VUE.JS ENTERPRISE CRASH COURSE - PDF

# LESSON 2

# FULL-STACK SCAFFOLDING WITH NODE & VUE CLI

VUE.JS DEVELOPERS

# Lesson 2

## Full-stack scaffold with Vue + Node

Welcome to the second lesson of this crash course on enterprise web development with Vue.js.

If you're creating any sort of app with Vue, you'll likely want to utilize the best-practice scaffolding provided by Vue CLI.

But if you're including a backend, it's not entirely clear how, or even if, it should fit into the Vue CLI scaffold, since Vue CLI was not designed to solve this problem.

In this lesson, I'm going to show you one approach to making a full-stack Vue and Node project that I've found to work well.

In this approach we'll be using Vue as a single-page app and the backend will essentially be an API. Even if you don't intend to use this structure, hopefully you can use this lesson as inspiration for your own full-stack enterprise projects.

## Project structure

```
server/
  routes/
  index.js

src/
  components/
  main.js

package.json
```

In this approach, we're going to create a regular Vue CLI project.

### Server source files

We'll then add our server source files in a sub-folder which you can call `server` or `api` or whatever you want.

### package.json

Note that we use a single package.json file and so we'll be sharing modules and scripts between the two aspects of the app.

## Server

*server/index.js*

```
const express = require("express");
const app = express();

const mongoose = require("mongoose");
mongoose.connect("mongodb://localhost:27017/full-stack-vue");

app.use("/items", require("./routes/items"));

app.listen(8070, () => {
  console.log("Listening on port 8070");
});
```

Here's the basic structure of a server which, in this example, is an Express app with Mongo DB database.

I'll briefly mention the key features.

### Express instance

Firstly, we create an Express instance and get it to listen for incoming requests from the Vue client app.

### MongoDB

We then create a connection with the MongoDB database.

### Routes

We can then include a routes module. This defines all the API endpoints and how the server should respond to requests.

## Dev servers

*package.json*

```
{
  ...
  "scripts": {
    ...
    "serve": "vue-cli-service serve",
    ...
  },
  ...
}
```

By default, a Vue CLI installation will include a `serve` script in the *package.json* file. This calls the Vue CLI 3 service binary, and boots a pre-configured Webpack dev server that serves the Vue client app.

But in this project, we're also going to need a dev server for the backend, too.

If the backend requires a build step, for example if you write it in TypeScript, you can make this a Webpack dev server.

If not, you can use a package like Nodemon, which is a simple dev server that you can run from the command line and will serve your app and also watch your files for changes.

## Dev servers

*package.json*

```
{
  ...
  "scripts": {
    ...
    "serve:client": "vue-cli-service serve",
    "serve:server": "nodemon server",
    "serve": "concurrently \"npm:serve:*\" -k",
    ...
  },
  ...
}
```

So how can we run two dev servers easily?

### Serve commands

You'll see here I've added to the NPM scripts so we have both a client serve command and also a server serve command.

It'd be nice to run these simultaneously so we had one command and didn't need two separate terminal instances.

We can do this by using the *concurrently* package.

### Concurrently

So I've created a new `serve` script which calls concurrently. The arguments to concurrently are the commands we want to run.

I've just included one call to `npm:serve` followed by a wildcard. This has the effect of calling both serve scripts.

I've also add the `-k` switch which will kill both processes if either one dies.

## Dev servers

*Terminal*

```
$ npm run serve

## Listening on ports 8080 and 8070
```

Now, when we call npm run serve, both dev servers will run.

Note, though, that these will have to run on separate ports. By default the Vue CLI dev server runs on localhost:8080, so you might put your API dev server on localhost:8070 or whatever you like.

## Requests from the client

*MyComponent.vue*

```
...
export default {
  ...
  created () {
    const response = await axios.get("/api/items");
  }
  ...
}
```

```
// 404: http://localhost:8080/api/items does not exist
```

So onece we have a backend API, our frontend app can call it using an HTTP client like Axios.

For example, we might make a GET request to the items endpoint to fetch all the items and display them.

### 404

But, as it is, we'd get a 404 error because by default the frontend will send all requests to the same host as it is running on.

But as I just mentioned, our frontend and backend dev servers are running on different ports.

What we'll need to do is proxy calls to the backend server during development.

## Dev server proxy

*vue.config.js*

```
module.exports = {
  ...
  devServer: {
    proxy: "http://localhost:8070"
  }
};
```

The Webpack dev server we're using for our client app can proxy any requests it receives that can't be resolved on that host.

We can use the Vue CLI config file to set this up by using the devServer.proxy option which we'll set to the backend's URL.

## Production build

*server/index.js*

```
const express = require("express");
const app = express();
const path = require("path");

const mongoose = require("mongoose");
mongoose.connect("mongodb://localhost:27017/full-stack-vue");

app.use("/items", require("./routes/items"));

if (process.env.NODE_ENV === "production") {
  app.use("/dist", express.static(path.join(__dirname, "dist")));
}

app.listen(8070, () => {
  console.log("Listening on port 8070");
});
```

So far we've focused on the development phase of the project, what about production?

When you go to build this app you'll use the npm run build command which will compile your frontend files into the dist folder of your project directory.

### Dist folder

You can then serve the built files from the backend by statically serving the dist folder.

Actually, you can serve your files like this in a prototype or staging envrionment, but in production you should probably serve from a CDN.

## Server-side rendering

*vue.config.js*

```
module.exports = {
  ...
  configureWebpack() {
    // SSR config
  }
};
```

*nuxt.config.js*

```
module.exports = {
  serverMiddleware: [
    '~/server'
  ],
};
```

Before we finish, I want to briefly discuss server-side rendering. This is a state-of-the-art feature that you'd reasonably want to include in an enterprise application.

You can add server-side rendering to a Vue CLI project pretty much the same way you would for any Vue and Node app.

For development, you'll also need to expand the Webpack settings in the Vue CLI config file using the `configureWebpack` option.

However, if server-side rendering is important, you might considering using Nuxt.js instead of Vue.

As you may know, Nuxt provides provides a Node-based server out of the box which already includes a robust server-side rendering set up.

### Nuxt.js

This server provides an option to add your own custom middleware which is all you need to add your own API endpoints.

Doing this would mean we wouldn't need nodemon or the devServer proxy either!

### In the next video...(unit testing)

So that wraps up our discussion about full-stack Vue scaffolding. In the next video be talking about another crucial enterprise topic which is...

unit testing. Thanks for watching and I'll see you in the next video.