ASP.NET Core Vue CLI Templates

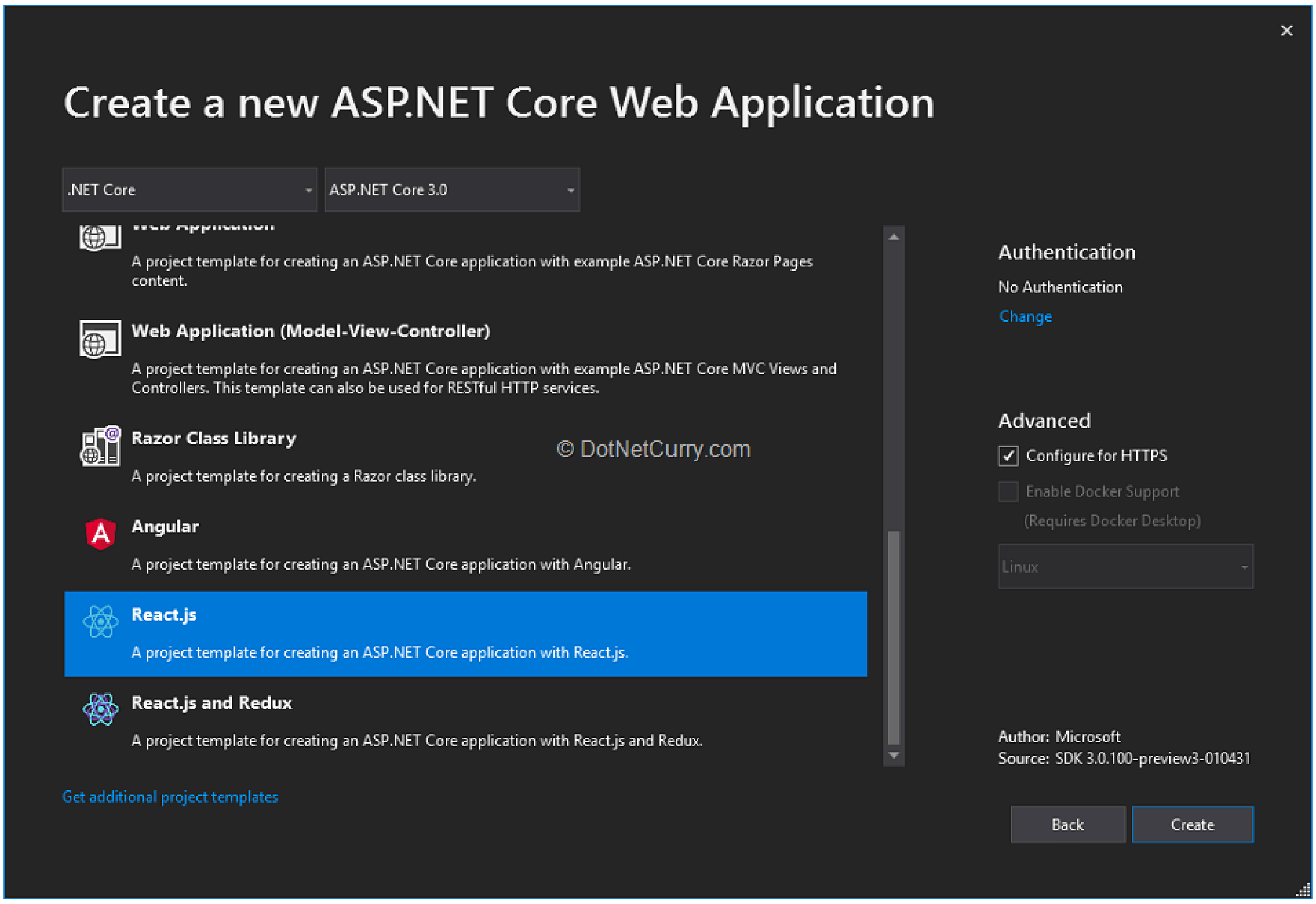
Posted by: Daniel Jimenez Garcia (../../Author.aspx?AuthorName=Daniel Jimenez Garcia) , on 6/26/2019, in **Category** ASP.NET Core (../../BrowseArticles.aspx?CatID=88)

Views: 6261 (https://facebook.com/sharer/sharer.php?u=https%3A%2F%2Fwww.dotnetcurry.com%2Faspnet-core%2F1500%2Faspnet-core-vuejs-template) (https://twitter.com/intent/tweet?text=ASP.NET%20Core%20Vue%20CLI%20Templates%20%7C%20DotNetCurry&url=https%3A%2F%2Fwww.dotnetcurry.com%core%2F1500%2Faspnet-core-vuejs-template) (https://www.linkedin.com/shareArticle?mini=true&amp;url=https%3A%2F%2Fwww.dotnetcurry.com%2Faspnet-core%2F1500%2Faspnet-core-vuejs-template&title=ASP.NET%20Core%20Vue%20CLI%20Templates%20%7C%20DotNetCurry)

Abstract: This tutorial discusses several approaches to integrate Vue.js with ASP.NET Core (including a template), all of them using Vue.js applications generated by the Vue CLI.

Vue.js is becoming one of the most popular and loved web frameworks, and its CLI 3.0 (https://cli.vuejs.org/) makes creating and working with Vue.js applications easier than ever.

However, when it comes to the official SPA templates provided by ASP.NET Core, you might have noticed they only support Angular and React out-of-the-box.



A project template for creating an ASP.NET Core application with example ASP.NET Core Razor Pages content.

Web Application (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

Razor Class Library
A project template for creating a Razor class library.

Angular
A project template for creating an ASP.NET Core application with Angular.

React.js
A project template for creating an ASP.NET Core application with React.js.

React.js and Redux
A project template for creating an ASP.NET Core application with React.js and Redux.

Author: Microsoft
Source: SDK 3.0.100-preview3-010431

Figure 1, SPA templates included in ASP.NET Core out of the box

What about Vue.js then?

Microsoft has so far declined (https://github.com/aspnet/JavaScriptServices/pull/1726)(at least for the moment) to include support for the Vue CLI, so it is up to the community to fill the gap in their SPA templates.

In this article we will discuss several options for integrating ASP.NET Core and the Vue CLI, one of them already available in NuGet (https://www.nuget.org/packages/SoftwareAteliers.AspNetCoreVueStarter/), thanks to Software Ateliers.

If you have chosen Vue.js and ASP.NET Core as your web stack, I hope this article will help you understand how both fit together, and at the same time, help you get started.

You can find the **companion source code** in GitHub (https://github.com/DaniJG/ASPCoreVueCLITemplates).

Things move fast in the web development world. I wrote about a year and a half ago a similar article (https://www.dotnetcurry.com/aspnet/1383/modern-web-dev-aspnet-core-webpack-vuejs) *, describing a template which can now be considered obsolete. This is mostly due to the rise of the Vue CLI 3.0 and the newer webpack versions!*

Combining ASP.NET Core projects and Vue CLI projects

Writing web applications using a modern framework like Vue.js isn’t a simple enterprise.

The Vue.js application code is composed of a mixture of vue files (https://vuejs.org/v2/guide/single-file-components.html), .js/.ts, .css/.sass/.less files and some static files like images or fonts. For stitching everything together there is tooling like Babel (https://babeljs.io/) and webpack (https://webpack.js.org/), which will combine all the different source files into a final set of JavaScript and CSS files that browsers can interpret.

This means a Vue.js application (and applications in most other modern frameworks like Angular or React for that matter) needs to be ***bundled*** before it can be run in a browser. You can consider the bundling process the equivalent of compiling a .NET application.

Running the application during development

The need to bundle the application introduces friction during the development process, since bundles need to be *regenerated* after making changes to the source files before you can execute the updated code in the browser.

The way Vue.js deals with this (and again, so do other modern frameworks) is to provide a development web server as part of its tooling. This web server generates the bundles on startup and keeps watching for file changes, automatically regenerating the bundles and even pushing the changes to the browser.

