLoader(sassLoader('source')))
```

Let's add a Sass source file:

```
/* src/style.scss */
$bluegrey: #2B3A42;

pre {
  padding: 20px;
  background: $bluegrey;
  color: #dedede;
  text-shadow: 0 1px 1px rgba(#000, .5);
}
```

You can now require Sass directly from your JavaScript, import it from the top of `app.js`.

```
// src/app.js
import './style.scss'

// ...
```

Reload `index.html` and you should see some styling.

CSS in JS

We just imported a Sass file from our JavaScript, as a module.

Open up `dist/bundle.js` and search for "pre {". Indeed, our Sass has been compiled to a string of CSS and saved as a module within our bundle. When we import this module in our JavaScript, `style-loader` outputs that string into an embedded `<style>` tag.

I know what you're thinking. **Why?**

I won't delve too far into this topic here, but here are a few reasons to consider:

- A JavaScript component you may want to include in your project may *depend* on other assets to function properly (HTML, CSS, Images, SVG), if these can all be bundled together it is far easier to import and use.
- Dead code elimination: When a JS component is no longer imported by your code, the CSS will no longer be imported either. The bundle produced will only ever contain code that does something.
- CSS Modules: The global namespace of CSS makes it very difficult to be confident that a change to your CSS will not have any side effects. CSS modules change this by making CSS local by default and exposing unique class names that you can reference in your JavaScript.
- Bring down the number of HTTP requests by bundling / splitting code in clever ways.

Images

The last example of loaders we'll look at is the handling of images with `url-loader`.

In a standard HTML document images are fetched when the browser encounters an `<img>` tag or an element with a `background-image` property. With webpack you can optimize this in the case of small images by storing the source of the images as strings inside your JavaScript. By doing this you preload them and the browser won't have to fetch them with separate requests later.

```
npm install file-loader url-loader --save-dev
```

Add one more rule for loading images:

```js
// webpack.config.js
rules: [{
  test: /\.(png|jpg)$/,
  use: [{
    loader: 'url-loader',
 options: { limit: 10000 } // Convert images < 10k to
↪ base64 strings
  }]
}, {
  // ...
}]
```

Restart the build with `Ctrl + C` and `npm start`.

Download a <u>test image</u> with this command:

```
 curl
↪ https://raw.githubusercontent.com/sitepoint-editors/webpack-demo/
master/src/code.png
↪ --output src/code.png
```

You can now import the image source from the top of `app.js`:

```js
// src/app.js
import codeURL from './code.png'
const img = document.createElement('img')
img.src = codeURL
img.style.backgroundColor = "#2B3A42"
img.style.padding = "20px"
img.width = 32
document.body.appendChild(img)
```

```
// ...
```

This will include an image where the `src` attribute contains a <u>data URI</u> of the image itself.

```
 <img src="data:image/png;base64,iVBO..."
↪ style="background: #2B3A42; padding: 20px" width="32">
```

Also, thanks to `css-loader` images referenced with `url()` also run through `url-loader` to *inline* them directly in the CSS.

```
/* src/style.scss */
pre {
 background: $bluegrey url('code.png') no-repeat center
↪ center / 32px 32px;
}
```

Compiles to this

```
pre {
 background: #2b3a42 url("data:image/png;base64,iVBO...")
↪ no-repeat scroll center center / 32px 32px;
}
```

Modules to Static Assets

You should be able to see now how loaders help to build up a tree of dependencies amongst your assets, this is what the image on the webpack homepage is demonstrating.
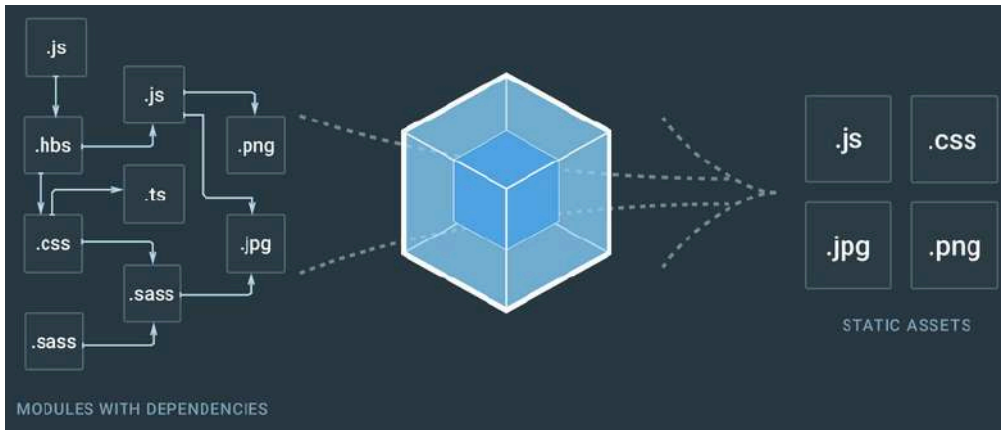
Though JavaScript is the entry point, webpack appreciates that your other asset types like HTML, CSS, and SVG each have dependencies of their own which should be considered as part of the build process.

Plugins

We've seen one example of a built in webpack plugin already, `webpack -p` which is called from our `npm run build` script uses the <u>UglifyJsPlugin</u> which ships with webpack to minify our bundle for production.

While *loaders* operate transforms on single files *plugins* operate across larger chunks of code.

Common code

The <u>commons-chunk-plugin</u> is another core plugin that ships with webpack that can be used to create a separate module with shared code across multiple entry points. Until now we've been using a single entry point and single output bundle. There are many <u>real-world scenarios</u> where you'll benefit from splitting this into multiple entry and output files.

If you have two distinct areas of your application that both share modules, for example `app.js` for a public facing app and `admin.js` for an administration area you can create separate entry points for them like so:

```
// webpack.config.js
const webpack = require('webpack')
const path = require('path')

 const extractCommons = new
↪ webpack.optimize.CommonsChunkPlugin({
  name: 'commons',
  filename: 'commons.js'
})

const config = {
  context: path.resolve(__dirname, 'src'),
  entry: {
    app: './app.js',
    admin: './admin.js'
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].bundle.js'
  },
  module: {
    // ...
  },
  plugins: [
    extractCommons
  ]
}

module.exports = config
```

Notice the change to the `output.filename` which now includes `[name]`, this is replaced with the chunk name so we can expect two output bundles from this configuration: `app.bundle.js` and `admin.bundle.js` for our two entry points.

The `commonschunk` plugin generates a third file `commons.js` which includes shared modules from our entry points.

```
// src/app.js
import './style.scss'
import {groupBy} from 'lodash/collection'
import people from './people'

const managerGroups = groupBy(people, 'manager')

const root = document.querySelector('#root')
 root.innerHTML = `<pre>${JSON.stringify(managerGroups,
↪ null, 2)}</pre>`
```

```
// src/admin.js
import people from './people'

const root = document.querySelector('#root')
 root.innerHTML = `<p>There are ${people.length}
↪ people.</p>`
```

These entry points would output the following files:

- **app.bundle.js** includes the `style` and `lodash/collection` modules
- **admin.bundle.js** doesn't include extra modules
- **commons.js** includes our `people` module

We could then include the commons chunk in both areas:

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello webpack</title>
  </head>
  <body>
    <div id="root"></div>
    <script src="dist/commons.js"></script>
```

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

```
<!-- admin.html -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello webpack</title>
  </head>
  <body>
    <div id="root"></div>
    <script src="dist/commons.js"></script>
    <script src="dist/admin.bundle.js"></script>
  </body>
</html>
```

Try loading `index.html` and `admin.html` in the browser to see them run with the automatically created commons chunk.

Extracting CSS

Another popular plugin is the extract-text-webpack-plugin which can be used to extract modules into their own output files.

Below we'll modify our `.scss` rule to compile our Sass, load the CSS, then *extract* each into its own CSS bundle, thus removing it from our JavaScript bundle.

```
 npm install extract-text-webpack-plugin@2.0.0-beta.4
↳ --save-dev
```

```
// webpack.config.js
 const ExtractTextPlugin =
```

```
↪ require('extract-text-webpack-plugin')
 const extractCSS = new
↪ ExtractTextPlugin('[name].bundle.css')

const config = {
  // ...
  module: {
    rules: [{
      test: /\.scss$/,
      loader: extractCSS.extract(['css-loader','sass-loader'])
    }, {
      // ...
    }]
  },
  plugins: [
    extractCSS,
    // ...
  ]
}
```

Restart webpack and you should see a new bundle `app.bundle.css` which you
can link to directly, as usual.

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello webpack</title>
    <link rel="stylesheet" href="dist/app.bundle.css">
  </head>
  <body>
    <div id="root"></div>
    <script src="dist/commons.js"></script>
    <script src="dist/app.bundle.js"></script>
  </body>
</html>
```

Refresh the page to confirm our CSS has been compiled and moved from `app.bundle.js` to `app.bundle.css`. Success!

Code Splitting

We've looked at a few ways to split our code already:

- Manually creating separate **entry points**
- Automatic splitting of shared code into a **commons chunk**
- Extracting chunks out of our compiled bundle with **extract-text-webpack-plugin**

Another way to split our bundle is with <u>System.import</u> or <u>require.ensure</u>. By wrapping sections of code in these functions you create a chunk to be loaded on demand at run time. This can significantly improve load time performance by not sending everything to the client at the start. `System.import` takes the module name as an argument and returns a Promise. `require.ensure` takes a list of dependencies, a callback and an optional name for the chunk.

If one section of your app has heavy dependencies that the rest of the app doesn't need, it's a good case for splitting into its own bundle. We can demonstrate this by adding a new module named `dashboard.js` that requires <u>d3</u>.

```
npm install d3 --save
```

```
// src/dashboard.js
import * as d3 from 'd3'

console.log('Loaded!', d3)

export const draw = () => {
  console.log('Draw!')
}
```

Import `dashboard.js` from the bottom of `app.js`

```
// ...

const routes = {
  dashboard: () => {
    System.import('./dashboard').then((dashboard) => {
      dashboard.draw()
    }).catch((err) => {
      console.log("Chunk loading failed")
    })
  }
}

// demo async loading with a timeout
setTimeout(routes.dashboard, 1000)
```

Because we've added asynchronous loading of modules, we need an
`output.publicPath` property in our config so that webpack knows where to fetch
them.

```
// webpack.config.js

const config = {
  // ...
  output: {
    path: path.resolve(__dirname, 'dist'),
    publicPath: '/dist/',
    filename: '[name].bundle.js'
  },
  // ...
}
```

Restart the build and you'll see a mysterious new bundle `0.bundle.js`

```
        Asset       Size  Chunks                     Chunk Names
   0.bundle.js     451 kB       0  [emitted]  [big]
 app.bundle.js     186 kB       1  [emitted]         app
admin.bundle.js  462 bytes      2  [emitted]         admin
   commons.js     5.93 kB       3  [emitted]         commons
```

Notice how webpack keeps you honest by highlighting the [big] bundles for you to keep an eye on.

This 0.bundle.js will be fetched on demand with a JSONP request, so loading the file directly from the file system isn't going to cut it anymore. We'll need to run a server, any server will do.

```
python -m SimpleHTTPServer 8001
```

Open http://localhost:8001/

One second after loading you should see a GET request for our dynamically generated bundle /dist/0.bundle.js and "Loaded!" logged to the console. Success!

Webpack Dev Server

Live reloading can really improve the developer experience by refreshing automatically whenever files are changed. Simply install it and start it with webpack-dev-server and you're off to the races.

```
npm install webpack-dev-server@2.2.0-rc.0 --save-dev
```

Modify the start script in package.json

```
"start": "webpack-dev-server --inline",
```

Run npm start to start the server and open http://localhost:8080 in your browser.

Try it out by changing any of the `src` files, e.g. change a name in `people.js` or a style in `style.scss` to see it transform before your very eyes.

Hot Module Replacement

If you're impressed by live reloading, <u>hot module replacement</u> (HMR) will knock your socks off.

It's the year 2017, chances are that you've already worked on single-page apps with global state. During development you'll be making a lot of small changes to components and you want to see these reflected in a real browser where you see the output and interact with it. By refreshing the page manually or with live reload your global state is blown away and you need to start from scratch. Hot module replacement has forever changed this.

In the developer workflow of your dreams you can make changes to a module and it is compiled and swapped out at run time without refreshing the browser (blowing away the local state) or touching other modules. There are still times when a manual refresh is required, but HMR can still save you a massive amount of time and it feels like the future.

Make one final edit to the `start` script in `package.json`.

```
"start": "webpack-dev-server --inline --hot",
```

At the top of `app.js` tell webpack to accept hot reloading of this module and any of its dependencies.

```
if (module.hot) {
  module.hot.accept()
}

// ...
```

Note: `webpack-dev-server --hot` sets `module.hot` to `true` which includes this for development only. When building in *production mode* `module.hot` is set to `false` so these are stripped out of the bundle.

Add `NamedModulesPlugin` to the list of plugins in `webpack.config.js` to improve logging in the console.

```
plugins: [
  new webpack.NamedModulesPlugin(),
  // ...
]
```

Lastly, add an `<input>` element to the page where we can add some text to confirm a full page refresh doesn't occur when we make a change to our module.

```
<body>
  <input />
  <div id="root"></div>
  ...
```

Restart the server with `npm start` and behold hot reloading!

To try this out, enter "HMR Rules" in the input and then change a name in `people.js` to see it swapped out *without* refreshing the page and losing the state of the input.

This is a simple example but hopefully you can see how wildly useful this is. It's especially true with *component* based development like React where you have a lot of "dumb" components separated from their state, components can be swapped out and re-rendered without losing state so you get an instant feedback loop.

Hot Reloading CSS

Change the background color of the `<pre>` element in `style.scss` and you'll notice that it's *not* being replaced with HMR.

```
pre {
  background: red;
}
```

It turns out that HMR of CSS comes for free when you're using `style-loader`, you don't need to do anything special. We just broke this link in the chain by extracting the CSS modules out into external CSS files which *can't* be replaced.

If we revert our Sass rule to its original state and remove `extractCSS` from the list of plugins, you'll be able see hot reloading of your Sass too.

