Development

> *This guide extends on code examples found in the Output Management guide.*

If you've been following the guides, you should have a solid understanding of some of the webpack basics. Before we continue, let's look into setting up a development environment to make our lives a little easier.

> *The tools in this guide are **only meant for development**, please **avoid** using them in production!*

Let's start by setting `mode` to `'development'`.

webpack.config.js

```
  const path = require('path');
  const HtmlWebpackPlugin = require('html-webpack-plugin');
  const { CleanWebpackPlugin } = require('clean-webpack-plugin');

  module.exports = {
+   mode: 'development',
    entry: {
      app: './src/index.js',
      print: './src/print.js',
    },
    plugins: [
      // new CleanWebpackPlugin(['dist/*']) for < v2 versions of CleanWebpackPlugin
      new CleanWebpackPlugin(),
      new HtmlWebpackPlugin({
        title: 'Development',
      }),
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist'),
    },
  };
```

Using source maps

When webpack bundles your source code, it can become difficult to track down errors and warnings to their original location. For example, if you bundle three source files (`a.js` , `b.js` , and `c.js`) into one bundle (`bundle.js`) and one of the source files contains an error, the stack trace will simply point to `bundle.js` . This isn't always helpful as you probably want to know exactly which source file the error came from.

In order to make it easier to track down errors and warnings, JavaScript offers source maps, which map your compiled code back to your original source code. If an error originates from `b.js` , the source map will tell you exactly that.

There are a lot of different options available when it comes to source maps. Be sure to check them out so you can configure them to your needs.

For this guide, let's use the `inline-source-map` option, which is good for illustrative purposes (though not for production):

webpack.config.js

```
  const path = require('path');
  const HtmlWebpackPlugin = require('html-webpack-plugin');
  const { CleanWebpackPlugin } = require('clean-webpack-plugin');

  module.exports = {
    mode: 'development',
    entry: {
      app: './src/index.js',
      print: './src/print.js',
    },
+   devtool: 'inline-source-map',
    plugins: [
      new CleanWebpackPlugin(),
      new HtmlWebpackPlugin({
        title: 'Development',
      }),
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist'),
    },
  };
```

Now let's make sure we have something to debug, so let's create an error in our `print.js` file:

src/print.js

```
  export default function printMe() {
-   console.log('I get called from print.js!');
+   cosnole.log('I get called from print.js!');
  }
```

Run an `npm run build`, it should compile to something like this:

```
...
          Asset       Size  Chunks                    Chunk Names
  app.bundle.js    1.44 MB    0, 1  [emitted]  [big]  app
print.bundle.js    6.43 kB       1  [emitted]         print
     index.html  248 bytes          [emitted]
...
```

Now open the resulting `index.html` file in your browser. Click the button and look in your console where the error is displayed. The error should say something like this:

```
Uncaught ReferenceError: cosnole is not defined
   at HTMLButtonElement.printMe (print.js:2)
```

We can see that the error also contains a reference to the file (`print.js`) and line number (2) where the error occurred. This is great because now we know exactly where to look in order to fix the issue.

Choosing a Development Tool

> *Some text editors have a "safe write" function that might interfere with some of the following tools. Read Adjusting Your Text Editor for a solution to these issues.*

It quickly becomes a hassle to manually run `npm run build` every time you want to compile your code.

There are a couple of different options available in webpack that help you automatically compile your code whenever it changes:

1. webpack's Watch Mode
2. webpack-dev-server
3. webpack-dev-middleware

In most cases, you probably would want to use `webpack-dev-server`, but let's explore all of the above options.

Using Watch Mode

You can instruct webpack to "watch" all files within your dependency graph for changes. If one of these files is updated, the code will be recompiled so you don't have to run the full build manually.

Let's add an npm script that will start webpack's Watch Mode:

package.json

```
  {
    "name": "webpack-demo",
    "version": "1.0.0",
    "description": "",
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1",
+     "watch": "webpack --watch",
      "build": "webpack"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "devDependencies": {
      "clean-webpack-plugin": "^2.0.0",
      "css-loader": "^0.28.4",
      "csv-loader": "^2.1.1",
      "file-loader": "^0.11.2",
      "html-webpack-plugin": "^2.29.0",
      "style-loader": "^0.18.2",
      "webpack": "^4.30.0",
      "xml-loader": "^1.2.1"
    }
  }
```

Now run `npm run watch` from the command line and see how webpack compiles your code. You can see that it doesn't exit the command line because the script is currently watching your files.

Now, while webpack is watching your files, let's remove the error we introduced earlier:

src/print.js

```
  export default function printMe() {
-   cosnole.log('I get called from print.js!');
+   console.log('I get called from print.js!');
  }
```

Now save your file and check the terminal window. You should see that webpack automatically recompiles the changed module!

The only downside is that you have to refresh your browser in order to see the changes. It would be much nicer if that would happen automatically as well, so let's try `webpack-dev-server` which will do exactly that.

Using webpack-dev-server

The `webpack-dev-server` provides you with a simple web server and the ability to use live reloading. Let's set it up:

```
npm install --save-dev webpack-dev-server
```

Change your config file to tell the dev server where to look for files:

webpack.config.js

```
  const path = require('path');
  const HtmlWebpackPlugin = require('html-webpack-plugin');
  const { CleanWebpackPlugin } = require('clean-webpack-plugin');

  module.exports = {
    mode: 'development',
    entry: {
      app: './src/index.js',
      print: './src/print.js',
    },
    devtool: 'inline-source-map',
+   devServer: {
+     contentBase: './dist',
+   },
    plugins: [
      // new CleanWebpackPlugin(['dist/*']) for < v2 versions of CleanWebpackPlugin
      new CleanWebpackPlugin(),
      new HtmlWebpackPlugin({
        title: 'Development',
      }),
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist'),
    },
  };
```

This tells `webpack-dev-server` to serve the files from the `dist` directory on `localhost:8080`.