If you and/or your team is working on a full-stack, as soon as you add a traditional web server framework like ASP.NET Core providing the backend API, the development cycle of your application suddenly increases in complexity. You now have two servers to get started during development:

- The Vue.js development server that provides the HTML/JS/CSS to be run in the browser
- The ASP.NET Core server providing the backend API that your application will send requests to

The following diagram explains this situation:

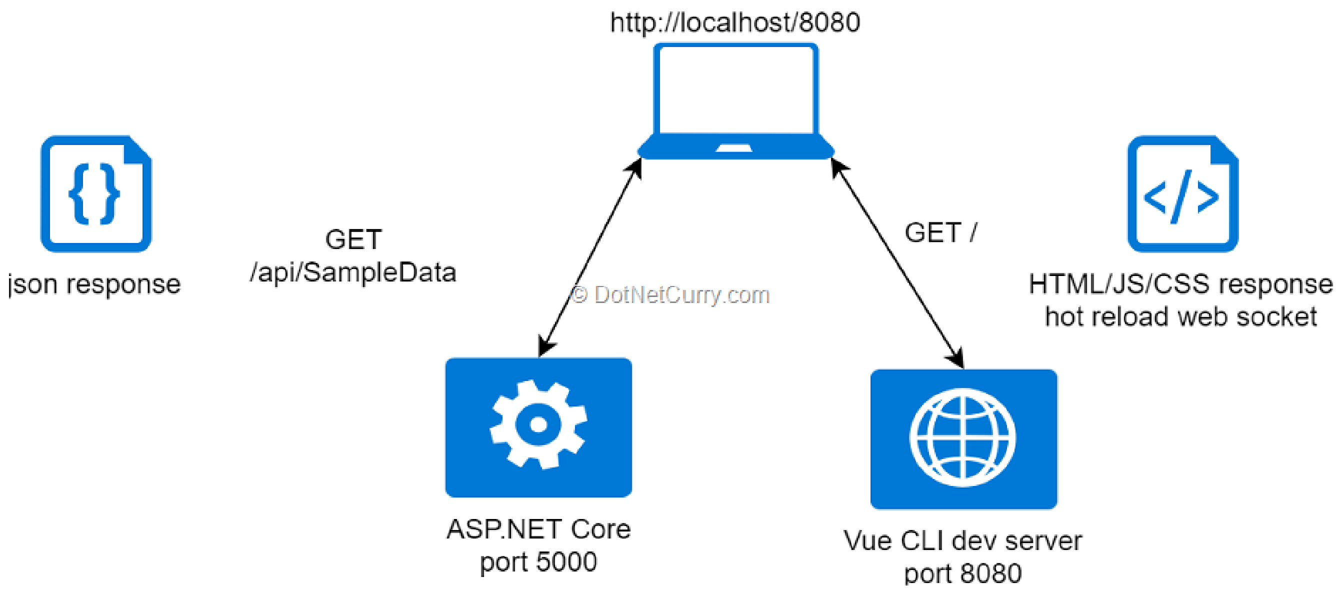


Figure 2, developing with a Vue.js application and an ASP.NET Core application

This approach requires no special templates or support from either Vue.js or ASP.NET Core. You will be able to use the best tool to write/debug each application, but you will have to manually coordinate starting/stopping them.

This can work great with bigger teams that split frontend and backend responsibilities or with experienced developers who can switch between each side and understand the different tooling involved in both.

An alternative approach is the one followed by the Angular and React SPA templates provided by ASP.NET Core. These templates use ASP.NET Core server as the main web server, with all the browser traffic directed to it. During startup, the Vue.js development server is launched as a child process, and traffic other than the one for API controllers, is proxied to it:

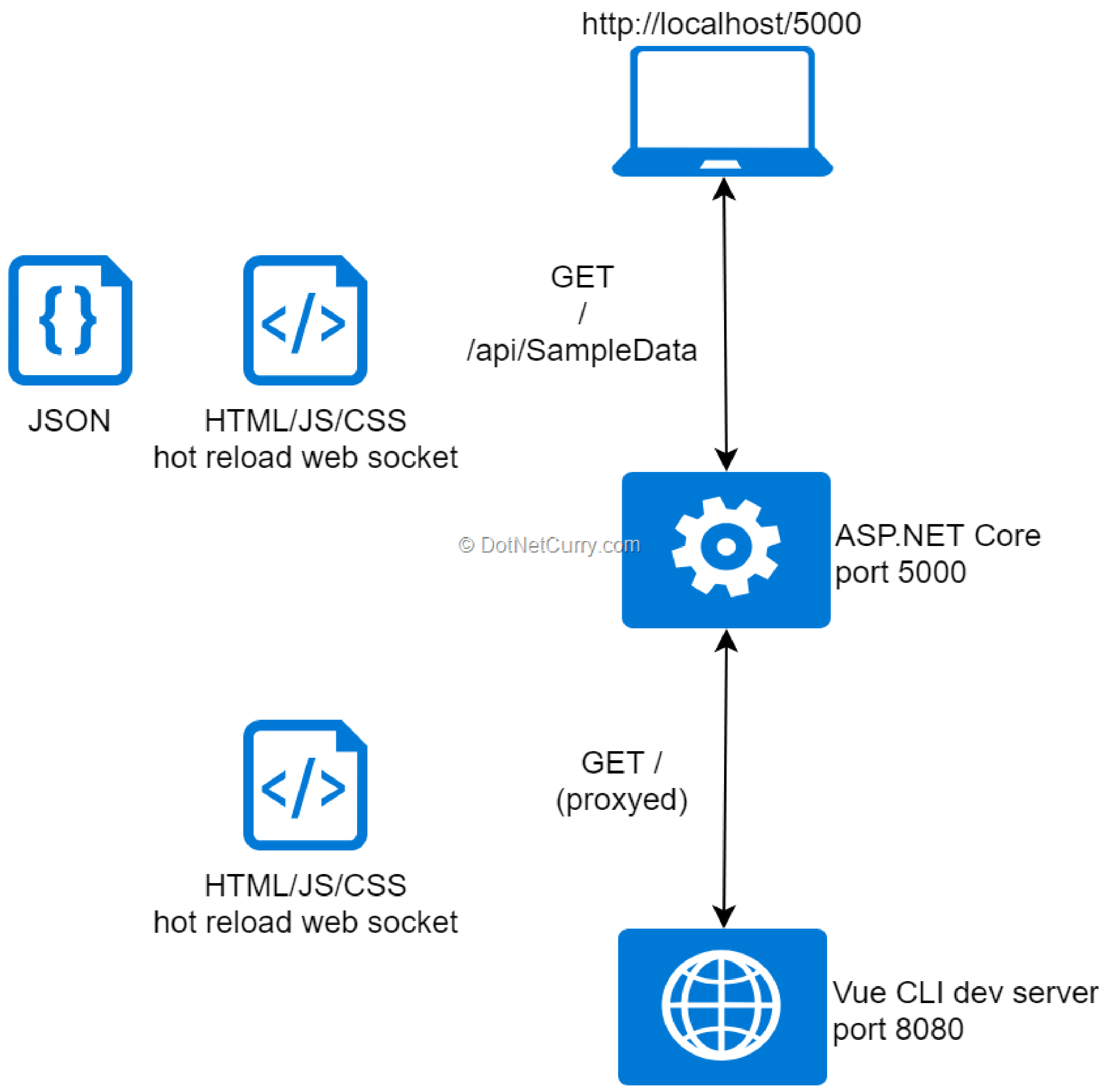


Figure 3, Using ASP.NET Core as the main server, proxying requests to the Vue.js development server

This approach lets you keep frontend and backend together during the development cycle, reducing the friction to run and debug the application on your local machine. With a single Start or F5 command, you get both servers started, while the *Build* command builds everything into a single package ready to be deployed.