```
{
  test: /\.scss$/,
  loader: ['style-loader', 'css-loader','sass-loader']
}
```

HTTP/2

One of the primary benefits of using a module bundler like webpack is that it can help you improve performance by giving you control over how the assets are *built* and then *fetched* on the client. It has been considered best practice for years to concatenate files to reduce the number of requests that need to made on the client. This is still valid but HTTP/2 now allows multiple files to be delivered in a *single* request so concatenation isn't a silver bullet anymore. Your app may actually benefit from having many small files individually cached, the client could then fetch a single changed module rather than having to fetch an entire bundle again with *mostly* the same contents.

The creator of Webpack Tobias Koppers has written an informative post explaining why bundling is still important, even in the HTTP/2 era.

Read more about this over at webpack & HTTP/2.

Over to You

I sure hope you have found this introduction to webpack 2 helpful and are able to start using it to great effect. It can take a little time to wrap your head around webpack's configuration, loaders and plugins but learning how this tool works will pay off.

The documentation is still being worked on but there's a handy <u>Migrating from v1 to v2</u> guide if you want to shift an existing Webpack 1 project across to the new hotness.

Chapter **5**

React vs Angular: An In-depth Comparison

by Pavels Jelisejevs

Peer reviewed by <u>Jurgen Van de Moere</u> and <u>Joan Yin</u>.

Should I choose Angular, or React? Today's bipolar landscape of JavaScript frameworks has left many developers struggling to pick a side in this debate. Whether you're a newcomer trying to figure out where to start, a freelancer picking a framework for your next project or an enterprise-grade architect planning a strategic vision for your company, you're likely to benefit from having an educated view on this topic.

To save you some time, let me tell you something up front: this article won't give a clear answer on which framework is better. But neither will hundreds of other articles with similar titles. I can't tell you that because the answer depends on a

wide range of factors which make a particular technology more or less suitable for your environment and use case.

Since we can't answer the question directly, we'll attempt something else. We'll compare Angular (2+, not the old AngularJS) and React to demonstrate how you can approach the problem of comparing any two frameworks in a structured manner on your own and tailor it to your environment. You know, the old "teach a man to fish" approach. That way, when both are replaced by a BetterFramework.js in a year's time, you will be able to re-create the same train of thought once more.

Where to Start?

Before you pick any tool you need to answer two simple questions: "Is this a good tool per se?" and "Will it work well for my use case?" None of them mean anything on their own, so you always need to keep both of them in mind. All right, the questions might not be that simple, so we'll try to break them down into smaller ones.

Questions on the tool itself:

- How mature is it and who's behind it?
- What kind of features does it have?
- What architecture, development paradigms, and patterns does it employ?
- What is the ecosystem around it?

Questions for self-reflection:

- Will I and my colleagues be able to learn this tool with ease?
- Does is fit well with my project?
- What is the developer experience like?

Using this set of questions you can start your assessment of any tool and we'll base our comparison of React and Angular on them as well.

There's another thing we need to take into account. Strictly speaking, it's not exactly fair to compare Angular to React, since Angular is a full-blown feature-

rich framework, and React just a UI component library. To even the odds, we'll talk about React in conjunction with some of the libraries often used with it.

Maturity

An important part of being a skilled developer is being able to keep the balance between established, time-proven approaches and evaluating new bleeding-edge tech. As a general rule, you should be careful when adopting tools which have not yet matured due to certain risks:

- The tool might be buggy and unstable.
- It might be unexpectedly abandoned by the vendor.
- There might not be a large knowledge base or community available in case you need help.

Both React and Angular come from good families, so it seems that we can be confident in this regard.

React

React is developed and maintained by Facebook and used in their own products, including Instagram and WhatsApp. It has been around for roughly three and a half years now, so it's not exactly new. It's also one of the most popular projects on GitHub with about 60,000 stars at the time of writing. Sounds good to me.

Angular

Angular (version 2 and above) has been around less then React, but if you count in the history of its predecessor, AngularJS, the picture evens out. It's maintained by Google and used in AdWords and Google Fiber. Since AdWords is one of the key projects in Google, it is clear they have made a big bet on it and is unlikely to disappear anytime soon.

Features

Like I mentioned earlier, Angular has more features out of the box than React. This can be both a good and a bad thing, depending on how you look at it.

Both frameworks share some key features in common: components, data binding, and platform-agnostic rendering.

Angular

Angular provides a lot of the features required for a modern web application out of the box. Some of the standard features are:

- Dependency injection;
- Templates, based on an extended version of HTML;
- Routing, provided by @angular/router;
- Ajax requests by @angular/http;
- @angular/forms for building forms;
- Component CSS encapsulation;
- XSS protection;
- Utilities for unit-testing components.

Having all of these features available out of the box is highly convenient when you don't want to spend time picking the libraries yourself. However, it also means that you're stuck with some of them, even if you don't need them. And replacing them will usually require additional effort. For instance, we believe that for small projects having a DI system creates more overhead than benefit, considering it can be effectively replaced by imports.

React

With React, you're starting off with a more minimalistic approach. If we're looking at just React, here's what we have:

- No dependency injection;
- Instead of classic templates it has JSX, an XML-like language built on top of JavaScript;

- XSS protection;
- Utilities for unit-testing components.

Not much. And this can be a good thing. It means that you have the freedom to choose whatever additional libraries to add based on your needs. The bad thing is that you actually have to make those choices yourself. Some of the popular libraries that are often used together with React are:

- React-router for routing.
- Fetch (or axios) for HTTP requests;
- A wide variety of techniques for CSS encapsulation;
- Enzyme for additional unit-testing utilities.

We've found the freedom of choosing your own libraries liberating. This gives us the ability to tailor our stack to particular requirements of each project and we didn't find the cost of learning new libraries that high.

Languages, Paradigms, and Patterns.

Taking a step back from the features of each framework, let's see what kind higher-level concepts are popular with both frameworks.

React

There are several important things that come to mind when thinking about React: JSX, Flow, and Redux.

JSX

JSX is a controversial topic for many developers: some enjoy it, and others think that it's a huge step back. Instead of following a classical approach of separating markup and logic, React decided to combine them within components using an XML-like language that allows you to write markup directly in your JavaScript code.

While the topic of mixing markup with JavaScript might be debatable, it has an indisputable benefit: static analysis. If you make an error in your JSX markup, the

compiler will emit an error instead of continuing in silence. This helps by instantly catching typos and other silly errors.

Flow

Flow is a type-checking tool for JavaScript also developed by Facebook. It can parse code and check for common type errors such as implicit casting or null dereferencing.

Unlike TypeScript, which has a similar purpose, it does not require you to migrate to a new language and annotate your code for type checking to work. In Flow, type annotations are optional and can be used to provide additional hints to the analyzer. This makes Flow a good option if you would like to use static code analysis, but would like to avoid having to rewrite your existing code.

[**Further reading**: Writing Better JavaScript with Flow]

Redux

Redux is a library that helps manage state changes in a clear manner. It was inspired by Flux but with some simplifications. The key idea of Redux is that the whole state of the application is represented by a single object, which is mutated by functions called reducers. Reducers themselves are pure functions and are implemented separately from the components. This enables better separation of concerns and testability.

If you're working on a simple project, then introducing Redux might be an over complication, but for medium and large-scale projects, it's a solid choice. The library has become so popular that there are projects implementing it in Angular as well.

All three features can greatly improve your developer experience: JSX and Flow allow you to quickly spot places with potential errors, and Redux will help achieve a clear structure for your project.

Angular

Angular has a few interesting things up its sleeve as well, namely TypeScript and RxJS.

TypeScript

TypeScript is a new language built on top of JavaScript and developed by Microsoft. It's a superset of JavaScript ES2015 and includes features from newer versions of the language. You can use it instead of Babel to write state of the art JavaScript. It also features an extremely powerful typing system that can statically analyze your code by using a combination of annotations and type inference.

There's also a more subtle benefit. TypeScript has been heavily influenced by Java and .NET, so if your developers have a background in one of these languages, they are likely to find TypeScript easier to learn than plain JavaScript (notice how we switched from the tool to your personal environment). Although Angular has been the first major framework to actively adopt TypeScript, it's also possible to use it together with React.

[**Further reading**: An Introduction to TypeScript: Static Typing for the Web]

RxJS

RxJS is a reactive programming library which allows for more flexible handling of asynchronous operations and events. It's a combination of the Observer and Iterator patterns blended together with functional programming. RxJS allows you to treat anything as a continuous stream of values and perform various operations on it such as mapping, filtering, splitting or merging.

The library has been adopted by Angular in their HTTP module as well for some internal use. When you perform an HTTP request, it returns an Observable instead of the usual Promise. Although this library is extremely powerful, it's also quite complex. To master it, you'll need to know your way around different types of Observables, Subjects, as well as around a hundred methods and operators. Yikes, that seems to be a bit excessive just to make HTTP requests!

RxJS is useful in cases when you work a lot with continuous data streams such as web sockets, however, it seems overly complex for anything else. Anyway, when working with Angular you'll need to learn it at least on a basic level.

[**Further reading**: <u>Introduction to Functional Reactive Programming with RxJS</u>]

We've found TypeScript to be a great tool for improving the maintainability of our projects, especially those with a large code base or complex domain/business logic. Code written in TypeScript is more descriptive and easier to follow. Since TypeScript has been adopted by Angular, we hope to see even more projects using it. RxJS, on the other hand, seems only to be beneficial in certain cases and should be adopted with care. Otherwise, it can bring unwanted complexity to your project.

Ecosystem

The great thing about open source frameworks is the number of tools created around them. Sometimes, these tools are even more helpful than the framework itself. Let's have a look at some of the most popular tools and libraries associated with each framework.

Angular

Angular CLI

A popular trend with modern frameworks is having a CLI tool that helps you bootstrap your project without having to configure the build yourself. Angular has <u>Angular CLI</u> for that. It allows you to generate and run a project with just a couple of commands. All of the scripts responsible for building the application, starting a development server and running tests are hidden away from you in `node_modules`. You can also use it to generate new code during development. This makes setting up new projects a breeze.

[**Further reading**: <u>The Ultimate Angular CLI Reference</u>]

Ionic 2

Ionic 2 is a new version of the popular framework for developing hybrid mobile applications. It provides a Cordova container that is nicely integrated with Angular 2, and a pretty material component library. Using it, you can easily set up and build a mobile application. If you prefer a hybrid app over a native one, this is a good choice.

Material design components

If you're a fan of material design, you'll be happy to hear that there's a Material component library available for Angular. Currently, it's still at an early stage and slightly raw but it has received lots of contributions recently, so we might hope for things to improve soon.

Angular universal

Angular universal is a seed project that can be used for creating projects with support for server-side rendering.

@ngrx/store

@ngrx/store is a state management library for Angular inspired by Redux, being based on state mutated by pure reducers. Its integration with RxJS allows you to utilize the push change detection strategy for better performance.

There are plenty of other libraries and tools available in the Awesome Angular list.

React

Create react app

Create-react-app is a CLI utility for React to quickly set up new projects. Similar to Angular CLI it allows you to generate a new project, start a development server and create a bundle. It uses Jest, a relatively new test runner from Facebook, for unit testing, which has some nice features of its own. It also supports flexible

application profiling using environment variables, backend proxies for local development, Flow, and other features. Check out this brief <u>introduction to create-react-app</u> for more information.

React Native

<u>React Native</u> is a platform developed by Facebook for creating native mobile applications using React. Unlike Ionic, which produces a hybrid application, React Native produces a truly native UI. It provides a set of standard React components which are bound to their native counterparts. It also allows you to create your own components and bind them to native code written in Objective-C, Java or Swift.