> *webpack-dev-server doesn't write any output files after compiling. Instead, it keeps bundle files in*

memory and serves them as if they were real files mounted at the server's root path. If your page expects to find the bundle files on a different path, you can change this with the `publicPath` option in the dev server's configuration.

Let's add a script to easily run the dev server as well:

package.json

```
  {
    "name": "development",
    "version": "1.0.0",
    "description": "",
    "private": true,
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1",
      "watch": "webpack --watch",
+     "start": "webpack-dev-server --open",
      "build": "webpack"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "devDependencies": {
      "clean-webpack-plugin": "^2.0.0",
      "css-loader": "^0.28.4",
      "csv-loader": "^2.1.1",
      "express": "^4.15.3",
      "file-loader": "^0.11.2",
      "html-webpack-plugin": "^2.29.0",
      "style-loader": "^0.18.2",
      "webpack": "^4.30.0",
      "webpack-dev-server": "^3.8.0",
      "xml-loader": "^1.2.1"
    }
  }
```

Now we can run `npm start` from the command line and we will see our browser automatically loading up our page. If you now change any of the source files and save them, the web server will automatically reload after the code has been compiled. Give it a try!

The `webpack-dev-server` comes with many configurable options. Head over to the documentation to learn more.

Now that your server is working, you might want to give Hot Module Replacement a try!

Using webpack-dev-middleware

`webpack-dev-middleware` is a wrapper that will emit files processed by webpack to a server. This is used in `webpack-dev-server` internally, however it's available as a separate package to allow more custom setups if desired. We'll take a look at an example that combines `webpack-dev-middleware` with an express server.

Let's install `express` and `webpack-dev-middleware` so we can get started:

```
npm install --save-dev express webpack-dev-middleware
```

Now we need to make some adjustments to our webpack configuration file in order to make sure the middleware will function correctly:

webpack.config.js

```
  const path = require('path');
  const HtmlWebpackPlugin = require('html-webpack-plugin');
  const { CleanWebpackPlugin } = require('clean-webpack-plugin');

  module.exports = {
    mode: 'development',
    entry: {
      app: './src/index.js',
      print: './src/print.js',
    },
    devtool: 'inline-source-map',
    devServer: {
      contentBase: './dist',
    },
    plugins: [
      new CleanWebpackPlugin(),
      new HtmlWebpackPlugin({
        title: 'Output Management',
      }),
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist'),
+     publicPath: '/',
    },
  };
```

The `publicPath` will be used within our server script as well in order to make sure files are served correctly on `http://localhost:3000`. We'll specify the port number later. The next step is setting up our custom `express` server:

project

```
  webpack-demo
  |- package.json
  |- webpack.config.js
+ |- server.js
  |- /dist
  |- /src
    |- index.js
    |- print.js
  |- /node_modules
```

server.js

```
const express = require('express');
const webpack = require('webpack');
const webpackDevMiddleware = require('webpack-dev-middleware');

const app = express();
const config = require('./webpack.config.js');
const compiler = webpack(config);

// Tell express to use the webpack-dev-middleware and use the webpack.config.js
// configuration file as a base.
app.use(webpackDevMiddleware(compiler, {
  publicPath: config.output.publicPath,
}));

// Serve the files on port 3000.
app.listen(3000, function () {
  console.log('Example app listening on port 3000!\n');
});
```

Now add an npm script to make it a little easier to run the server:

package.json

```
  {
    "name": "development",
    "version": "1.0.0",
    "description": "",
    "private": true,
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1",
      "watch": "webpack --watch",
      "start": "webpack-dev-server --open",
+     "server": "node server.js",
      "build": "webpack"
    },
```

```
    "keywords": [],
    "author": "",
    "license": "ISC",
    "devDependencies": {
      "clean-webpack-plugin": "^2.0.0",
      "css-loader": "^0.28.4",
      "csv-loader": "^2.1.1",
      "express": "^4.15.3",
      "file-loader": "^0.11.2",

      "html-webpack-plugin": "^2.29.0",
      "style-loader": "^0.18.2",
      "webpack": "^4.30.0",
      "webpack-dev-middleware": "^1.12.0",
      "webpack-dev-server": "^3.8.0",
      "xml-loader": "^1.2.1"
    }
  }
```

Now in your terminal run `npm run server`, it should give you an output similar to this:

```
Example app listening on port 3000!
...
          Asset       Size  Chunks                    Chunk Names
  app.bundle.js    1.44 MB    0, 1  [emitted]  [big]  app
print.bundle.js    6.57 kB       1  [emitted]         print
     index.html  306 bytes          [emitted]
...
webpack: Compiled successfully.
```

Now fire up your browser and go to `http://localhost:3000`. You should see your webpack app running and functioning!

> *If you would like to know more about how Hot Module Replacement works, we recommend you read the Hot Module Replacement guide.*

Adjusting Your Text Editor

When using automatic compilation of your code, you could run into issues when saving your files. Some editors have a "safe write" feature that can potentially interfere with recompilation.

To disable this feature in some common editors, see the list below:

- **Sublime Text 3**: Add `atomic_save: 'false'` to your user preferences.
- **JetBrains IDEs (e.g. WebStorm)**: Uncheck "Use safe write" in `Preferences > Appearance &`

`Behavior > System Settings`.

- **Vim**: Add `:set backupcopy=yes` to your settings.

Conclusion

Now that you've learned how to automatically compile your code and run a simple development server, you can check out the next guide, which will cover Hot Module Replacement.