There is a third option which inverts the roles of both servers. Since the Vue.js development server is nothing but a webpack development server (https://webpack.js.org/configuration/dev-server/), we can use its proxying capabilities to invert the situation:

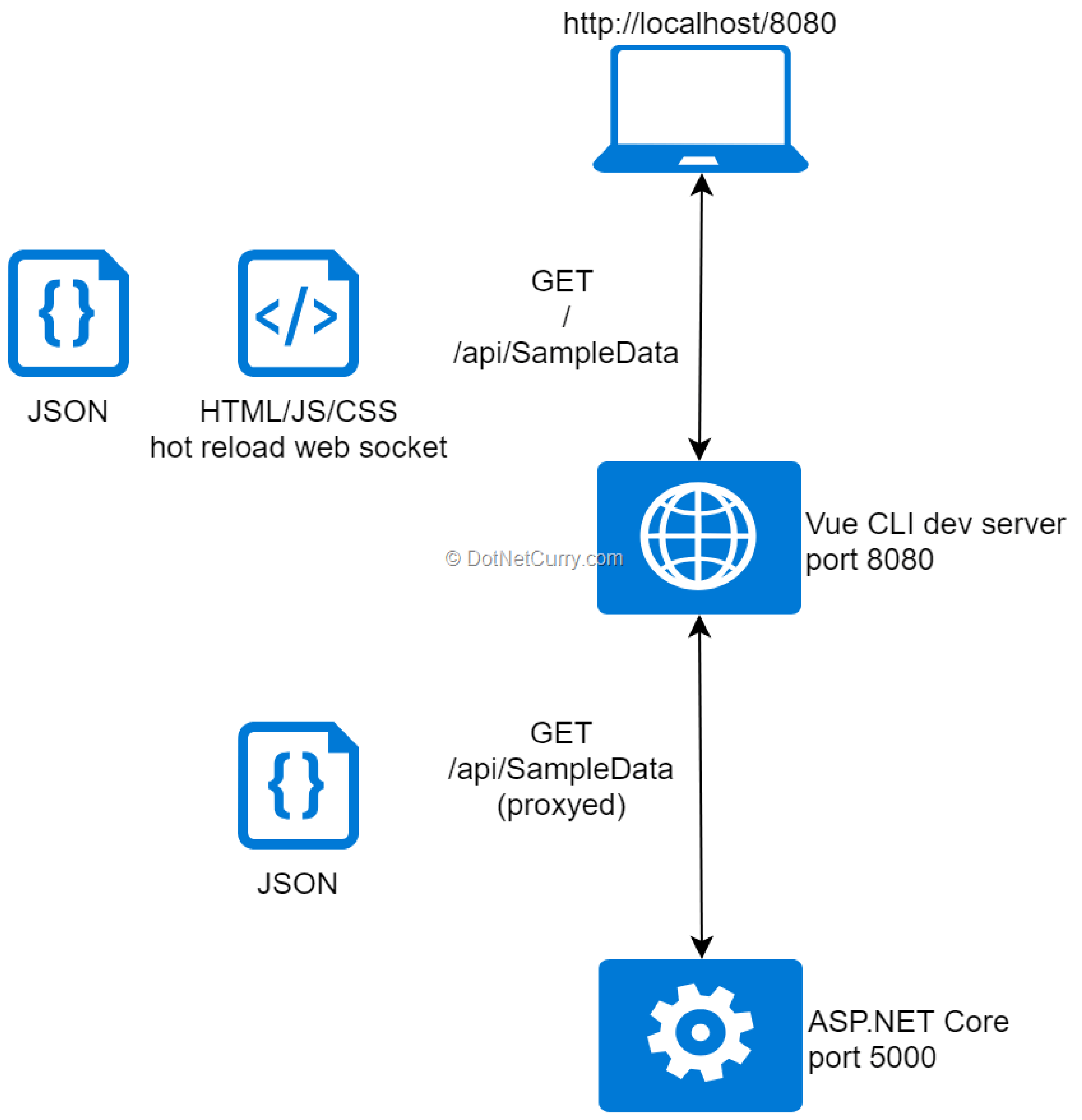


Figure 4, Using Vue.js development server as the main server, proxying requests to the ASP.NET Core server

While this option might seem redundant, there are certain advantages compared to the previous scenario:

- The Vue.js development server has been designed to bundle and push Vue.js applications to the browser. It makes sense for it to be the one the browser directly talks to.
- The hot reload features rely on web sockets so the Vue.js development server can push the new bundles to the browser as soon as there are any code changes. Proxying them introduces unneeded complexity into the ASP.NET Core SPA proxy and extra latency.
- The ASP.NET Core server works mainly as an API server, which is better aligned to the role it plays in your system.

Running the application during production

The previous discussion focused on how to run your application during development, reducing the friction between making changes to your code and running them. When it comes to deploying to production, things are different.

During build time, your Vue.js application is *bundled* by webpack into a set of static HTML/JS/CSS files, while your ASP.NET Core application is *compiled* into dll/exe files. The simplest approach is to host everything together in an ASP.NET Core web server that serves both the static files and the requests to the API controllers:

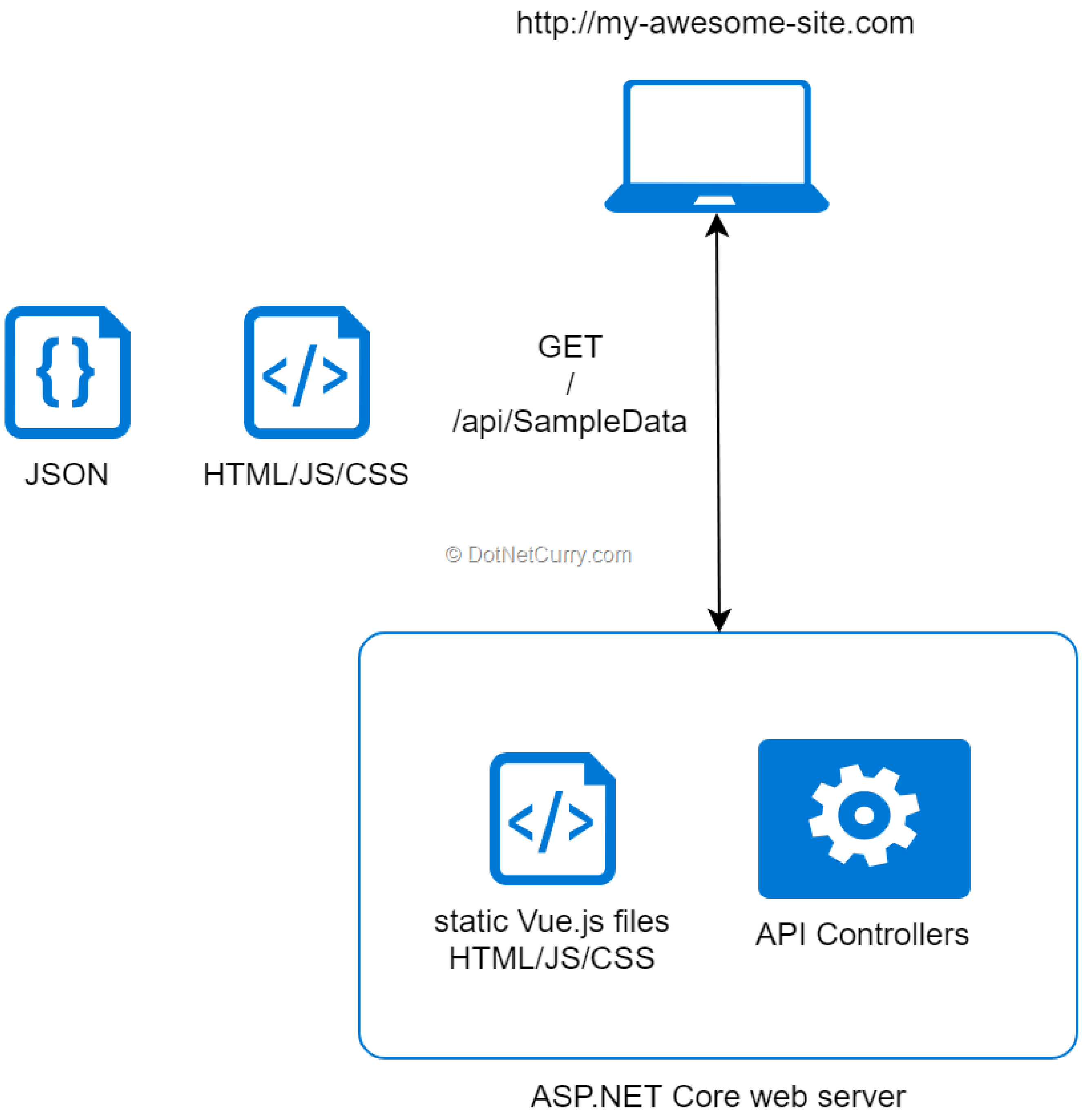


Figure 5, Hosting together the bundled Vue.js application and the ASP.NET Core server

More advanced options are available, hosting the generated bundles and the ASP.NET Core application separately. While this is a more complicated model, it certainly has its advantages, letting you scale each independently and use hosting options that suit each of them. (For example, taking advantage of CDNs for your bundles.)