Material UI

There's a <u>material design component</u> library available for React as well. Compared to Angular's version, this one is more mature and has a wider range of components available.

Next.js

<u>Next.js</u> is a framework for the server-side rendering of React applications. It provides a flexible way to completely or partially render your application on the server, return the result to the client and continue in the browser. It tries to make the complex task of creating universal applications as simple as possible so the set up is designed to be as simple as possible with a minimal amount of new primitives and requirements for the structure of your project.

MobX

<u>MobX</u> is an alternative library for managing the state of an application. Instead of keeping the state in a single immutable store, like Redux does, it encourages you to store only the minimal required state and derive the rest from it. It provides a set of decorators to define observables and observers and introduce reactive logic to your state.

[**Further reading**: <u>How to Manage Your JavaScript Application State with MobX</u>]

Storybook

Storybook is a component development environment for React. It allows you to quickly set up a separate application to showcase your components. On top of that, it provides numerous add-ons to document, develop, test and design your components. We've found it to be extremely useful to be able to develop components independently from the rest of the application. You can learn more about Storybook from a previous article.

> There are plenty of other libraries and tools available in the Awesome React list.

Adoption, Learning Curve and Development Experience

An important criterion for choosing a new technology is how easy it is to learn. Of course, the answer depends on a wide range of factors such as your previous experience and a general familiarity with the related concepts and patterns. However, we can still try to assess the number of new things you'll need to learn to get started with a given framework. Now, if we assume that you already know ES6+, build tools and all of that, let's see what else you'll need to understand.

React

With react the first thing you'll encounter is JSX. It does seem awkward to write for some developers, however, it doesn't add that much complexity; Just expressions, which are actually JavaScript, and a special HTML-like syntax. You'll also need to learn how to write components, use props for configuration and manage internal state. You don't need to learn any new logical structures or loops since all of this is plain JavaScript.

The official tutorial is an excellent place to start learning React. Once you're done with that, get familiar with the router. The react router v4 might be slightly complex and unconventional, but nothing to worry about. Using Redux will require a paradigm shift to learn how to accomplish already familiar tasks in a manner suggested by the library. The free Getting Started with Redux video course can quickly introduce you to the core concepts. Depending on the size and

the complexity of your project you'll need to find and learn some additional libraries and this might be the tricky part, but after that everything should be smooth sailing.

We were genuinely surprised at how easy it was to get started using React. Even people with a backend development background and limited experience in frontend development were able to catch up quickly. The error messages you might encounter along the way are usually clear and provide explanations on how to resolve the underlying problem. The hardest part may be finding the right libraries for all of the required capabilities, but structuring and developing an application is remarkably simple.

Angular

Learning Angular will introduce you to more new concepts than React. First of all, you'll need to get comfortable with TypeScript. For developers with experience in statically typed languages such as Java or .NET this might be easier to understand than JavaScript, but for pure JavaScript developers, this might require some effort.

The framework itself is rich in topics to learn, starting from basic ones such as modules, dependency injection, decorators, components, services, pipes, templates, and directives, to more advanced topics such as change detection, zones, AoT compilation, and Rx.js. These are all covered in the documentation. Rx.js is a heavy topic on its own and is described in much detail on the official website. While relatively easy to use on a basic level it gets more complicated when moving on to advanced topics.

All in all, we noticed that the entry barrier for Angular is higher than for React. The sheer number of new concepts is confusing to newcomers. And even after you've started, the experience might be a bit rough since you need to keep in mind things like Rx.js subscription management, change detection performance and bananas in a box (yes, this is an actual advice from the documentation). We often encountered error messages that are too cryptic to understand, so we had to google them and pray for an exact match.

It might seem that we favor React here, and we definitely do. We've had experience onboarding new developers to both Angular and React projects of

comparable size and complexity and somehow with React it always went smoother. But, like I said earlier, this depends on a broad range of factors and might work differently for you.

Putting it Into Context

You might have already noted that each framework has its own set of capabilities, both with their good and bad sides. But this analysis has been done outside of any particular context and thus doesn't provide an answer on which framework should you choose. To decide on that, you'll need to review it from a perspective of your project. This is something you'll need to do on your own.

To get started, try answering these questions about your project and when you do, match the answers against what you've learned about the two frameworks. This list might not be complete, but should be enough to get you started:

1. How big is the project?
2. How long is it going to be maintained for?
3. Is all of the functionality clearly defined in advance or are you expected to be flexible?
4. If all of the features are already defined, what capabilities do you need?
5. Are the domain model and business logic complex?
6. What platforms are you targeting? Web, mobile, desktop?
7. Do you need server-side rendering? Is SEO important?
8. Will you be handling a lot of real-time event streams?
9. How big is your team?
10. How experienced are your developers and what is their background?
11. Are there any ready-made component libraries that you would like to use?

If you're starting a big project and you would like to minimize the risk of making a bad choice, consider creating a proof-of-concept product first. Pick some of the key features of the projects and try to implement them in a simplistic manner using one of the frameworks. PoCs usually don't take a lot if time to build but will give you some valuable personal experience on working with the framework and allow you to validate the key technical requirements. If you're satisfied with the results, you can continue with full-blown development. If not, failing fast will save you lot of headaches in the long run.

One Framework to Rule Them All?

Once you've picked a framework for one project, you'll get tempted to use the exact same tech stack for your upcoming projects. Don't. Even though it's a good idea to keep your tech stack consistent, don't blindly use the same approach every time. Before starting each project, take a moment to answer the same questions once more. Maybe for the next project, the answers will be different or the landscape will change. Also, if you have the luxury of doing a small project with a non-familiar tech stack, go for it. Such experiments will provide you with invaluable experience. Keep your mind open and learn from your mistakes. At some point, a certain technology will just feel natural and *right.*

Chapter **6**

Retrofit Your Website as a Progressive Web App

by Craig Buckler

Peer reviewed by <u>AJ Latour</u>, <u>Panayiotis «pvgr» Velisarakos</u> and <u>Dave Maxwell</u>.

There's been a lot of buzz around Progressive Web Apps (PWAs) lately, with many people questioning whether they represent the future of the (mobile) web. I'm not going to get into the whole native app vs PWA debate, but one thing is for sure — they go a long way to enhancing mobile and improving its user experience. With mobile web access destined to surpass that of all other devices combined by 2018, can you afford to ignore this trend?

The good news is that making a PWA is not hard. In fact, it's quite possible to take an existing website and convert it into a PWA. And that's exactly what I'll be doing in this tutorial — by the time you're finished, you'll have a website that behaves like a native web app. It will work offline and have its own home screen icon.

What Are Progressive Web Apps?

Progressive Web Apps (referred to as *"PWAs"*) are an exciting innovation in web technology. PWAs comprise a mixture of technologies to make a web app function like a native mobile app. The benefits for developers and users overcome the constraints imposed by web-only and native-only solutions:

1. You only need one app developed with open, standard W3C web technologies. There's no need to develop separate native codebases.
2. Users can discover and try your app before installation.
3. There's no need to use an AppStore, abide with arcane rules or pay fees. Application updates occur automatically without user interaction.
4. Users are prompted to "install" which adds an icon to their home screen.
5. When launched, the PWA displays an attractive splash screen.
6. The browser chrome options can be modified if necessary to provide a full-screen experience.
7. Essential files are cached locally so PWAs respond faster than standard web apps *(they can even be faster than native apps)*.
8. Installation is lightweight - perhaps a few hundred KB of cached data.
9. All data exchanges must occur over a secure HTTPS connection.
10. PWAs function offline and can synchronize data when the connection returns.

It's early days, but case studies are positive. Flipkart, India's largest e-commerce site, experienced a 70% increase in sales conversions and trebled on-site time when they abandoned their native app for a PWA. Alibaba, the world's largest business trading platform, experienced a similar conversion rate increase of 76%.

Solid PWA technology support is available in Firefox, Chrome and the other Blink-based browsers. Microsoft is working on an Edge implementation. Apple remains silent although there are promising comments in the WebKit five-year plan. Fortunately, browser support is mostly irrelevant...

Progressive Web Apps are Progressive Enhancements

Your app will still run in browsers which don't support PWA technology. The user won't get the benefits of offline functionality but everything will continue to work as before. Given the cost-to-benefit rewards, there's little reason *not* to add PWA technologies to your system.

It's Not Just Apps

Google has led the PWA movement so most tutorials describe how to build a Chrome-based native-looking mobile app from the ground-up. However, you don't need a special single-page app or have to follow material interface design guidelines. Most websites can be PWA-ized within a few hours. That includes your WordPress or static site. Smashing Magazine announced they were running as a PWA while this article was being written!

Demonstration Code

Demonstration code is available from https://github.com/sitepoint-editors/pwa-retrofit

It provides a simple four-page website with a few images, one stylesheet and a single main JavaScript file. The site works in all modern browsers (IE10+). If the browser supports PWA technologies the user can read previously-viewed pages when they're offline.

To run the code, ensure Node.js is installed then start the provided web server in your terminal with:
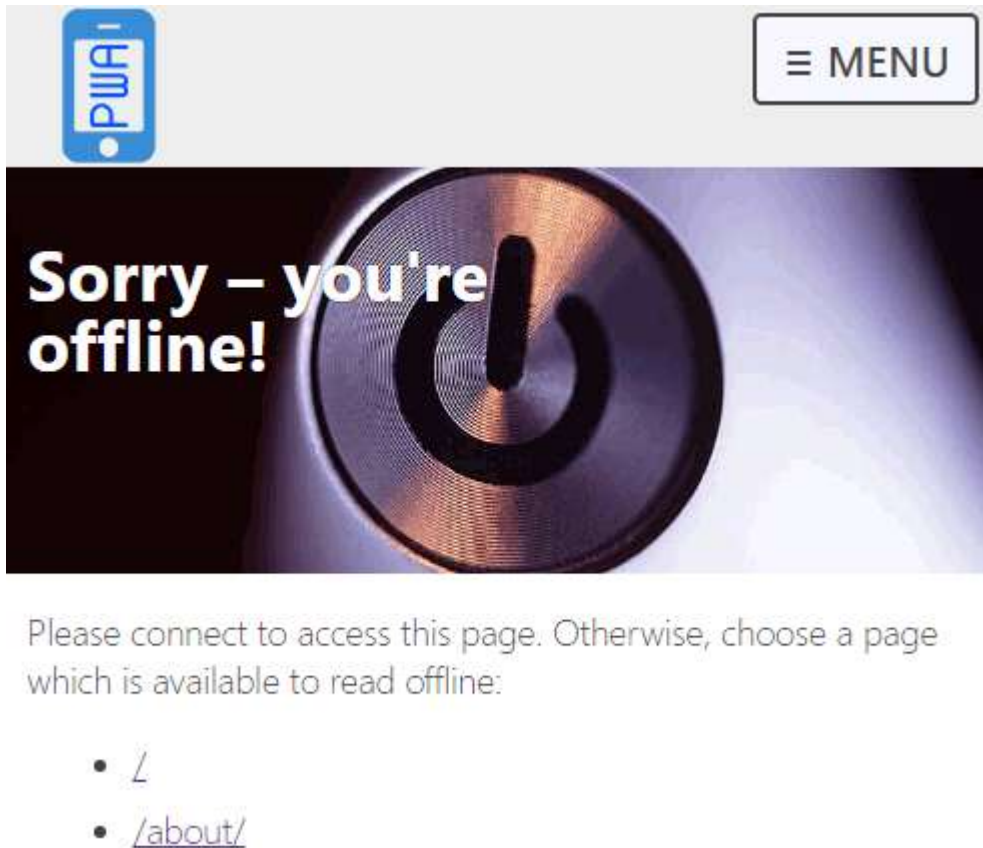
```
node ./server.js [port]
```

where [port] is optional and defaults to 8888. Open Chrome or another Blink-based browser such as Opera or Vivaldi then navigate to http://localhost:8888/ (or whichever port you specified). You can also open the Developer Tools (F12 or Cmd/Ctrl + Shift + I) to view various console messages.

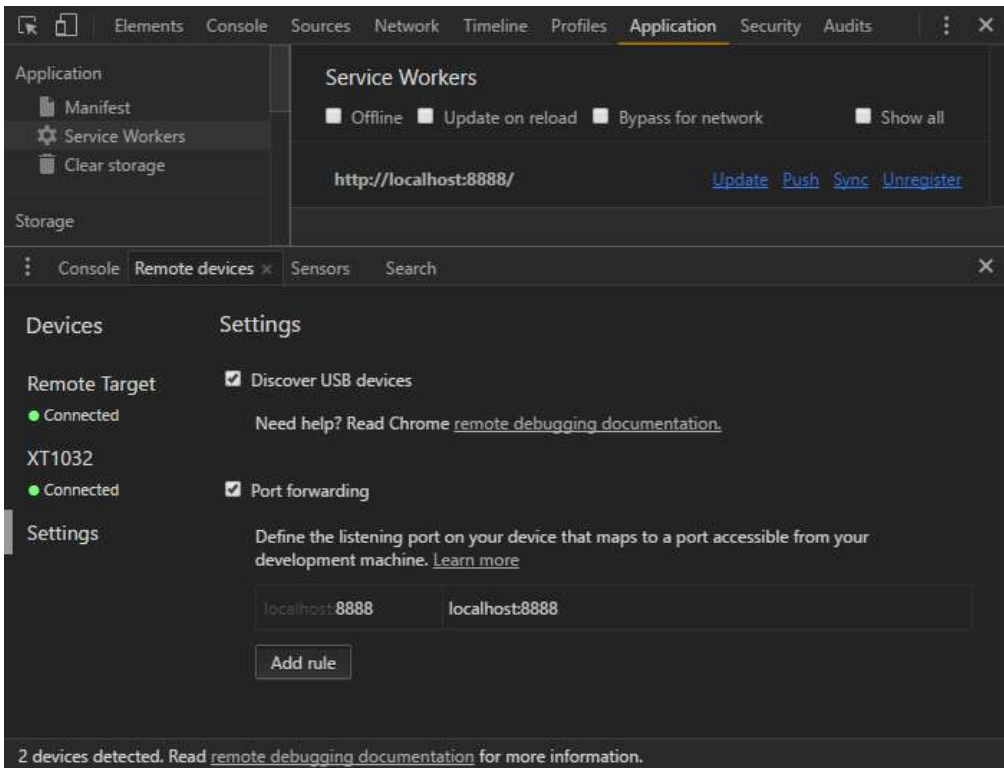View the home page, and perhaps one other, then go offline by either:

1. Stopping the web server with Cmd/Ctrl + C, or
2. Check the **Offline** checkbox in the **Network** or **Application - Service Workers** tab of the Developer Tools.

Revisit any of the pages you viewed earlier and they will still load. Visit a page you've not seen to be presented with a "you're offline" page containing a list of viewable pages:

Please connect to access this page. Otherwise, choose a page which is available to read offline:

- /
- /about/

Connect a Device

You can also view the demonstration page on an Android smartphone connected to your PC/Mac via USB. Open the **Remote devices** panel from **More tools** in the top-left three-dot menu.

Select **Settings** on the left and click **Add Rule** to forward port 8888 to localhost:8888. You can now open Chrome on the smartphone and navigate to http://localhost:8888/.

You can use the browser menu to "Add to Home screen". Make a couple of visits and the browser should prompt you to "install". Both options create a new icon on your home screen. Browse a few pages then close Chrome and disconnect your device. You can then launch the **PWA Website** app - you'll see a splash screen and be able to view pages you read previously despite having no connection to the server.

There are three essential steps to transform your website into a Progressive Web App...

Step 1: Enable HTTPS

PWAs require an HTTPS connection for reasons which will become apparent shortly. Prices and processes will differ across hosts but it's worth the cost and effort, given that Google search is ranking secure sites higher.

HTTPS is not necessary for the demonstration above because Chrome permits the use of localhost or any 127.x.x.x address for testing. You can also test PWA technology on HTTP sites if you launch Chrome with the following command line flags:

- `--user-data-dir`
- `--unsafety-treat-insecure-origin-as-secure`

Step 2: Create a Web App Manifest

The web app manifest provides information about the application such as the name, description, and images which are used by the OS to configure home screen icons, splash pages and the viewport. In essence, the manifest is a single file alternative to the numerous vendor-specific icon and theme meta tags you may already have in your pages.

The manifest is a JSON text file in the root of your app. It must be served with a `Content-Type: application/manifest+json` or `Content-Type: application/json` HTTP header. The file can be called anything but has been named `/manifest.json` in the demonstration code:

```
{
  "name"              : "PWA Website",
  "short_name"        : "PWA",
  "description"       : "An example PWA website",
  "start_url"         : "/",
  "display"           : "standalone",
  "orientation"       : "any",
  "background_color"  : "#ACE",
  "theme_color"       : "#ACE",
  "icons": [
```

```
    {
      "src"             : "/images/logo/logo072.png",
      "sizes"           : "72x72",
      "type"            : "image/png"
    },
    {
      "src"             : "/images/logo/logo152.png",
      "sizes"           : "152x152",
      "type"            : "image/png"
    },
    {
      "src"             : "/images/logo/logo192.png",
      "sizes"           : "192x192",
      "type"            : "image/png"
    },
    {
      "src"             : "/images/logo/logo256.png",
      "sizes"           : "256x256",
      "type"            : "image/png"
    },
    {
      "src"             : "/images/logo/logo512.png",
      "sizes"           : "512x512",
      "type"            : "image/png"
    }
  ]
}
```

A link to this file is required in the `<head>` of all your pages:

```
<link rel="manifest" href="/manifest.json">
```

The main manifest properties are:

- **name** - the full name of the application to be displayed to the user
- **short_name** - the short name for situations where there is insufficient space for the full name

- **description** - a long description of the application
- **start_url** - the relative URL to start the application (typically `/`)
- **scope** - the navigation scope. For example, a scope of `/app/` would restrict the app to that folder
- **background-color** - the background color used for splash screens and browser chrome (if required)
- **theme_color** - the application's colour, typically the same as the background, which can affect how the app is displayed
- **orientation** - the preferred orientation: `any`, `natural`, `landscape`, `landscape-primary`, `landscape-secondary`, `portrait`, `portrait-primary`, and `portrait-secondary`
- **display** - the preferred view: `fullscreen` (no chrome), `standalone` (looks like a native app), `minimal-ui` (a small set of UI controls) and `browser` (a conventional browser tab)
- **icons** - an array of image objects defining the `src` URL, `sizes` and `type`. A range of icons should be defined.

MDN provides a full list of <u>Web App Manifest properties</u>.

The **Manifest** section of Chrome's Development Tools **Application** tab validates your manifest JSON and provides an "Add to homescreen" link which functions on desktop devices:

Step 3: Create a Service Worker

Service Workers are programmable proxies which can intercept and respond to network requests. They are a single JavaScript file which resides in the application root.

Your page JavaScript (`/js/main.js` in the demonstration code) can check for service worker support and register the file:

```
if ('serviceWorker' in navigator) {

  // register service worker
  navigator.serviceWorker.register('/service-worker.js');

}
```

If you don't need offline capabilities, simply create an empty
`/service-worker.js` file - users will be prompted to install your app!

Service workers can be bewildering but you should be able to adapt the
demonstration code for your own purposes. It is a standard web worker script
which the browser downloads (when possible) and runs on a separate thread. It
has no access to the DOM or other page APIs but will intercept network requests
triggered by page changes, asset downloads, and Ajax calls.

This is the primary reason your site requires HTTPS. Imagine the chaos if a third-
party script could inject its own service worker from another domain. It would be
able to examine and modify all data exchanges between the client and server!

Service workers react to three primary events: *install*, *activate* and *fetch*.

Install Event

This occurs when the application is installed. It is typically used to cache
essential files using the Cache API.

First, we'll define some configuration variables for:

1. The cache name (`CACHE`) and version (`version`). Your application can have
   multiple cache stores but we only require one. A version number is applied,
   so if we make significant changes a new cache will be used and all previously
   cached files are ignored.
2. An offline page URL (`offlineURL`). This is a page which will be presented
   when the user is offline and attempts to load a page they have not visited
   before.
3. An array of essential files to install which ensure the site functions offline
   (`installFilesEssential`). This should include assets such as CSS and
   JavaScript but I've also included the home page (`/`) and logo. You should also
   include variations such as `/` and `/index.html` if URLs can be addressed in
   more than one way. Note that `offlineURL` is added to this array.
4. Optionally, an array of desirable files (`installFilesDesirable`). These will be
   downloaded if possible but will not make the installation abort on failure.

```
// configuration
const
  version = '1.0.0',
  CACHE = version + '::PWAsite',
  offlineURL = '/offline/',
  installFilesEssential = [
    '/',
    '/manifest.json',
    '/css/styles.css',
    '/js/main.js',
    '/js/offlinepage.js',
    '/images/logo/logo152.png'
  ].concat(offlineURL),
  installFilesDesirable = [
    '/favicon.ico',
    '/images/logo/logo016.png',
    '/images/hero/power-pv.jpg',
    '/images/hero/power-lo.jpg',
    '/images/hero/power-hi.jpg'
  ];
```

The `installStaticFiles()` function adds files to the cache using the promise-based Cache API. A return value is only generated when the essential files are cached:

```
// install static assets
function installStaticFiles() {

  return caches.open(CACHE)
    .then(cache => {

      // cache desirable files
      cache.addAll(installFilesDesirable);

      // cache essential files
      return cache.addAll(installFilesEssential);

    });
```

```
  }
```

Finally, we add an `install` event listener. The `waitUntil` method ensures the service worker will not install until all enclosed code has executed. It runs `installStaticFiles()` then `self.skipWaiting()` to make the service worker active:

```
// application installation
self.addEventListener('install', event => {

  console.log('service worker: install');

  // cache core files
  event.waitUntil(
    installStaticFiles()
    .then(() => self.skipWaiting())
  );

});
```

Activate Event

This occurs when the service worker is activated either immediately after installation or on return. You may not require this handler but the demonstration code uses one to delete old caches when they exist:

```
// clear old caches
function clearOldCaches() {

  return caches.keys()
    .then(keylist => {

      return Promise.all(
        keylist
```

```
        .filter(key => key !== CACHE)
        .map(key => caches.delete(key))
    );

  });

}

// application activated
self.addEventListener('activate', event => {

  console.log('service worker: activate');

    // delete old caches
  event.waitUntil(
    clearOldCaches()
    .then(() => self.clients.claim())
    );

});
```

Note the final `self.clients.claim()` call sets this service worker as the active worker for the site.

Fetch Event

This occurs whenever a network request is made. It calls the `respondWith()` method to hijack GET requests and return:

1. An asset from the cache.
2. If #1 fails, the asset is loaded from the network using the Fetch API) (unrelated to the service worker fetch event). That asset is then added to the cache.
3. If #1 and #2 fail, an appropriate response is returned.

```
// application fetch network data
self.addEventListener('fetch', event => {
```

```javascript
  // abandon non-GET requests
  if (event.request.method !== 'GET') return;

  let url = event.request.url;

  event.respondWith(

    caches.open(CACHE)
      .then(cache => {

        return cache.match(event.request)
          .then(response => {

            if (response) {
              // return cached file
              console.log('cache fetch: ' + url);
              return response;
            }

            // make network request
            return fetch(event.request)
              .then(newreq => {

                console.log('network fetch: ' + url);
                if (newreq.ok) cache.put(event.request,
newreq.clone());
                return newreq;

              })
              // app is offline
              .catch(() => offlineAsset(url));

          });

      })

  );

});
```

The final call to `offlineAsset(url)` returns an appropriate response using a couple of helper functions:

```
// is image URL?
 let iExt = ['png', 'jpg', 'jpeg', 'gif', 'webp',
↪ 'bmp'].map(f => '.' + f);
function isImage(url) {

 return iExt.reduce((ret, ext) => ret ||
↪ url.endsWith(ext), false);

}


// return offline asset
function offlineAsset(url) {

  if (isImage(url)) {

    // return image
    return new Response(
 '<svg role="img" viewBox="0 0 400 300"
↪ xmlns="http://www.w3.org/2000/svg"><title>offline</title><path
↪ d="M0 0h400v300H0z" fill="#eee" /><text x="200" y="150"
↪ text-anchor="middle" dominant-baseline="middle"
↪ font-family="sans-serif" font-size="50"
↪ fill="#ccc">offline</text></svg>',
      { headers: {
        'Content-Type': 'image/svg+xml',
        'Cache-Control': 'no-store'
      }}
    );

  }
  else {

    // return page
    return caches.match(offlineURL);

  }
```

```
}
```

The `offlineAsset()` function checks whether the request is for an image and returns an SVG containing the text "offline". All other requests return the `offlineURL` page.
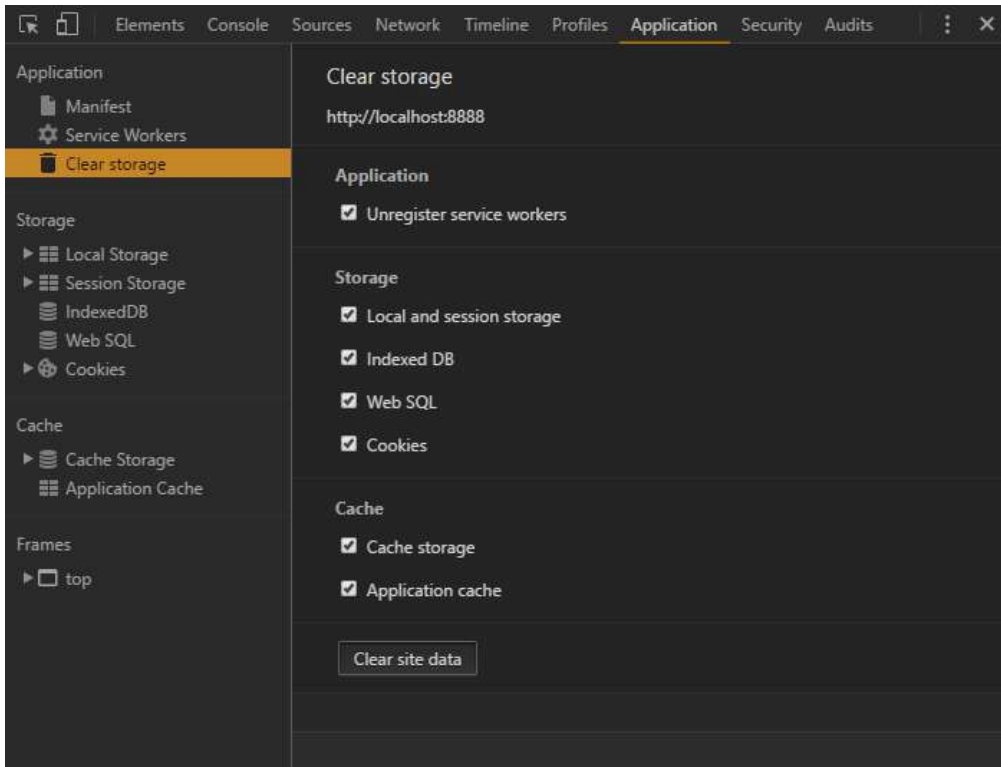
The **Service Worker** section of Chrome's Development Tools **Application** tab provides information about your workers, with errors and facilities to force reload and make the browser go offline:

The **Cache Storage** section lists all caches within the current scope and the cached assets they contain. You may need to click the refresh button at the bottom of the pane if when the cache is updated:

Unsurprisingly, the **Clear storage** section can delete your service worker and caches:

Bonus Step 4: Create a Useful Offline Page

The offline page can be static HTML informing the user that the page they requested is not available offline. However, we can also provide a list of page URLs which are available to read.

The Cache API can be accessed within our `main.js` script. However, the API uses promises which fail in unsupported browsers and will cause all JavaScript to halt execution. To avoid this, we'll add code which checks whether the offline list element and the Caches API is available before loading another `/js/offlinepage.js` JavaScript file (which must be present in the `installFilesEssential` array above):

```
// load script to populate offline page list
 if (document.getElementById('cachedpagelist') &&
↪ 'caches' in window) {
```

```
    var scr = document.createElement('script');
    scr.src = '/js/offlinepage.js';
    scr.async = 1;
    document.head.appendChild(scr);
}
```

`/js/offlinepage.js` locates the most recent cache by version name, gets a list of all URL keys, removes non-page URLs, sorts the list and appends it to the DOM node with the ID `cachedpagelist`:

```
// cache name
const
  CACHE = '::PWAsite',
  offlineURL = '/offline/',
  list = document.getElementById('cachedpagelist');

// fetch all caches
window.caches.keys()
  .then(cacheList => {

    // find caches by and order by most recent
    cacheList = cacheList
      .filter(cName => cName.includes(CACHE))
      .sort((a, b) => a - b);

    // open first cache
    caches.open(cacheList[0])
      .then(cache => {

        // fetch cached pages
        cache.keys()
          .then(reqList => {

            let frag = document.createDocumentFragment();

            reqList
              .map(req => req.url)
  .filter(req => (req.endsWith('/') ||
```

```
↳ req.endsWith('.html')) && !req.endsWith(offlineURL))
              .sort()
              .forEach(req => {
                let
                  li = document.createElement('li'),
                  a = li.appendChild(document.createElement('a'));
                a.setAttribute('href', req);
                a.textContent = a.pathname;
                frag.appendChild(li);
              });

          if (list) list.appendChild(frag);

        });

    })

  });
```

Development Tools

If you think JavaScript debugging is tough, service workers won't be much fun! Chrome's **Application** tab of the Developer Tools provides a solid set of features and logging statements are also output to the console.

You should consider running your app in an **Incognito window** during development since cached files are not retained after you close the tab.

Firefox offers a JavaScript debugger accessed from the **Service Workers** option of the tools menu. Better facilities are promised soon.

Finally, the Lighthouse extension for Chrome also provides useful information about your PWA's implementation.

PWA Gotchas

Progressive Web Apps require new technologies so some caution is advised. That said, they are an enhancement of your existing website which should take no longer than a few hours and have no negative effect on unsupported browsers.

Developer opinions vary but there are several points to consider...

URL Hiding

The demonstration site hides the URL bar which I would not recommend unless you have a single-URL app such as a game. The manifest options `display: minimal-ui` or `display: browser` are possibly best for most sites.

Cache Overload

You could cache every page and asset on your site. That's fine for small sites but would it be practical for those with thousands of pages? No one is likely to be interested in all your content and device storage limits could be exceeded. Even if you only store visited pages and assets like the demonstration, the cache could grow excessively.

Perhaps consider:

- only caching important pages such as the home, contact, and the most recent articles
- not caching images, videos and other large files
- regularly wiping older cached files
- providing a "store this page for offline reading" button so the user can choose what to cache.

Cache Refreshing

The demonstration looks for assets in the cache before loading from the network. That's great when users are offline but means they could be viewing old pages even when they're online.

URLs for assets such as images and videos should never change so long-term caching is rarely a problem. You can ensure they remain cached for at least a year (31,536,000 seconds) with the `Cache-Control` HTTP header:

```
Cache-Control: max-age=31536000
```

Pages, CSS and script files can change more frequently so you could set a shorter expiry of 24 hours and ensure it is validated against the server version when online:

```
Cache-Control: must-revalidate, max-age=86400
```

You could also consider cache-busting techniques to ensure older assets cannot be used, e.g. naming your CSS file `styles-abc123.css` and changing the hash on every release.

Caching can become complex so I'd recommend you read Jake Archibold's Caching best practices & max-age gotchas.

Useful Links

The following resources are useful if you want to know more about Progressive Web Apps:

- PWA.rocks example applications
- Progressive Web Apps
- Your First PWA
- Mozilla Service Worker Cookbook
- MDN Using Service Workers

Chapter **7**

10 Tips to Become a Better Node Developer

by Azat Mardan

I started working with Node full-time in 2012 when I joined Storify. Since then, I have never looked back or felt that I missed Python, Ruby, Java or PHP — languages with which I had worked during my previous decade of web development.

Storify was an interesting job for me, because unlike many other companies, Storify ran (and maybe still does) everything on JavaScript. You see, most companies, especially large ones such as PayPal, Walmart, or Capital One, only use Node for certain parts of their stack. Usually they use it as an API gateway or an orchestration layer. That's great. But for a software engineer, nothing compares with full immersion into a Node environment.

In this post I'll outline ten tips to help you become a better Node developer in 2017. These tips come from me, who saw and learned them in the trenches, as well as people who have written the most popular Node and npm modules. Here's what we'll be covering:

1. <u>Avoid complexity</u> — Organize your code into the smallest chunks possible until they look too small and then make them even smaller.
2. <u>Use asynchronous code</u> — Avoid synchronous code like the plague.
3. <u>Avoid blocking require</u> — Put ALL your require statements at the top of the file because they are synchronous and will block the execution.
4. <u>Know that require is cached</u> — This could be a feature or a bug in your code.
5. <u>Always check for errors</u> — Errors are not footballs. Never throw errors and never skip the error check.
6. <u>Use try...catch only in sync code</u> — `try...catch` is useless for async code, plus V8 can't optimize code in `try...catch` as well as plain code.
7. <u>Return callbacks or use if ... else</u> — Just to be sure, return a callback to prevent execution from continuing.
8. <u>Listen to the error events</u> — Almost all Node classes/objects extend the event emitter (observer pattern) and emit the `error` event. Be sure to listen to that.
9. <u>Know your npm</u> — Install modules with `-S` or `-D` instead of `--save` or `--save-dev`
10. <u>Use exact versions in package.json</u>: npm stupidly adds a caret by default when you use `-S`, so get rid of them manually to lock the versions. Never trust semver in your apps, but do so in open-source modules.
11. *Bonus* — Use different dependencies. Put things your project needs only in development in `devDependencies` and then use `npm i --production`. The more un-required dependencies you have, the greater the risk of vulnerability.

So let's bisect and take a look at each one of them individually. Shall we?

Avoid Complexity

Take a look at some of the modules written by Isaac Z. Schlueter, the creator of npm. For example, <u>use-strict</u> enforces JavaScript strict mode for modules, and it's just *three* lines of code:

```
var module = require('module')
module.wrapper[0] += '"use strict";'
Object.freeze(module.wrap)
```

So why avoid complexity? A famous phrase which originated in the US Navy according to one of the legends proclaims: KEEP IT SIMPLE STUPID (or is it *"Keep it simple, stupid"*?). That's for a reason. The human brain can hold only five to seven items in its working memory at any one time. This is just a fact.

By keeping your code modularized into smaller parts, you and other developers can understand and reason about it better. You can also test it better. Consider this example,

```
app.use(function(req, res, next) {
  if (req.session.admin === true) return next()
  else return next(new Error('Not authorized'))
}, function(req, res, next) {
  req.db = db
  next()
})
```

Or this code:

```
const auth = require('./middleware/auth.js')
const db = require('./middleware/db.js')(db)

app.use(auth, db)
```

I'm sure most of you will prefer the second example, especially when the names are self-explanatory. Of course, when you write the code you might think that you understand how it works. Maybe you even want to show off how smart you are by chaining several methods together in one line. Please, code for the dumber version of you. Code for the you who hasn't looked at this code for six months, or a tried or drunk version of you. If you write code at the peak of your mental

capacity, then it will be harder for you to understand it later, not to even mention your colleagues who are not even familiar with the intricacies of the algorithm. Keeping things simple is especially true for Node which uses the asynchronous way.

And yes, there was the left-pad incident but that only affected projects dependent on the public registry and the replacement was published in 11 minutes. The benefits of going small far outweigh the downsides. Also, npm has changed its unpublish policy, and any serious project should be using a caching strategy or a private registry (as a temporary solution).

Use Asynchronous Code

Synchronous code *does* have a (small) place in Node. It's mostly for writing CLI commands or other scripts not related to web apps. Node developers mostly build web apps, hence they use async code to avoid blocking threads.

For example, this might be okay if we are just building a database script, and not a system to handle parallel/concurrent tasks:

```
let data = fs.readFileSync('./acconts.json')
db.collection('accounts').insert(data, (results))=>{
 fs.writeFileSync('./accountIDs.json', results,
↳ ()=>{process.exit(1)})
})
```

But this would be better when building a web app:

```
app.use('/seed/:name', (req, res) => {
  let data = fs.readFile(`./${req.params.name}.json`, ()=>{
    db.collection(req.params.name).insert(data, (results))=>{
 fs.writeFile(`./${req.params.name}IDs.json`, results,
↳ ()={res.status(201).send()})
    })
  })
```

```
})
```

The difference is whether you are writing concurrent (typically long running) or non-concurrent (short running) systems. As a rule of thumb, always write async code in Node.

Avoid Blocking require

Node has a simple module loading system which uses the CommonJS module format. Its built-in `require` function is an easy way to include modules that exist in separate files. Unlike AMD/requirejs, the Node/CommonJS way of module loading is synchronous. The way `require` works is: *you import what was exported in a module, or a file.*

```
const react = require('react')
```

What most developers don't know is that `require` is cached. So, as long as there are no drastic changes to the resolved filename (and in the case of npm modules there are none), then the code from the module will be executed and loaded into the variable just once (for that process). This is a nice optimization. However, even with caching, you are better off putting your require statements first. Consider this code which only loads the `axios` module on the route which actually uses it. The `/connect` route will be slower than needed because the module import is happening when the request is made:

```
app.post('/connect', (req, res) => {
  const axios = require('axios')
  axios.post('/api/authorize', req.body.auth)
    .then((response)=>res.send(response))
})
```

A better, more performant way is to load the modules before the server is even defined, not in the route:

```
const axios = require('axios')
const express = require('express')
app = express()
app.post('/connect', (req, res) => {
  axios.post('/api/authorize', req.body.auth)
    .then((response)=>res.send(response))
})
```

Know That require Is Cached

I mentioned that `require` is cached in the previous section, but what's interesting is that we can have code *outside* of the `module.exports`. For example,

```
 console.log('I will not be cached and only run once, the
↪ first time')

module.exports = () => {
 console.log('I will be cached and will run every time this
↪ module is invoked')
}
```

Knowing that some code might run only once, you can use this feature to your advantage.

Always Check for Errors

Node is not Java. In Java, you throw errors because most of the time if there's an error you don't want the application to continue. In Java, you can handle *multiple* errors at a higher levels with a single `try...catch`.

Not so with Node. Since Node uses the <u>event loop</u> and executes asynchronously, any errors are separated from the context of any error handler (such as `try...catch`) when they occur. This is useless in Node:

```
try {
  request.get('/accounts', (error, response)=>{
    data = JSON.parse(response)
  })
} catch(error) {
  // Will NOT be called
  console.error(error)
}
```

But `try...catch` still can be used in synchronous Node code. So this is a better refactoring of the previous snippet:

```
request.get('/accounts', (error, response)=>{
  try {
    data = JSON.parse(response)
  } catch(error) {
    // Will be called
    console.error(error)
  }
})
```

If we cannot wrap the `request` call in a `try...catch` block, that leaves us with errors coming from request unhandled. Node developers solve this by providing you with `error` as a callback argument. Thus, you need to always manually handle the `error` in each and every callback. You do so by checking for an error (make sure it's not `null`) and then either displaying the error message to the user or a client and logging it, or passing it back up the call stack by calling the callback with `error` (if you have the callback and another function up the call stack).

```
request.get('/accounts', (error, response)=>{
  if (error) return console.error(error)
  try {
    data = JSON.parse(response)
  } catch(error) {
    console.error(error)
```

```
  }
})
```

A little trick you can use is the okay library. You can apply it like this to avoid manual error check on myriads of nested callbacks (Hello, callback hell).

```
var ok = require('okay')

request.get('/accounts', ok(console.error, (response)=>{
  try {
    data = JSON.parse(response)
  } catch(error) {
    console.error(error)
  }
}))
```

Return Callbacks or Use if ... else

Node is concurrent. So it's a feature which can turn into a bug if you are not careful. To be on the safe side terminate the execution with a return statement:

```
let error = true
if (error) return callback(error)
console.log('I will never run - good.')
```

Avoid some unintended concurrency (and failures) due to mishandled control flow.

```
let error = true
if (error) callback(error)
console.log('I will run. Not good!')
```

Just to be sure, return a callback to prevent execution from continuing.

Listen to the `error` Events

Almost all Node classes/objects extend the event emitter (observer pattern) and emit the `error` event. This is an opportunity for developers to catch those pesky errors and handle them before they wreak havoc.

Make it a good habit to create event listeners for `error` by using `.on()`:

```
var req = http.request(options, (res) => {
  if (('' + res.statusCode).match(/^2\d\d$/)) {
    // Success, process response
  } else if (('' + res.statusCode).match(/^5\d\d$/))
    // Server error, not the same as req error. Req was ok.
  }
})

req.on('error', (error) => {
 // Can't even make a request: general error, e.g.
↪ ECONNRESET, ECONNREFUSED, HPE_INVALID_VERSION
  console.log(error)
})
```

Know Your npm

Many Node and event front-end developers know that there is `--save` (for `npm install`) which will not only install a module but create an entry in `package.json` with the version of the module. Well, there's also `--save-dev`, for `devDependencies` (stuff you don't need in production). But did you know you can just use `-S` and `-D` instead of `--save` and `--save-dev`? Yes, you can.

And while you're in the module installation mode, go ahead and remove those ^ signs which `-S` and `-D` will create for you. They are dangerous because they'll allow `npm install` (or its shortcut `npm i`) to pull the latest minor (second digit in the semantic versioning) version from npm. For example, v6.1.0 to v6.2.0 is a minor release.

npm team believes in <u>semver</u>, but you should not. What I mean is that they put caret ^ because they trust open source developers to not introduce breaking changes in minor releases. No one sane should trust it. Lock your versions. Even better, use <u>shrinkwrap</u>: `npm shrinkwrap` which will create a new file with exact versions of dependencies of dependencies.

Conclusion

This post was part one of two. We've already covered a lot of ground, from working with callbacks and asynchronous code, to checking for errors and locking down dependencies. I hope you've found something new or useful here. If you liked it, be sure to check out part two: <u>10 Node.js Best Practices: Enlightenment from the Node Gurus</u>.

Chapter **8**

An Introduction to Functional JavaScript

by M. David Green

You've heard that JavaScript is a functional language, or at least that it's capable of supporting functional programming. But what is functional programming? And for that matter, if you're going to start comparing programming paradigms in general, how is a functional approach different from the JavaScript that you've always written?

Well, the good news is that JavaScript isn't picky when it comes to paradigms. You can mix your imperative, object-oriented, prototypal, and functional code as you see fit, and still get the job done. But the bad news is what that means for your code. JavaScript can support a wide range of programming styles

simultaneously within the same codebase, so it's up to you to make the right choices for maintainability, readability, and performance.

Functional JavaScript doesn't have to take over an entire project in order to add value. Learning a little about the functional approach can help guide some of the decisions you make as you build your projects, regardless of the way you prefer to structure your code. Learning some functional patterns and techniques can put you well on your way to writing cleaner and more elegant JavaScript regardless of your preferred approach.

Imperative JavaScript

JavaScript first gained popularity as an in-browser language, used primarily for adding simple hover and click effects to elements on a web page. For years, that's most of what people knew about it, and that contributed to the bad reputation JavaScript earned early on.

As developers struggled to match the flexibility of JavaScript against the intricacy of the browser document object model (DOM), actual JavaScript code often looked something like this in the real world:

```
var result;
function getText() {
  var someText = prompt("Give me something to capitalize");
  capWords(someText);
  alert(result.join(" "));
};
function capWords(input) {
  var counter;
  var inputArray = input.split(" ");
  var transformed = "";
  result = [];
 for (counter = 0; counter < inputArray.length; counter++)
 ↪ {
    transformed = [
      inputArray[counter].charAt(0).toUpperCase(),
      inputArray[counter].substring(1)
    ].join("");
```

```
    result.push(transformed);
   }
};
document.getElementById("main_button").onclick = getText;
```

So many things are going on in this little snippet of code. Variables are being defined on the global scope. Values are being passed around and modified by functions. DOM methods are being mixed with native JavaScript. The function names are not very descriptive, and that's due in part to the fact that the whole thing relies on a context that may or may not exist. But if you happened to run this in a browser inside an HTML document that defined a `<button id="main_button">`, you might get prompted for some text to work with, and then see the an `alert` with first letter of each of the words in that text capitalized.

Imperative code like this is written to be read and executed from top to bottom (give or take a little variable hoisting). But there are some improvements we could make to clean it up and make it more readable by taking advantage of JavaScript's object-oriented nature.

Object-Oriented JavaScript

After a few years, developers started to notice the problems with imperative coding in a shared environment like the browser. Global variables from one snippet of JavaScript clobbered global variables set by another. The order in which the code was called affected the results in ways that could be unpredictable, especially given the delays introduced by network connections and rendering times.

Eventually, some better practices emerged to help encapsulate JavaScript code and make it play better with the DOM. An updated variation of the same code above, written to an object-oriented standard, might look something like this:

```
(function() {
  "use strict";
  var SomeText = function(text) {
```

```
    this.text = text;
  };
  SomeText.prototype.capify = function(str) {
    var firstLetter = str.charAt(0);
    var remainder = str.substring(1);
    return [firstLetter.toUpperCase(), remainder].join("");
  };
  SomeText.prototype.capifyWords = function() {
    var result = [];
    var textArray = this.text.split(" ");
 for (var counter = 0; counter < textArray.length;
↪ counter++) {
      result.push(this.capify(textArray[counter]));
    }
    return result.join(" ");
  };


↪ document.getElementById("main_button").addEventListener("click",
↪ function(e) {
    var something = prompt("Give me something to capitalize");
    var newText = new SomeText(something);
    alert(newText.capifyWords());
  });
}());
```

In this object-oriented version, the constructor function simulates a class to model the object we want. Methods live on the new object's prototype to keep memory use low. And all of the code is isolated in an anonymous immediately-invoked function expression so it doesn't litter the global scope. There's even a `"use strict"` directive to take advantage of the latest JavaScript engine, and the old-fashioned `onclick` method has been replaced with a shiny new `addEventListener`, because who uses IE8 or earlier anymore? A script like this would likely be inserted at the end of the `<body>` element on an HTML document, to make sure all the DOM had been loaded before it was processed so the `<button>` it relies on would be available.

But despite all this reconfiguration, there are still many artifacts of the same imperative style that led us here. The methods in the constructor function rely on variables that are scoped to the parent object. There's a looping construct for iterating across all the members of the array of strings. There's a `counter` variable that serves no purpose other than to increment the progress through the `for` loop. And there are methods that produce the side effect of modifying variables that exist outside of their own definitions. All of this makes the code more brittle, less portable, and makes it harder to test the methods outside of this narrow context.

Functional JavaScript

The object-oriented approach is much cleaner and more modular than the imperative approach we started with, but let's see if we can improve it by addressing some of the drawbacks we discussed. It would be great if we could find ways to take advantage of JavaScript's built-in ability to treat functions as first-class objects so that our code could be cleaner, more stable, and easier to repurpose.

```javascript
(function() {
  "use strict";
  var capify = function(str) {
 return [str.charAt(0).toUpperCase(),
↳ str.substring(1)].join("");
  };
  var processWords = function(fn, str) {
    return str.split(" ").map(fn).join(" ");
  };

↳ document.getElementById("main_button").addEventListener("click",
↳ function(e) {
    var something = prompt("Give me something to capitalize");
    alert(processWords(capify, something));
  });
}());
```

Did you notice how much shorter this version is? We're only defining two functions: `capify` and `processWords`. Each of these functions is pure, meaning that they don't rely on the state of the code they're called from. The functions don't create side effects that alter variables outside of themselves. There is one and only one result a function returns for any given set of arguments. Because of these improvements, the new functions are very easy to test, and could be snipped right out of this code and used elsewhere without any modifications.

There might have been one keyword in there that you wouldn't recognize unless you've peeked at some functional code before. We took advantage of the new map method on `Array` to apply a function to each element of the temporary array we created when we split our string. Map is just one of a handful of convenience methods we were given when modern browsers and server-side JavaScript interpreters implemented the ECMAscript 5 standards. Just using `map` here, in place of a `for` loop, eliminated the `counter` variable and helped make our code much cleaner and easier to read.

Start Thinking Functionally

You don't have to abandon everything you know to take advantage of the functional paradigm. You can get started thinking about your JavaScript in a functional way by considering a few questions when you write your next program:

- Are my functions dependent on the context in which they are called, or are they pure and independent?
- Can I write these functions in such a way that I could depend on them always returning the same result for a given input?
- Am I sure that my functions don't modify anything outside of themselves?
- If I wanted to use these functions in another program, would I need to make changes to them?

This introduction barely scratches the surface of functional JavaScript, but I hope it whets your appetite to learn more.

Chapter

9

An Introduction to Chart.js 2.0 — Six Simple Examples

by Jack Rometty

Peer reviewed by <u>Tim Severien</u> and <u>Simon Codrington</u>.

If your website is data-intensive, then you will need to find a way to make that data easy to visualize. Humans, after all, are not wonderful at understanding long lists of raw numbers. That's where charts and graphs come in — they can make complicated statistical relationships obvious and intuitive, as well as more accessibile to non-English speakers. Everyone understands basic charts at the same speed, the same can't be said for paragraphs rife with technical jargon. Using charts when it's beneficial, will make your website easier to understand and visually more appealing.

In this article I'll introduce you to a JavaScript charting library called <u>Chart.js</u>. Using six stylish examples, I'll demonstrate how you can use Chart.js to visualize data on your website, as well as configure it to meet your needs.

Why Chart.js?

I chose Chart.js because it can be learned and leveraged quickly. It's designed with simplicity in mind, yet is extremely customizable. In my experience, charting libraries fall onto a spectrum of complexity, where more complex libraries offer deeper customization, but have steeper learning curves. Chart.js is one of the quickest and easiest libraries to learn that doesn't heavily limit your options. It comes with eight different chart types that will cover almost all of your data visualization needs.

Chart.js is actively maintained to a high standard by the open source community. It recently reached version 2.0, which came with a few fundamental syntax changes to make code more consistent, as well as offer mobile support. In this article, I'm going to use Chart.js 2.0 and it's updated syntax. At the end of this article, after giving you a chance to see how Chart.js 2.0 works, there is a section covering the 1.0 -> 2.0 transition and what to expect when reading old Chart.js examples online.

Installing Chart.js

Again, Chart.js is focused on being easy. Easy to learn, easy to leverage, and easy to install. If you'd like to dive into the actual code, <u>check out the GitHub project</u>.

You only need two things to use Chart.js.

1) The library - for this guide, I recommend using a CDN because it's the easiest way to get up and running fast.

```
 <script
↳ src="https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.1.4/
Chart.min.js"></script>
```

2) A `<canvas>` element, as Chart.js leverages HTML5 canvas.

```
</canvas><canvas id="myChart"></canvas>
```

Alternatively, you can use a package manager to download the library. For more information, see the Getting Started guide.

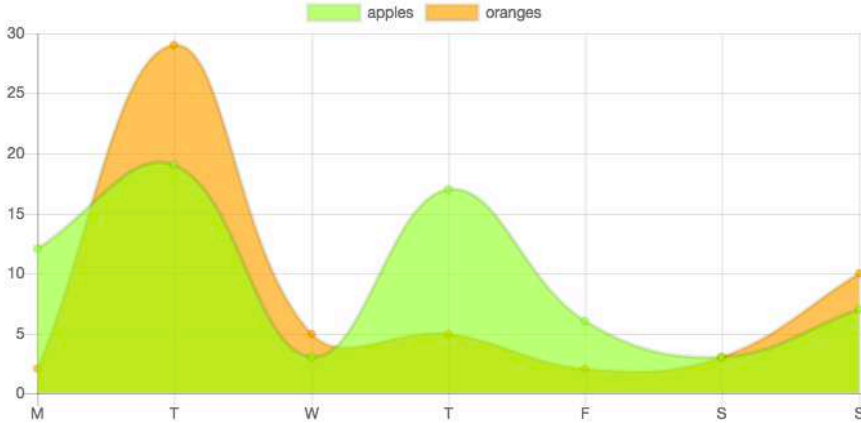Simple, eh? Now without further ado, let's look at what Chart.js has to offer.

Line Chart

This is all you need to create a minimum line chart in Chart.js. Just put it inside of a `<script></script>` somewhere in your `<body>` **after** you declare the HTML5 canvas.

```
 var ctx =
↳ document.getElementById('myChart').getContext('2d');
var myChart = new Chart(ctx, {
  type: 'line',
  data: {
    labels: ['M', 'T', 'W', 'T', 'F', 'S', 'S'],
    datasets: [{
      label: 'apples',
      data: [12, 19, 3, 17, 6, 3, 7],
      backgroundColor: "rgba(153,255,51,0.4)"
    }, {
      label: 'oranges',
      data: [2, 29, 5, 5, 2, 3, 10],
      backgroundColor: "rgba(255,153,0,0.4)"
    }]
  }
});
```

See the Pen 2 - Line chart by SitePoint (@SitePoint) on CodePen.

Chart.js — Line Chart Demo

If this code looks intense, don't worry! All Chart.js examples follow the above format for the most part, so you only have to learn it once. Lets go line by line to understand what's happening.

```
 var ctx =
↳ document.getElementById("myChart").getContext('2d');
```

This line gets a reference to the `<canvas>` element we created earlier, then calls the `getContext` method on it. The `getContext` method returns an object that provides methods and properties for drawing on the canvas. We store this in a variable named `ctx`.

```
var myChart = new Chart(ctx, {
  type: 'line',
  data: // array of line data goes here
});
```

Here we are creating the chart object. I've excluded the data for a moment to focus on the `type` property, which determines the type of chart we want. Chart.js' `new Chart()` constructor takes two parameters:

1. Either a reference to a `</canvas><canvas>` element that the chart will be rendered on, or a reference to its 2d drawing context (here we are using the 2d context). Regardless of which you use, the Chart.js convention is to call it `ctx`.
2. An object literal containing the data and the configuration options that Chart.js will use to build your chart. The required properties are `type` and `data`. In our example `type` is 'line' because we want a line chart. `data` is the data you used to populate the chart.

Chart.js uses array location to determine graph position, so the first point of 'apples' will have the value '12', the second will have '19', and so on. Adding new lines is as easy as adding a new object with a `label` and `data`.

Finally, I have set an rgba background color for each data set to make it more visually appealing.

To learn more about line charts with Chart.js, <u>check out the docs</u>

Pro tip

clicking on any of the legends for the charts ("Apples" and "Oranges" here) will toggle that particular data set. This works for all chart types.

Bar Chart

Bar charts are (mostly) just line charts that look a bit different. By changing one line of our previous example, we can create a bar chart.

```
type: 'line'
```

to:

```
type: 'bar'
```

Yes, it's really that easy.

See the Pen 2. Bar Chart by SitePoint (@SitePoint) on CodePen.

Chart.js — Bar Chart Demo

The full documentation on bar charts can be found here.

Here's the full code for this example:

```javascript
 var ctx =
↪ document.getElementById("myChart").getContext('2d');
var myChart = new Chart(ctx, {
  type: 'bar',
  data: {
    labels: ["M", "T", "W", "R", "F", "S", "S"],
    datasets: [{
      label: 'apples',
      data: [12, 19, 3, 17, 28, 24, 7]
    }, {
      label: 'oranges',
      data: [30, 29, 5, 5, 20, 3, 10]
    }]
  }
});
```

Radar Charts

Radar charts are my favorite type, and again they are in the same family as line and bar charts. Radar charts are just line charts with a radial X axis opposed to a straight line. To get a quick radar chart, change:

```
type: 'bar'
```

to:

```
type: 'radar'
```

Because *that's just how Chart.js rolls.*

Unfortunately, the result is a bit ugly and very hard to read. Bar charts don't have overlap, so solid colors are beneficial. This is not the case with radar charts, which do leverage overlap. We can accommodate this by updating the opactity value of our `backgroundColor` and adding a `borderColor`.

```
{
  label: 'apples',
  backgroundColor: "rgba(179,11,198,.2)",
  borderColor: "rgba(179,11,198,1)",
  data: [12, 19, 3, 17, 6, 3, 7]
}
```

This adds a clearish background and lets us visualize the overlap.

3. Radar Charts by SitePoint (@SitePoint) on CodePen.

Chart.js — Radar Chart Demo

To read more about radar charts, underline check out the docs.

Here's the full code from this example:

```
var ctx = document.getElementById("myChart");
var myChart = new Chart(ctx, {
  type: 'radar',
  data: {
    labels: ["M", "T", "W", "T", "F", "S", "S"],
    datasets: [{
      label: 'apples',
      backgroundColor: "rgba(153,255,51,0.4)",
      borderColor: "rgba(153,255,51,1)",
      data: [12, 19, 3, 17, 28, 24, 7]
    }, {
      label: 'oranges',
      backgroundColor: "rgba(255,153,0,0.4)",
      borderColor: "rgba(255,153,0,1)",
      data: [30, 29, 5, 5, 20, 3, 10]
    }]
  }
```

```
});
```

Polar Charts

Polar charts give each data point an equal amount of radial space. Segments with larger values extend further from the center of the graph. Here's the polar chart for our `apples` data set.

4. Polar Charts by SitePoint (@SitePoint) on CodePen.

Chart.js — Polar Chart Demo (apples)

As usual, specifying that this is a polar chart can be done with a single line. Change:

```
type: 'radar'
```

to:

```
type: 'polarArea'
```

But, the polar area is the first chart I've covered that can't be used to compare two data sets. The previous examples were different ways of contrasting two arrays of equal length, whereas the polar chart (and pie chart, which will be covered next) only visualize a single group of numbers.

Here's the full code for this example:

```
 var ctx =
↪ document.getElementById("myChart").getContext('2d');
var myChart = new Chart(ctx, {
  type: 'polarArea',
  data: {
    labels: ["M", "T", "W", "T", "F", "S", "S"],
    datasets: [{
      backgroundColor: [
        "#2ecc71",
        "#3498db",
        "#95a5a6",
        "#9b59b6",
        "#f1c40f",
        "#e74c3c",
        "#34495e"
      ],
      data: [12, 19, 3, 17, 28, 24, 7]
    }]
  }
});
```

The only new code is a `backgroundColor` array. Each color matches with the `data` element of the same index.

To read more about polar area charts, <u>check out the docs</u>.

Pie & Doughnut Charts
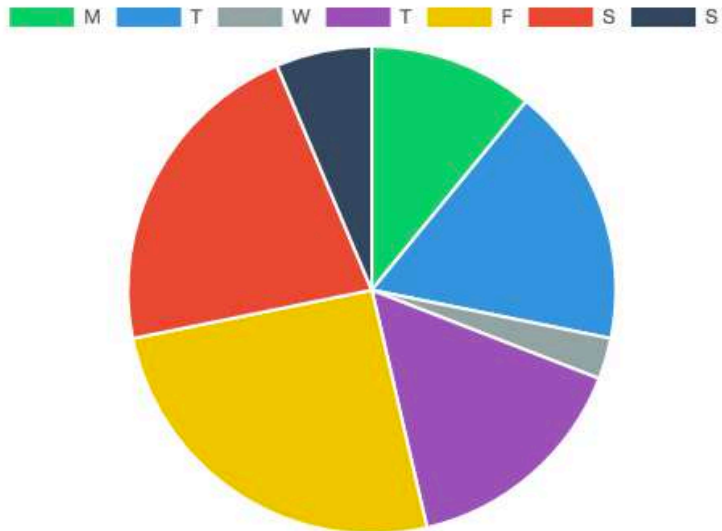
You can probably guess this part by now. Change:

```
type: 'polarArea'
```

to:

```
type: 'pie'
```

The `type` property is the key to Chart.js. Remember how easy it was to transition from a line chart to bar and radar chart? Well, polar, pie, and doughnut charts are equally interchangeable. With that single change, we can alternate from a polar chart to a pie chart.

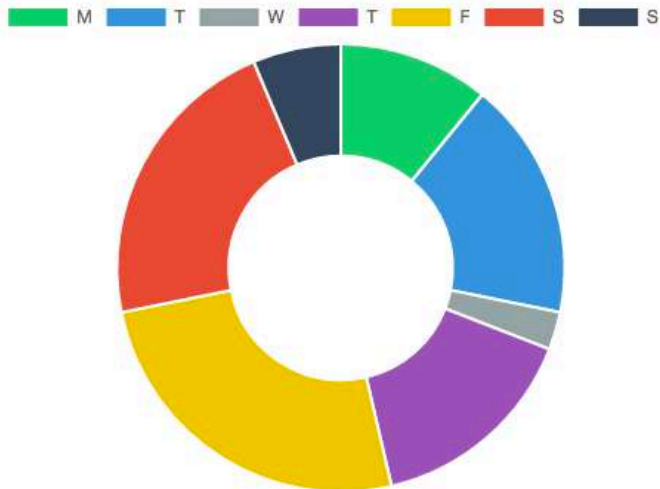5. Pie Chart by SitePoint (@SitePoint) on CodePen.

And for a Doughnut chart:

```
type: 'pie'
```

to:

```
type: 'doughnut'
```

6. Doughnut Chart by SitePoint (@SitePoint) on CodePen.

To read more about pie and doughnut charts, check out the docs.

Here's the full code for the pie chart:

```
 var ctx =
↪ document.getElementById("myChart").getContext('2d');
var myChart = new Chart(ctx, {
  type: 'pie',
  data: {
    labels: ["M", "T", "W", "T", "F", "S", "S"],
```

```
    datasets: [{
      backgroundColor: [
        "#2ecc71",
        "#3498db",
        "#95a5a6",
        "#9b59b6",
        "#f1c40f",
        "#e74c3c",
        "#34495e"
      ],
      data: [12, 19, 3, 17, 28, 24, 7]
    }]
  }
});
```

Doughnut charts have an interesting property called `cutoutPercentage` that dictates how big the center hole is. To dive into that, I first need to show you something about Chart.js I've ignored to help you speed through the basic chart types.

Configuring Chart.js

In every example so far, we've used the format:

```
var myChart = new Chart(ctx, {
  type: //chart type,
  data: // chart data
});
```

But there's a third property called `options`. It fits in right below `data`.

```
var myChart = new Chart(ctx, {
  type: //chart type,
  data: // chart data,
  options: // chart options
```

```
});
```

Now that you're familiar with the fundamentals of Chart.js, it's time to cover some of the tricks availible with `options`.

Titles

It's easy to add a title to any Chart.js chart by adding this set of options. Native titles are awesome, but it's worth noting that they are mostly static and unchanging. This will matter when we try to add custom events in a minute.

```
options: {
  title: {
    display: true,
    text: 'Custom Chart Title'
  }
}
```

The doughnut hole

The `cutoutPercentage` property is a value from 0 to 50. Pie charts are just doughnut charts with a `cutoutPercentage` of 0.

```
options: {
  cutoutPercentage: 10,
}
```

Stacking bar charts

If you would prefer that your bar charts were stacked, just add the following set of options into your bar chart code:

```
options: {
  scales: {
    yAxes: [{
      stacked: true
    }]
  }
}
```

Each chart type has plenty of options for you to dig through. I encourage you to do so.

Handling Events

As mentioned previously clicking on a legend will toggle the data set associated with that particular legend. Let's augment that with our own functionality:

```
var original = Chart.defaults.global.legend.onClick;
 Chart.defaults.global.legend.onClick = function(e,
↪ legendItem) {
  // Insert your custom functionality here

  original.call(this, e, legendItem);
};
```

This code saves a reference to the legend item's `onClick` function into a variable called `original` . It then overwrites this function with our own customized version. The `e` parameter that we are passing to it is a reference to the click event that caused the function to fire and the `legendItem` parameter is a reference to the legend that was clicked on. Once we're done adding our own code, we <u>call</u> the original function specifying a `this` value and passing through the parameters it is expecting. This results in the default action for clicking on a legend (toggling the data set) being carried out.

In other words, We can now package any functionality we want on top of the `onClick()` call as long as we put it above `original.call()`.

A Concrete Example

Let's augment our previous code so that when a user clicks on a legend, the caption at the bottom of the chart updates automatically.

We are only changing the caption, but you can add any functionality you want. For example, a dashboard might have columns of the daily `apples` and `oranges` values. The dashboard could also dynamically update based on the status of your chart with the power of a custom callback. Creating interactive data is easy with Chart.js.

Here's the code

```
var labels = {
  "apples": true,
  "oranges": true
};

var caption = document.getElementById("caption");

var update_caption = function(legend) {
  labels[legend.text] = legend.hidden;

  var selected = Object.keys(labels).filter(function(key) {
    return labels[key];
  });

 var text = selected.length ? selected.join(" & ") :
↪ "nothing";

  caption.innerHTML = "The above chart displays " + text;
};
```

As you can see, we're using an object literal to keep track of the status of the legends. We're also taking advantage of the `legend.text` and `legend.hidden` properties to update its state. The filter function will return any of the object keys whose value is `true` which we use to build our caption.

7. Bar Chart with Custom onClick() by SitePoint (@SitePoint) on CodePen.

Chart.js 2.0 vs 1.0

This article has used Chart.js 2.0 syntax. Chart.js 2.0 is relatively new. The most obvious difference between 2.0 and 1.0 being how to declare charts.

1.0

```
var LineChartDemo = new Chart(ctx).Line(
    //data here,
    //options here
);
```

2.0

```
var myChart = new Chart(ctx, {
    type: 'line',
    data: //data here,
    options: //options here
}
```

Version 1.0 focuses on using function chaining to create a specific type of chart, and then passing in data and options. Version 2.0 switches this up by letting the user create a generic chart object and then pass in type as well as data and options. The second approach matches up more with the philosophy of Chart.js by being as modular and individual as possible. It's worth noting Chart.js 2.0 is backwards compatible and still accepts 1.0 syntax.

Another key feature of Chart.js 2.0 is mobile support. Charts can now scale to fit mobile screens and handle touch events on mobile browsers. With the current proliferation of mobile devices, this is a must-have feature for websites in 2016.

Another feature new to 2.0 that we used in this guide is `title`. Charts now have integrated titles that will cooperate with the chart they're attached to.

The full list of updates can be found <u>in the 2.0.0 release notes</u>.

Conclusion

Chart.js is a perfect match for rapid prototyping of simple charts. There are eight main chart types, of which we have covered: `line`, `bar`, `radar`, `polarArea`, `pie` and `doughnut`. These diverse charts cover most common ways to visualize data, meaning that Chart.js is probably the only graphing library you'll need for your next project.

If you want to learn more about Chart.js, I highly recommend the docs, which you can find <u>on the Chart.js website</u>.

Chapter

10

Learning JavaScript Test-Driven Development by Example

by James Wright

Peer reviewed by <u>Vildan Softic</u>.

You're probably already familiar with automated testing and its benefits. Having a set of tests for your application allows you to make changes to your code with confidence, knowing that the tests have your back should you break anything. It's possible to take things a step further and write your tests *before* you write the code; a practice known as Test-driven development (TDD).

In this tutorial, we will talk about what TDD is and what benefits it brings to you as a developer. We'll use TDD to implement a form validator, which ensures that any values input by the user conform to a specified set of rules.
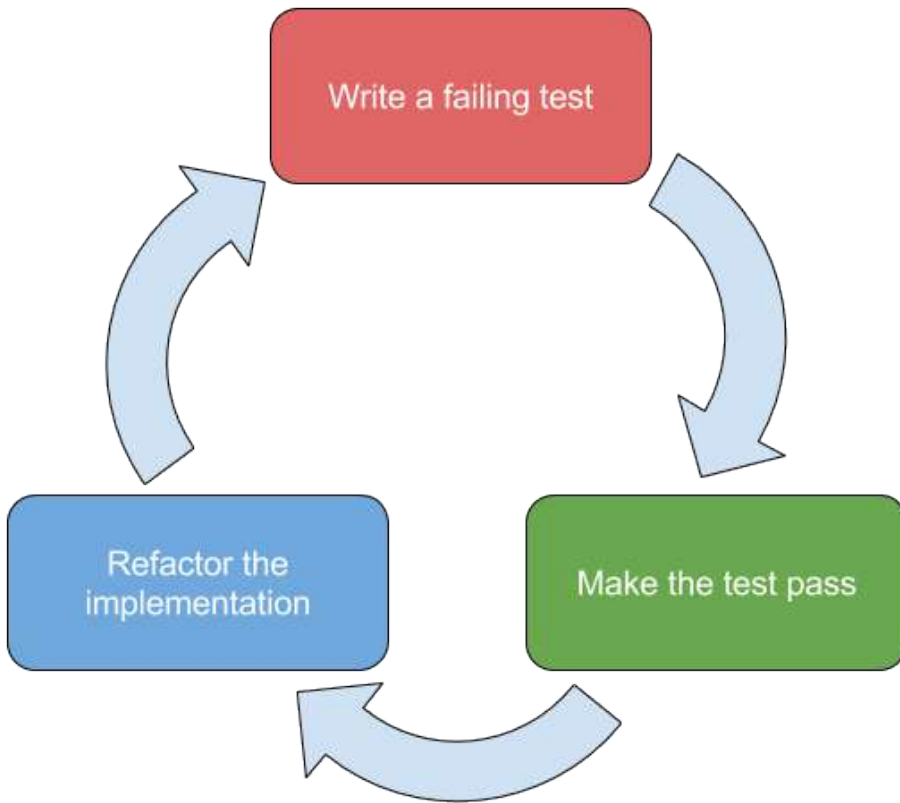
TDD in Node

Note that this article will focus on testing front-end code. If you're looking for something focused on the backend, be sure to check out our course: Test-Driven Development in Node.js

What is TDD?

Test-driven development is a programming methodology with which one can tackle the design, implementation, and testing of units of code, and to some extent the expected functionality of a program.

Complementing the test-first approach of Extreme Programming, in which developers write tests before implementing a feature or a unit, TDD also facilitates the refactoring of code; this is commonly referred to as the *Red-Green-Refactor Cycle.*

/>

- **Write a failing test** - write a test that invokes your logic and assert that the correct behavior is produced

  - In a unit test, this would be asserting the return value of a function or verifying that a mocked dependency was called as expected
  - In a functional test, this would be ensuring that a UI or an API behaves predictably across a number of actions

- **Make the test pass** - implement the minimum amount of code that results in the test passing, and ensure that all other tests continue to pass

- **Refactor the implementation** - update or rewrite the implementation, without breaking any public contracts, to improve its quality without breaking the new and existing tests

I've used TDD to some extent since I was introduced to it at the beginning of my career, but as I have progressed to working on applications and systems with more complex requirements, I have personally found the technique to be time-saving and conducive to the quality and robustness of my work.

Before proceeding, it might be worth familiarizing yourself with some of the various types of automated tests that can be written. Eric Elliot summarises them well:

- **Unit tests** - ensure that individual units of the app, such as functions and classes, work as expected. Assertions test that said units return the expected output for any given inputs
- **Integration tests** - ensure that unit collaborations work as expected. Assertions may test an API, UI, or interactions that may result in side-effects (such as database I/O, logging, etc…)
- **End-to-end tests** - ensure that software works as expected from the user's perspective and that every unit behaves correctly in the overall scope of the system. Assertions primarily test the user interface

Benefits of Test-Driven Development

Immediate test coverage

By writing test cases for a feature before its implementation, code coverage is immediately guaranteed, plus behavioral bugs can be caught earlier in the development lifecycle of a project. This, of course, necessitates tests that cover all behaviors, including error handling, but one should always practice TDD with this mindset.

Refactor with confidence

Referring to the red-green-refactor cycle above, any changes to an implementation can be verified by ensuring that the existing tests continue to pass. Writing tests that run as quickly as possible will shorten this feedback loop; while it's important to cover all possible scenarios, and execution time can vary slightly

between different computers, authoring lean and well-focused tests will save time in the long term.

Design by contract

Test-driven development allows developers to consider how an API will be consumed, and how easy it is to use, without having to worry about the implementation. Invoking a unit in a test case essentially mirrors a call site in production, so the external design can be modified before the implementation stage.

Avoid superfluous code

As long as one is frequently, or even automatically, running tests upon changing the associated implementation, satisfying existing tests reduces the likelihood of unnecessary additional code, arguably resulting in a codebase that's easier to maintain and understand. Consequently, TDD helps one to follow the KISS (Keep it simple, stupid!) principle.

No dependence upon integration

When writing unit tests, if one is conforming to the required inputs, then units will behave as expected once integrated into the codebase. However, integration tests should also be written to ensure that the new code's call site is being invoked correctly.

For example, let's consider the function below, which determines if a user is an admin:

```
'use strict'

function isUserAdmin(id, users) {
    const user = users.find(u => u.id === id);
    return user.isAdmin;
}
```

Rather than hard code the users data, we expect it as a parameter. This allows us to pass a prepopulated array in our test:

```
const testUsers = [
    {
        id: 1,
        isAdmin: true
    },

    {
        id: 2,
        isAdmin: false
    }
];

const isAdmin = isUserAdmin(1, testUsers);
// TODO: assert isAdmin is true
```

This approach allows the unit to be implemented and tested in isolation from the rest of the system. Once there are users in our database, we can integrate the unit and write integration tests to verify that we are correctly passing the parameters to the unit.

Test-Driven Development With JavaScript

With the advent of full-stack software written in JavaScript, a plethora of testing libraries has emerged that allow for the testing of both client-side and server-side code; an example of such a library is Mocha, which we will be using in the exercise.

A good use case for TDD, in my opinion, is form validation; it is a somewhat complex task that typically follows these steps:

1. Read the value from an `<input>` that should be validated
2. Invoke a rule (e.g. alphabetical, numeric) against said value
3. If it is invalid, provide a meaningful error to the user
4. Repeat for the next validatable input

There is a CodePen for this exercise that contains some boilerplate test code, as well as an empty `validateForm` function. **Please fork this before we start.**

Our form validation API will take an instance of `HTMLFormElement` (`<form>`) and validate each input that has a `data-validation` attribute, the possible values of which are:

- `alphabetical` - any case-insensitive combination of the 26 letters of the English alphabet
- `numeric` - any combination of digits between 0 and 9

We will write an end-to-end test to verify the functionality of `validateForm` against real DOM nodes, as well as against the two validation types we'll initially support. Once our first implementation works, we will gradually refactor it by writing smaller units, also following TDD.

Here's the form that our tests will use:

```
<form class="test-form">
 <input name="first-name" type="text"
↪ data-validation="alphabetical" />
 <input name="age" type="text" data-validation="numeric"
↪ />
</form>
```

Between each test, we create a new clone of the form to remove the risk of potential side effects. The `true` parameter passed to `cloneNode` ensures that the form's child nodes are also cloned:

```
let form = document.querySelector('.test-form');

beforeEach(function () {
    form = form.cloneNode(true);
});
```

Writing our first test case

The describe('the validateForm function', function () {}) suite will be used to test our API. Within the inner function, write the first test case, which will ensure that legal values for both the alphabetical and numeric rules will be recognized as valid:

```
 it('should validate a form with all of the possible
↳ validation types', function () {
    const name = form.querySelector('input[name="first-name"]');
    const age = form.querySelector('input[name="age"]');

    name.value = 'Bob';
    age.value = '42';

    const result = validateForm(form);
    expect(result.isValid).to.be.true;
    expect(result.errors.length).to.equal(0);
});
```

Upon saving the changes to your fork, you should see the test fail:

the form validator
  the validateForm function
    ✖ should validate a form with all of the possible validation types

```
TypeError: Cannot read property 'isValid' of undefined
    at n.<anonymous> (pen.js:78:30)
    at r (https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.min.js:2:8564)
    at r.run (https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.min.js:2:9635)
    at i.runTest (https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.min.js:2:14414)
    at https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.min.js:2:15054
    at r (https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.min.js:2:13879)
    at https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.min.js:2:13855
    at n (https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.min.js:2:13646)
    at https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.min.js:2:13710
    at i (https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.min.js:1:572)
```

Now let's make this test green! Remember that we should endeavor to write the minimum, reasonable (no return true;!) amount of code to satisfy the test, so let's not worry about error reporting for now.

Here's the initial implementation, which iterates over our form's `input` elements
and validates the values of each using regular expressions:

```
function validateForm(form) {
    const result = {
        errors: []
    };

    const inputs = Array.from(form.querySelectorAll('input'));
    let isValid = true;

    for (let input of inputs) {
        if (input.dataset.validation === 'alphabetical') {
            isValid = isValid && /^[a-z]+$/i.test(input.value);
        } else if (input.dataset.validation === 'numeric') {
            isValid = isValid && /^[0-9]+$/.test(input.value);
        }
    }

    result.isValid = isValid;
    return result;
}
```

You should now see that our test passes:

the form validator
the validateForm function
✓ should validate a form with all of the possible validation types

Error handling

Below our first test, let's write another which verifies that the return `result`
object's `error` array contains an `Error` instance with the expected message when
an alphabetical field is invalid:

```
 it('should return an error when a name is invalid', function
↳ () {
    const name = form.querySelector('input[name="first-name"]');
    const age = form.querySelector('input[name="age"]');

    name.value = '!!!';
    age.value = '42';

    const result = validateForm(form);

    expect(result.isValid).to.be.false;
    expect(result.errors[0]).to.be.instanceof(Error);
 expect(result.errors[0].message).to.equal('!!! is not a
↳ valid first-name value');
});
```

Upon saving your CodePen fork, you should see the new failing test case in the output. Let's update our implementation to satisfy both test cases:

```
function validateForm(form) {
    const result = {
        get isValid() {
            return this.errors.length === 0;
        },

        errors: []
    };

    const inputs = Array.from(form.querySelectorAll('input'));

    for (let input of inputs) {
        if (input.dataset.validation === 'alphabetical') {
            let isValid = /^[a-z]+$/i.test(input.value);

            if (!isValid) {
 result.errors.push(new Error(`${input.value} is not a valid
↳ ${input.name} value`));
            }
```

```
        } else if (input.dataset.validation === 'numeric') {
            // TODO: we'll consume this in the next test
            let isValid = /^[0-9]+$/.test(input.value);
        }
    }

    return result;
}
```

Now let's add a test that asserts that numeric validation errors are handled correctly:

```
 it('should return an error when an age is invalid', function
↪ () {
    const name = form.querySelector('input[name="first-name"]');
    const age = form.querySelector('input[name="age"]');

    name.value = 'Greg';
    age.value = 'a';

    const result = validateForm(form);

    expect(result.isValid).to.be.false;
    expect(result.errors[0]).to.be.instanceof(Error);
 expect(result.errors[0].message).to.equal('a is not a valid
↪ age value');
});
```

Once you've witnessed the test fail, update the `validateForm` function:

```
} else if (input.dataset.validation === 'numeric') {
    let isValid = /^[0-9]+$/.test(input.value);

    if (!isValid) {
 result.errors.push(new Error(`${input.value} is not a valid
↪ ${input.name} value`));
```

```
    }
}
```

Finally, let's add a test to ensure that multiple errors are handled:

```
 it('should return multiple errors if more than one field is
↪ invalid', function () {
    const name = form.querySelector('input[name="first-name"]');
    const age = form.querySelector('input[name="age"]');

    name.value = '!!!';
    age.value = 'a';

    const result = validateForm(form);

    expect(result.isValid).to.be.false;
    expect(result.errors[0]).to.be.instanceof(Error);
 expect(result.errors[0].message).to.equal('!!! is not a
↪ valid first-name value');
    expect(result.errors[1]).to.be.instanceof(Error);
 expect(result.errors[1].message).to.equal('a is not a valid
↪ age value');
});
```

Given our error handling implementation for the second and third test, this new case should pass immediately. You can confirm that you've followed the steps correctly by <u>verifying your implementation against mine</u>.

Refactoring Our Validator

Although we have a working function that is covered with tests, it emits a number of code smells:

◾ **Multiple responsibilities**

- We're querying the inner DOM nodes of our input, specifying our ruleset, and computing our overall result in the same function. In terms of the _SOLID principles_, this violates the _Single responsibility principle_
- Additionally, a lack of abstraction results in code that is more difficult for other developers to understand

- **Tight coupling**

  - Our current implementation interweaves the above responsibilities in a way that makes updates to each concern brittle; changes to one detail of our large method will make debugging difficult in the case that we introduce an issue
  - Furthermore, we can't add or alter validation rules without updating the `if` statements. This violates SOLID's _Open/closed principle_

- **Duplication of logic** - if we wish to update the format of our error messages, or push another object to our array, then we must update this in two places

Fortunately, as we've written the functional tests for our validator function, we can make our code better with the confidence that we won't break it.

Let's use TDD to write separate functions for:

1. Mapping our inputs to validation queries
2. Reading our validation rules from an appropriate data structure

The `createValidationQueries` function

By mapping our `NodeList` of `HTMLInputElement`s to objects representing the name of a form field, the type against which it should be validated, and the value of said field, not only will we decouple `validateForm` function from the DOM, but we'll facilitate validation rule lookup when we replace our hard-coded regular expressions.

For example, the validation query object for the `first-name` field would be:

```
{
    name: 'first-name',
```

```
    type: 'alphabetical',
    value: 'Bob'
}
```

Above the `validateForm` function, create an empty function called
`createValidationQueries`. Then, **outside of the** `describe` **suite for**
`validateForm`, create another `describe suite` named 'the
createValidationQueries function'.

It should include the single test case:

```
 describe('the createValidationQueries function', function ()
↳ {
    it(
 'should map input elements with a data-validation attribute
↳ to an array of validation objects',

        function () {
            const name =
form.querySelector('input[name="first-name"]');
            const age = form.querySelector('input[name="age"]');

            name.value = 'Bob';
            age.value = '42';

            const validations = createValidationQueries([name,
age]);

            expect(validations.length).to.equal(2);

            expect(validations[0].name).to.equal('first-name');
            expect(validations[0].type).to.equal('alphabetical');
            expect(validations[0].value).to.equal('Bob');

            expect(validations[1].name).to.equal('age');
            expect(validations[1].type).to.equal('numeric');
            expect(validations[1].value).to.equal('42');
        }
```

```
    );
});
```

Once you've witnessed this fail, write the code for the implementation:

```javascript
function createValidationQueries(inputs) {
    return Array.from(inputs).map(input => ({
        name: input.name,
        type: input.dataset.validation,
        value: input.value
    }));
}
```

When this passes, update `validateForm`'s `for` loop to call our new function and to use the query objects to determine the validity of our form:

```javascript
 for (let validation of
↪ createValidationQueries(form.querySelectorAll('input'))) {
    if (validation.type === 'alphabetical') {
        let isValid = /^[a-z]+$/i.test(validation.value);

        if (!isValid) {
 result.errors.push(new Error(`${validation.value} is not a
↪ valid ${validation.name} value`));
        }
    } else if (validation.type === 'numeric') {
        let isValid = /^[0-9]+$/.test(validation.value);

        if (!isValid) {
 result.errors.push(new Error(`${validation.value} is not a
↪ valid ${validation.name} value`));
        }
    }
}
```

If both our new test and the existing tests pass, <u>as demonstrated in this pen</u>, then
we can make a bigger change; decoupling the validation rules.

The `validateItem` function

To remove our hard-coded rules, let's write a function that takes our rules as a <u>Map</u>
and asserts the validity of our inputs.

Like `createValidationQueries`, we'll write a new test suite before our
implementation. Above the implementation of `validateForm`, write an empty
function called `validateItem`. Then in our main `describe` suite, write another
`describe` suite for our new addition:

```javascript
describe('the validateItem function', function () {
    const validationRules = new Map([
        ['alphabetical', /^[a-z]+$/i]
    ]);

    it(
 'should return true when the passed item is deemed valid
↪ against the supplied validation rules',

        function () {
            const validation = {
                type: 'alphabetical',
                value: 'Bob'
            };

            const isValid = validateItem(validation,
validationRules);
            expect(isValid).to.be.true;
        }
    );
});
```

We're explicitly passing a `Map` of rules to our implementation from the test as we
want to verify its behavior independently of our main function; this makes it a
unit test. Here's our first implementation of `validateItem()`:

```
function validateItem(validation, validationRules) {
 return
↪ validationRules.get(validation.type).test(validation.value);
}
```

Once this test has passed, write a second test case to verify that our function returns `false` when a validation query is invalid; this should pass due to our current implementation:

```
it(
 'should return false when the passed item is deemed
↪ invalid',

    function () {
        const validation = {
            type: 'alphabetical',
            value: '42'
        };

        const isValid = validateItem(validation, validationRules);
        expect(isValid).to.be.false;
    }
);
```

Finally, write a test case to determine that `validateItem` returns `false` when the validation type is not found:

```
it(
 'should return false when the specified validation type is
↪ not found',

    function () {
        const validation = {
            type: 'foo',
            value: '42'
        };
```

```
        const isValid = validateItem(validation, validationRules);
        expect(isValid).to.be.false;
    }
);
```

Our implementation should check if the specified validation type exists in the
`validationRules` Map before testing any values against their corresponding
regular expressions:

```
function validateItem(validation, validationRules) {
    if (!validationRules.has(validation.type)) {
        return false;
    }

 return
↪ validationRules.get(validation.type).test(validation.value);
}
```

Once we see this test passing, let's create a new `Map` above
`createValidationQueries`, which will contain the actual validation rules used
by our API:

```
const validationRules = new Map([
    ['alphabetical', /^[a-z]+$/i],
    ['numeric', /^[0-9]+$/]
]);
```

Finally, let's refactor the `validateForm` function to use the new function and
rules:

```
function validateForm(form) {
    const result = {
        get isValid() {
```

```
            return this.errors.length === 0;
        },

        errors: []
    };

 for (let validation of
↪ createValidationQueries(form.querySelectorAll('input'))) {
        let isValid = validateItem(validation, validationRules);

        if (!isValid) {
            result.errors.push(
 new Error(`${validation.value} is not a valid
↪ ${validation.name} value`)
            );
        }
    }

    return result;
}
```

Hopefully, you'll see that all of the tests pass. Congratulations on using test-driven development to refactor and improve the quality of our code! Your final implementation should resemble this Pen.

Wrapping Up

By following TDD, we have been able to take the initial implementation of our form validation and separate it into independent and understandable parts. I hope you've enjoyed this tutorial and take this practice forward with you into your everyday work.