With the development and build cycle being the same, we will leave this outside of the scope of the article, but I encourage you to do some research if interested.

The next sections discuss the three different options we saw to organize your project during development. You can **find the source code** on GitHub (https://github.com/DaniJG/ASPCoreVueCLITemplates).

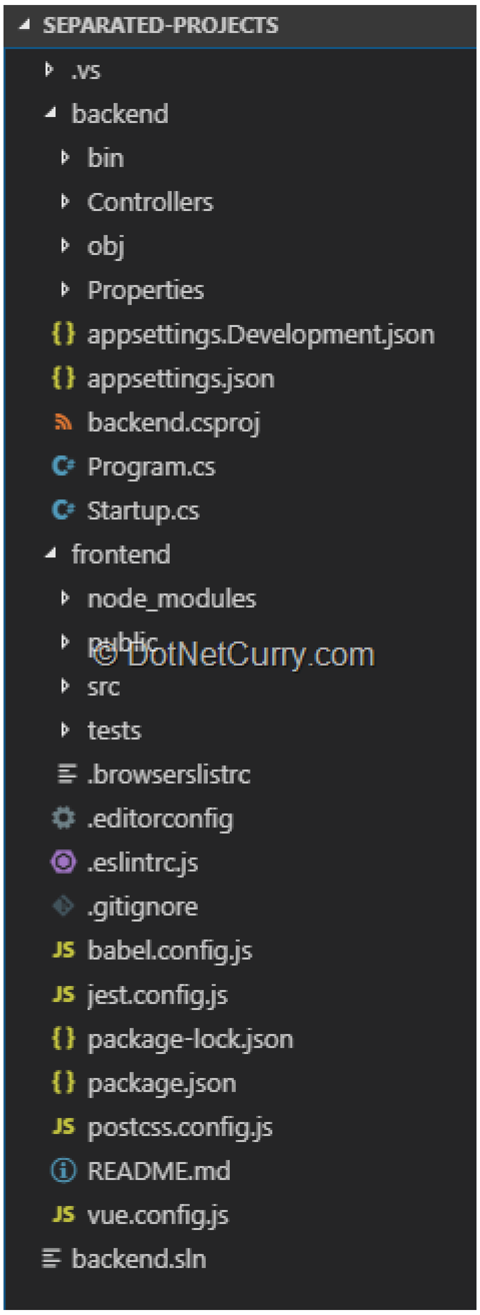
Separate Projects

Although this is the simplest of the solutions, it is a powerful one and will be very frequently chosen by big and/or experienced teams. It decouples frontend from backend, making it easier to adopt different tooling, development process, etc.

You will typically create two folders or even two repos, usually following some naming convention like backend/frontend or client/server. In one of them, simply create a new ASP.NET Core web application using the Web API template. You can use either the Visual Studio new application wizard or run in the command line `dotnet new webapi` .

In the other folder, create a new Vue.js application using the Vue CLI (https://cli.vuejs.org/) by running `vue create` from the command line. Then select the options that you want to enable for your Vue application (Vuex, Router, TypeScript, Linting, Testing, etc.)

At the end of this process, you should have a project structure like the following one, with the ASP.NET Core and Vue.js applications in separate folders:



```
SEPARATED-PROJECTS
  .vs
  backend
    bin
    Controllers
    obj
    Properties
    appsettings.Development.json
    appsettings.json
    backend.csproj
    Program.cs
    Startup.cs
  frontend
    node_modules
    public
    src
    tests
    .browserslistrc
    .editorconfig
    .eslintrc.js
    .gitignore
    babel.config.js
    jest.config.js
    package-lock.json
    package.json
    postcss.config.js
    README.md
    vue.config.js
  backend.sln
```

Figure 6, folder structure with separate frontend and backend projects

There is one thing you need to do before you can start both applications.

Since each will run on its own web server (Vue.js typically on localhost:8080 and ASP.NET Core on localhost:5000 with Kestrel or an address like localhost:49960 with IISExpress) you will need to enable communication between the two processes. You can enable CORS by adding the following line to the ConfigureServices method of the Startup class:

```
services.AddCors();
```

Next you need to specify which CORS policy should be used.

Update the Configure method of the Startup class to use a wide policy in case of development:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseCors(policy => policy
        .AllowAnyHeader()
        .AllowAnyMethod()
        .WithOrigins("http://localhost:8080")
        .AllowCredentials());
}
```

This way when you send a request to the ASP.NET Core backend application from your Vue.js application running on localhost:8080, the browser won't block the request.

An alternative approach is to send requests from the Vue.js application as if the backend was hosted in the same location as the Vue.js application (i.e., send a request to localhost:8080/api/SampleData/WeatherForecast rather than localhost:5000/api/SampleData/WeatherForecast).

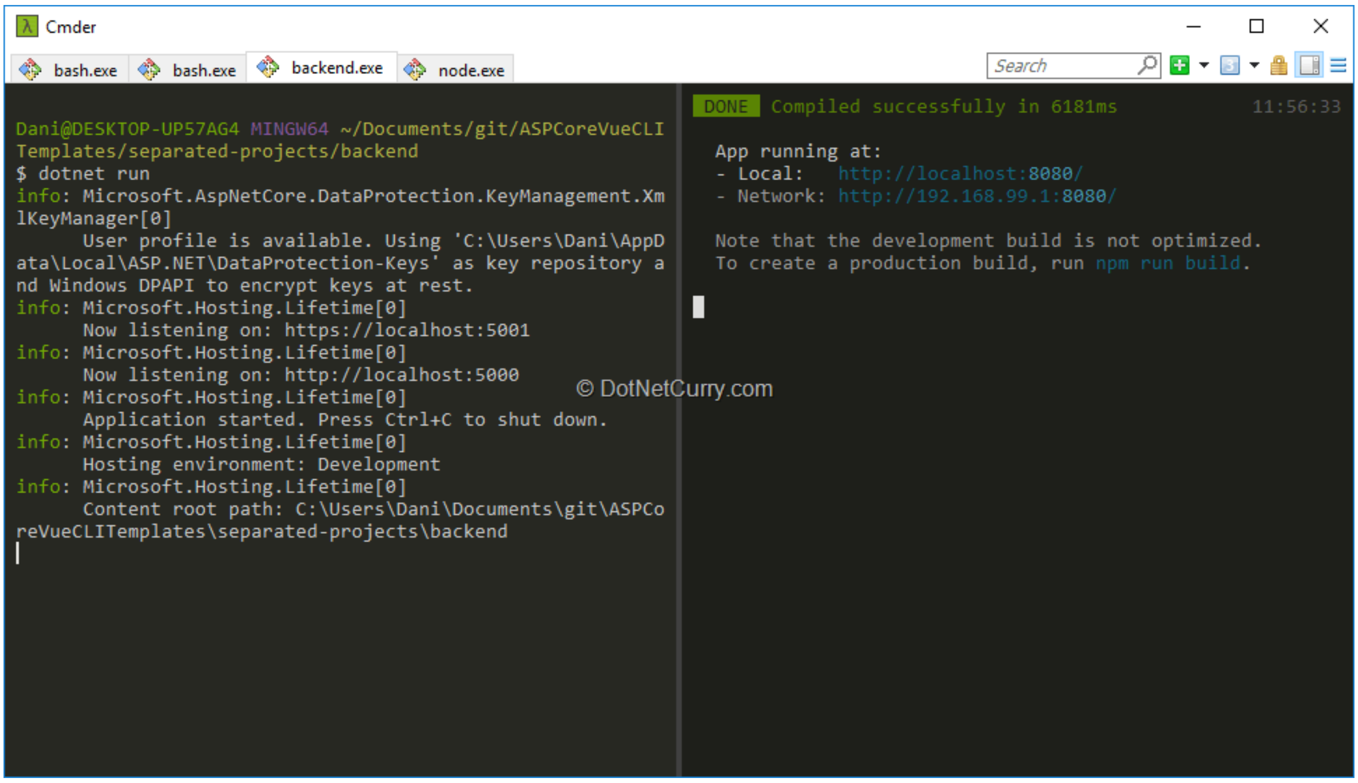
For this approach you will need to enable the proxy (https://cli.vuejs.org/config/#devserver-proxy) in the Vue.js web development server, pointing it to the ASP.NET Core application. The Vue.js development server will then send to the ASP.NET Core those application requests that it cannot solve itself.

Simply add a **vue.config.js** file to the root of the Vue.js application with these contents:

```
module.exports = {
  // The URL where the .NET Core app will be listening.
  // Specific port depends on whether IISExpress/Kestrel and HTTP/HTTPS are
used
  devServer: {
    proxy: 'http://localhost:5000'
  },
}
```

Let's verify if it all works.

Open two separate terminals and run dotnet run in one and npm run serve in the other. They should start the ASP.NET Core and Vue.js applications respectively, and they will print on which specific HTTP(S) port each is listening.



```
Dani@DESKTOP-UP57AG4 MINGW64 ~/Documents/git/ASPCoreVueCLI
Templates/separated-projects/backend
$ dotnet run
info: Microsoft.AspNetCore.DataProtection.KeyManagement.Xm
lKeyManager[0]
      User profile is available. Using 'C:\Users\Dani\AppD
ata\Local\ASP.NET\DataProtection-Keys' as key repository a
nd Windows DPAPI to encrypt keys at rest.
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\Dani\Documents\git\ASPCo
reVueCLITemplates\separated-projects\backend
```

```
 DONE  Compiled successfully in 6181ms                    11:56:33

  App running at:
  - Local:   http://localhost:8080/
  - Network: http://192.168.99.1:8080/

  Note that the development build is not optimized.
  To create a production build, run npm run build.
```

Figure 7, Running separately the frontend and backend with “dotnet run” and “npm run serve”

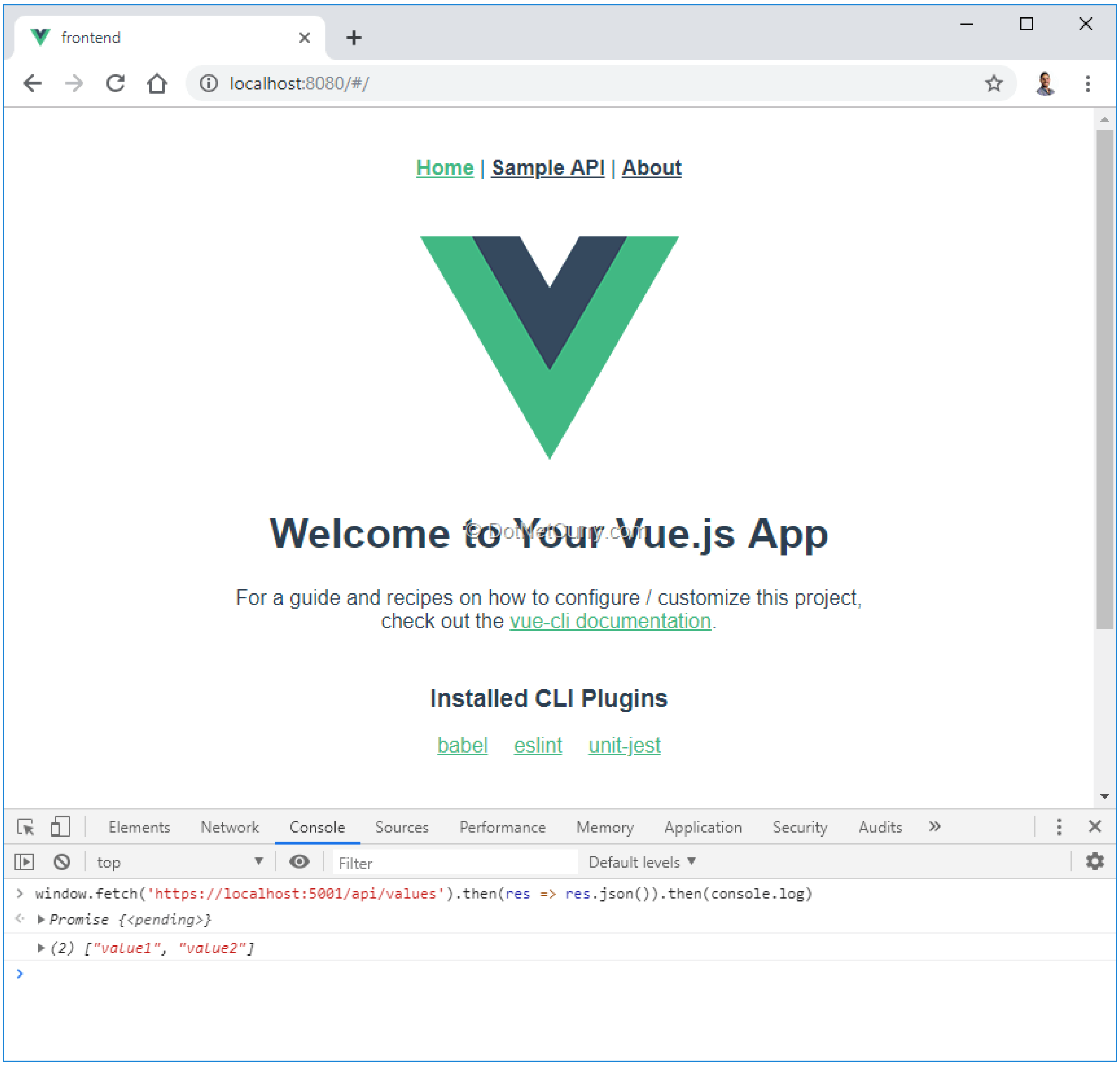
The Vue development server will typically start on http://localhost:8080 (http://localhost:8080). If you open that URL in the browser, you should see the default home page. Try changing the message in the Home.vue component and you should immediately see it changing in the browser – thanks to the hot reload capabilities of the Vue.js dev server.

Now let’s try to fetch the sample values returned by the ValuesController.

We need to send a request to the location where the ASP.NET Core application is running, in my case https://localhost:5001 (https://localhost:5001) since I had HTTPS enabled (You should see this address in the command line where you run dotnet run or in the Visual Studio output if running from Visual Studio).

Execute the following in the browser dev tools and you should see the values logged to the console:

```
window.fetch('https://localhost:5001/api/values')
  .then(res => res.json())
  .then(console.log)
```


frontend
localhost:8080/#/
Home | Sample API | About
Welcome to Your Vue.js App
For a guide and recipes on how to configure / customize this project, check out the vue-cli documentation.
Installed CLI Plugins
babel eslint unit-jest
Elements Network Console Sources Performance Memory Application Security Audits
top
Filter
Default levels

```
> window.fetch('https://localhost:5001/api/values').then(res => res.json()).then(console.log)
< ▶Promise {<pending>}
  ▶(2) ["value1", "value2"]
>
```

Figure 8, Accessing the ASP.NET Core backend from the Vue.js application

You now have a project made of a separate Vue.js frontend application and an ASP.NET Core backend application!

This is the simplest approach for integrating Vue.js and ASP.NET Core, using out-of-the-box tooling in each case, but that doesn't mean this isn't a good approach! Quite the contrary, you can now fully embrace the different tooling and development process for each.

For example:

- You might run the Vue.js application by running `npm run serve` from the command line. However, you might run the ASP.NET Core application from Visual Studio or by running `dotnet run` from the command line.
- You might use Visual Studio Code with plugins like Vetur (https://vuejs.github.io/vetur/) and Chrome with the Vue Dev Tools (https://github.com/vuejs/vue-devtools) for developing and debugging the Vue.js application. However, you might use Visual Studio to develop and debug the ASP.NET Core application.
- When it comes to publishing and hosting them, you might decide to keep them separate, with the ASP.NET Core application as a pure API server while a different server takes care of serving the built

Vue.js application bundles. Or you can decide to include the built Vue.js application bundles as static files in the ASP.NET Core application, using the Static middleware to serve them.

It certainly introduces an additional layer of complexity in your project, but using the right tool for the job might be worth the steeper learning curve. If you have been following the DNC magazine, you might have recognized this approach in articles like the one integrating SignalR with Vue.js (https://www.dotnetcurry.com/aspnet-core/1480/aspnet-core-vuejs-signalr-app).

The next two sections discuss approaches aiming to provide an integrated experience for the entire project.

Adapting the React SPA template

If you would rather keep a more integrated development experience for both projects, simplifying the tooling involved and avoiding the need to run frontend and backend separately, you are probably looking at the official SPA templates included with ASP.NET Core.

These templates essentially add some middleware to the ASP.NET Core application so:

- The frontend development server (manually run with `npm run serve` in our previous approach) is automatically started whenever you start your ASP.NET Core application in development mode.
- Requests for routes that don't match any ASP.NET Core controller are proxied to the frontend development server, giving the browser the impression that everything is running from within the ASP.NET Core application.

We saw a diagram of this approach in the initial section:

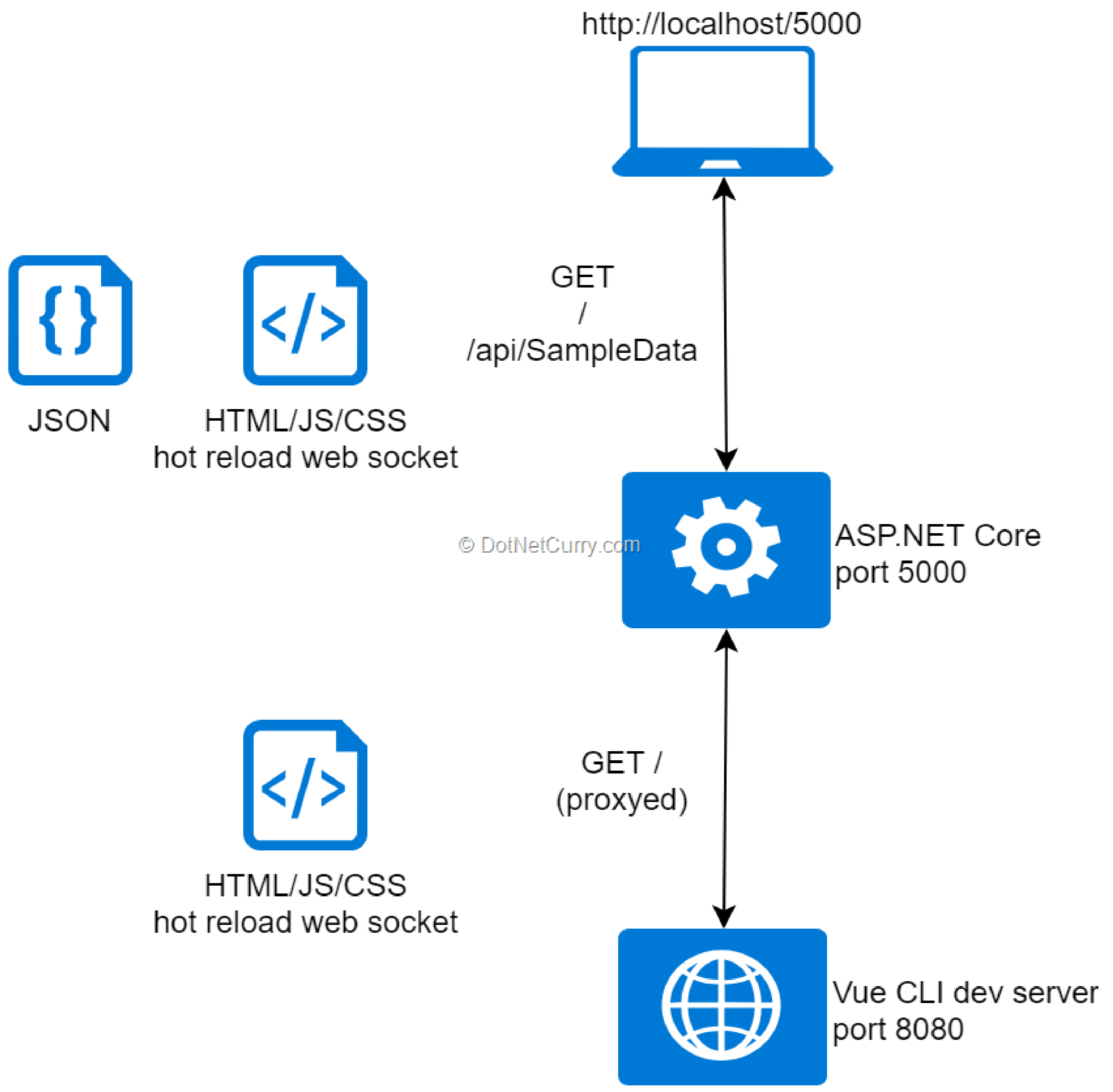


Figure 9, SPA template where ASP.NET Core proxies the Vue.js development server

Now the problem you will find is that ASP.NET Core only provides SPA templates for Angular and React, with Vue.js sadly missing and officially unsupported by Microsoft (https://github.com/aspnet/JavaScriptServices/pull/1726) (at least for the moment).

The good news is that it isn't hard to adapt one of the existing templates, since it is mostly a matter of executing a different npm script during startup.

The bad news is that the utilities needed to run an npm script during startup and to wait for the Vue.js development server to be started are internal to Microsoft's SPAServices.Extensions package (https://github.com/aspnet/AspNetCore/tree/master/src/Middleware/SpaServices.Extensions/src), so you will need to manually copy their source code into your project.

Adapting the React template is exactly what Software Ateliers has done, so you can now install their template (https://github.com/SoftwareAteliers/asp-net-core-vue-starter) and generate a Vue.js SPA application using this approach.

In the rest of this section I will show you how you would manually create such a template, so you understand how it works.

Manually adapting the React template

Step 1. Replacing React with Vue.js in the React SPA template

Start by creating a new ASP.NET Core application, selecting the React template. Either use the new project wizard in Visual Studio or run `dotnet new react` from the command line.

Once generated, the first thing we need to do is to replace the contents of the /ClientApp folder with a Vue application. To do so, take the following steps:

- Remove the existing /ClientApp folder
- From the project root, run `vue create client-app` (The Vue CLI does not allow upper case laters in project names). Use this step as a chance to select any features you want in your Vue.js project, like TypeScript or Unit Testing, rather than relying on the ones selected by someone in a template!
- Once the Vue CLI finishes generating the project, rename the /client-app folder to /ClientApp

With these three sub-steps, we have fully replaced the React frontend app with a Vue.js frontend app.

Step 2. Using Middleware to launch the Vue.js development server during startup

Here comes the most complicated part! If you look at the `Configure` method of the `Startup` class, you will notice the following piece of code:

```
app.UseSpa(spa =>
{
    spa.Options.SourcePath = "ClientApp";
    if (env.IsDevelopment())
    {
        spa.UseReactDevelopmentServer(npmScript: "start");
    }
});
```

This code is using the methods `UseSpa` and `UseReactDevelopmentServer` that are part of the Microsoft.AspNetCore.SpaServices.Extensions NuGet package.

With `UseReactDevelopmentServer`, the development server gets started on a free port, and any request waits for it to be started before continuing.

With `UseSpa`, a proxy between the ASP.NET Core server and the development server is established, redirecting unknown requests to said development server.

The problem comes from the fact that although `UseReactDevelopmentServer` allows you to specify which npm script should be run (since with Vue.js we need to run `npm run serve` rather than `npm start`), it has internal hardcoded logic specific to React:

- Specify which port the development server should be started on
- Determine when the development server is started based on the output of the npm command

Since these two points are different in the case of a Vue.js application, and `UseReactDevelopmentServer` does not provide any options to change its behavior, we will need to create our own middleware that knows how to start the Vue.js development server. The process will be a little more involved than expected since it relies on utilities which are internal to the Microsoft.AspNetCore.SpaServices.Extensions package, but it won't be too hard.

1. Start by creating a new **Middleware** folder in you project, and manually copy the **Util** folder from Microsoft's source (https://github.com/aspnet/AspNetCore/tree/master/src/Middleware/SpaServices.Extensions/src/Util) (do not remove the license attribution!).

2. Next, you will also need to add to the Util folder the NpmScriptRunner (https://github.com/aspnet/AspNetCore/blob/master/src/Middleware/SpaServices.Extensions/src/Npm/NpmScriptRunner.cs) class.

3. Finally copy the React middleware and extension (https://github.com/aspnet/AspNetCore/tree/master/src/Middleware/SpaServices.Extensions/src/ReactDevelopmentServer) classes into your project's Middleware folder, and rename them as VueDevelopmentServerMiddleware and VueDevelopmentServerMiddlewareExtensions respectively.

The end result should look like this:

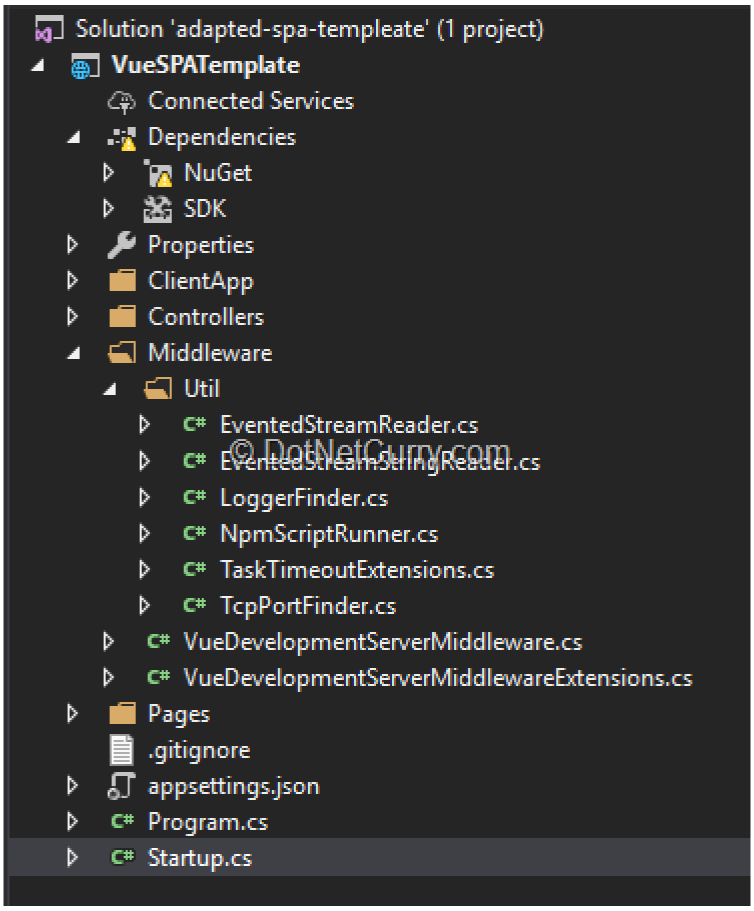


Figure 10, Adapting the React development server middleware for Vue.js

You can ignore the classes in the Util folder since we will use them as-is, the problem is that they are internal classes to the Microsoft.AspNetCore.SpaServices.Extensions package, so we couldn't access them. Simply rename their namespace, and you are free to concentrate on the middleware and extension classes.

Open the middleware class, which is the sole place where we actually need to make the changes!

Rename the StartCreateReactAppServerAsync method to StartVueDevServerAsync. Remove the environment variables and replace the creation of the NpmScriptRunner with:

```
var npmScriptRunner = new NpmScriptRunner(
  sourcePath, npmScriptName, $"--port {portNumber} --host localhost", null);
```

This is just updating the way the port is provided to the Vue.js development server. Now replace the WaitForMatch line so it waits for the Vue.js development server to print "DONE":

```
startDevelopmentServerLine = await npmScriptRunner.StdOut.WaitForMatch(
    new Regex("DONE", RegexOptions.None, RegexMatchTimeout));
```

Then rename the method UseReactDevelopmentServer in the middleware extension class to UseVueDevelopmentServer.

Once renamed, update the Configure method of the Startup class to call app.UseVueDevelopmentServer with the npm run serve script:

```
app.UseSpa(spa =>
{
    spa.Options.SourcePath = "ClientApp";
    if (env.IsDevelopment())
    {
        spa.UseVueDevelopmentServer(npmScript: "serve");
    }
});
```

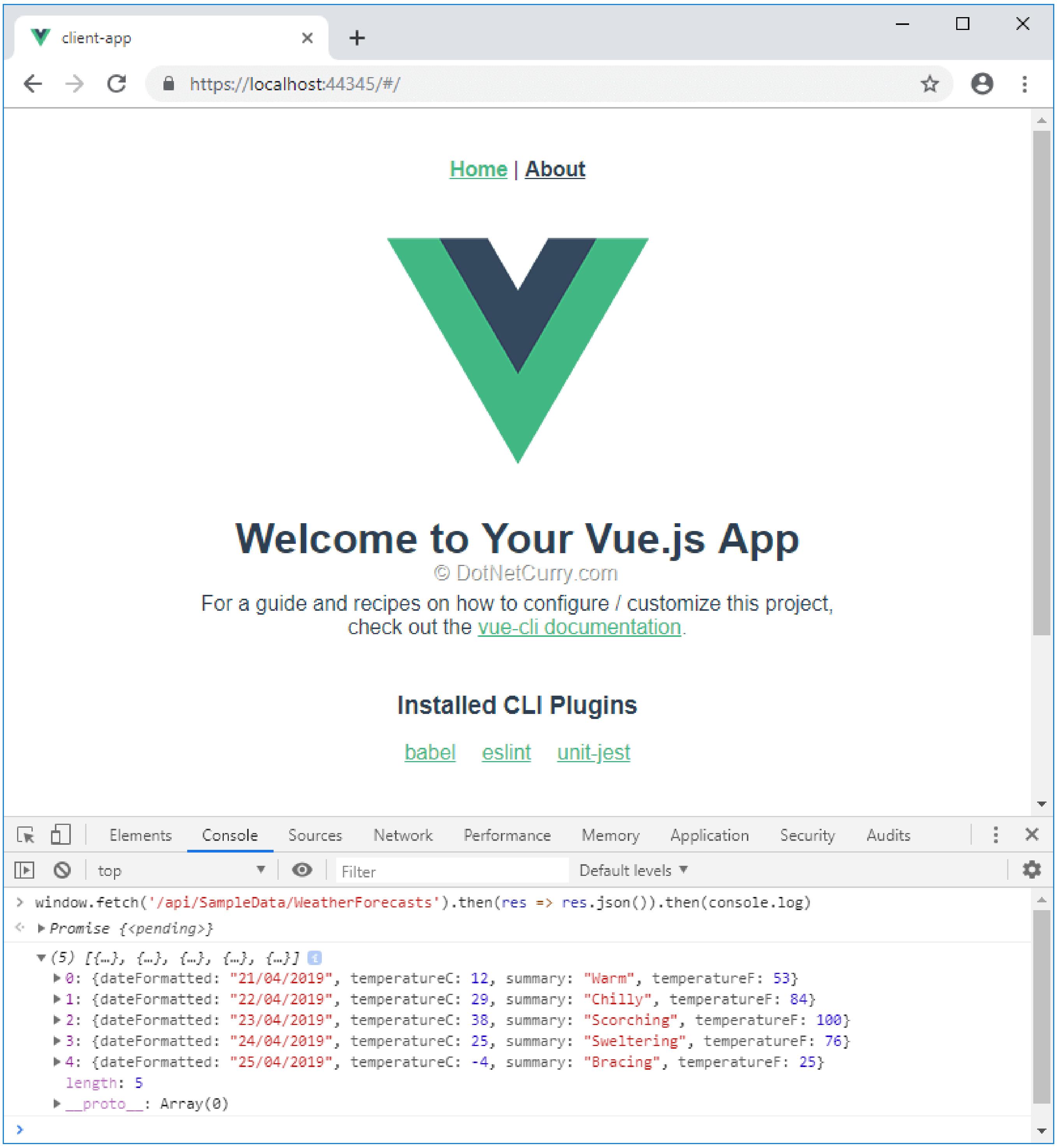
Once you are done, you should be able to start the project from Visual Studio or from the command line. The middleware will start the Vue.js development server and will wait for it to be initialized before loading the home page.

You should eventually see the Vue.js home page, but notice it is getting loaded from the port of the ASP.NET Core server rather than the Vue.js development server.

Try modifying the message in the Home.vue file and notice how the browser immediately reloads the changes.

That's great, it means the hot reload functionality keeps working even after we introduced the SPA proxy.

Now you can also try and load the sample forecast data without the need to specify the host since everything is running from the ASP.NET Core server:


client-app
https://localhost:44345/#/
Home | About
Welcome to Your Vue.js App
© DotNetCurry.com
For a guide and recipes on how to configure / customize this project, check out the vue-cli documentation.
Installed CLI Plugins
babel eslint unit-jest
Elements Console Sources Network Performance Memory Application Security Audits
top
Filter
Default levels

```
> window.fetch('/api/SampleData/WeatherForecasts').then(res => res.json()).then(console.log)
< ▶Promise {<pending>}
  ▼(5) [{…}, {…}, {…}, {…}, {…}]
    ▶0: {dateFormatted: "21/04/2019", temperatureC: 12, summary: "Warm", temperatureF: 53}
    ▶1: {dateFormatted: "22/04/2019", temperatureC: 29, summary: "Chilly", temperatureF: 84}
    ▶2: {dateFormatted: "23/04/2019", temperatureC: 38, summary: "Scorching", temperatureF: 100}
    ▶3: {dateFormatted: "24/04/2019", temperatureC: 25, summary: "Sweltering", temperatureF: 76}
    ▶4: {dateFormatted: "25/04/2019", temperatureC: -4, summary: "Bracing", temperatureF: 25}
     length: 5
    ▶__proto__: Array(0)
>
```

Figure 11, Running the ASP.NET Core template

If you look at the output from your ASP.NET Core application, you will be able to see the port where the Vue.js development server has started (Remember our middleware is finding a free port and explicitly telling the Vue.js development server to use it, so it can later proxy requests to such known port).

This means you can open the Vue.js home page in the browser both by using the ASP.NET Core address and the Vue.js development server address! What the ASP.NET Core middleware is doing behind the scenes is proxying the requests to the Vue.js development server.

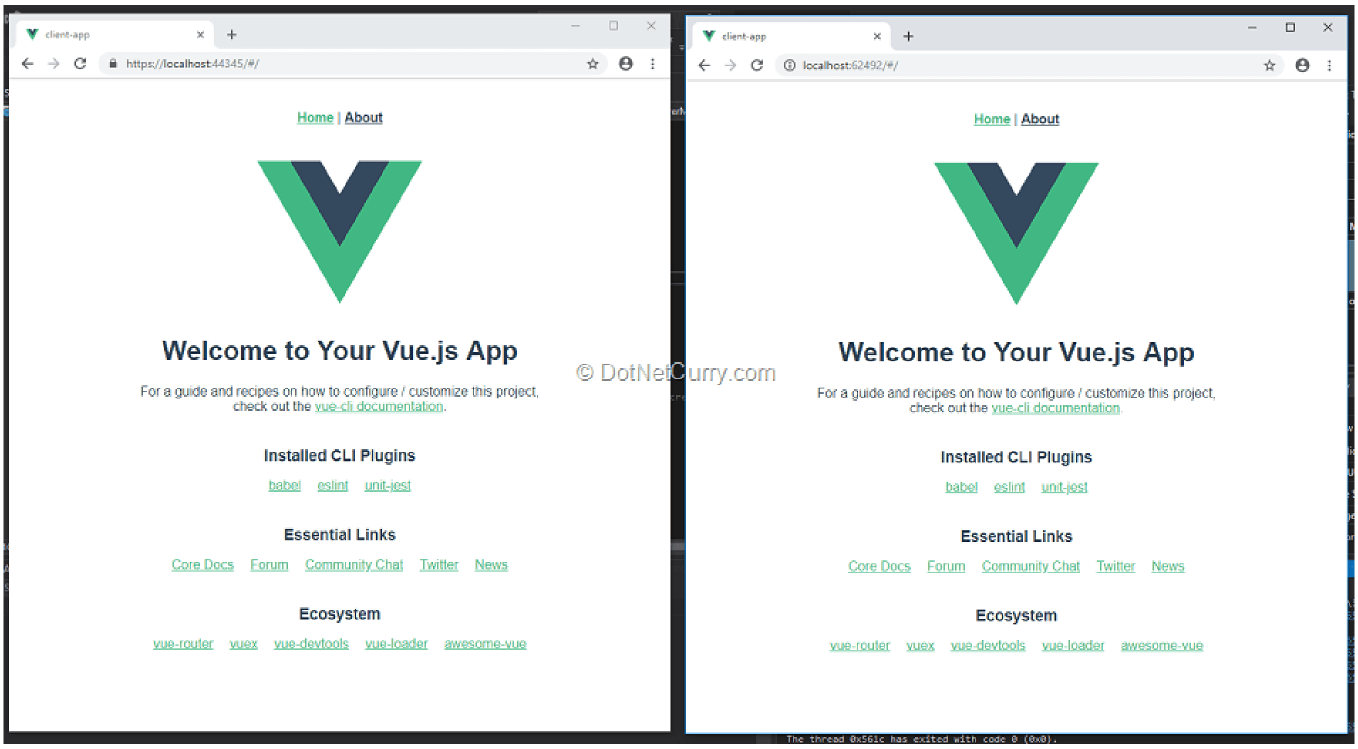

client-app
https://localhost:44345/#/
Home | About
Welcome to Your Vue.js App
For a guide and recipes on how to configure / customize this project, check out the vue-cli documentation.
Installed CLI Plugins
babel eslint unit-jest
Essential Links
Core Docs Forum Community Chat Twitter News
Ecosystem
vue-router vuex vue-devtools vue-loader awesome-vue
© DotNetCurry.com
client-app
localhost:62492/#/
Home | About
Welcome to Your Vue.js App
The thread 0x561c has exited with code 0 (0x0).

Figure 12, ASP.NET Core (left) proxes requests to the Vue.js development server (right)

At this point, it is worth mentioning you will see some errors in the console.

This is caused by a known issue (https://github.com/aspnet/AspNetCore/issues/7812) with ASP.NET Core trying to proxy all requests to the Vue.js development server and the web sockets used for hot reload. While hot reload will still work, you will see those errors during development.

Note: The alternative SPA template we will see later **does not have this problem.**

Another issue worth mentioning is that if you start the application from Visual Studio using the IISExpress profile, the npm process is not closed upon stopping the application!

This means the Vue.js development server is left running (if you get the port the development server was started, you will be able to still load it on the browser, like http://localhost:62495 (http://localhost:62495) in Figure 12) until you manually kill that process from the Task Manager.

This does not happen when starting with dotnet run, so you might want to keep using the alternate launch profile rather than IISExpress.

While everything will work since every time you start the server, a new free port is assigned to the Vue.js development server, the amount of running processes can become quite a burden after a few debugging sessions.

Step 3. Debugging from Visual Studio and Visual Studio Code

It is possible to debug C# and JavaScript code at the same time from both Visual Studio and Visual Studio Code.

But before we can do so, we need to update the generated source maps so their path is from the root of the project folder and not just from the ClientApp folder.

Fortunately for us, this is something relatively easy to do with the webpack development server used by Vue.js, with Vue.js giving us a hook to update the webpack configuration in the form of the **vue.config.js** file. You just need to create such a file with the following contents:

```
module.exports = {
  configureWebpack: {
    // Using source-map allows VS Code to correctly debug inside vue files
    devtool: 'source-map',
    // Breakpoints in VS and VSCode won't work since the source maps
    // consider ClientApp the project root, rather than its parent folder
    output: {
      devtoolModuleFilenameTemplate: info => {
        const resourcePath = info.resourcePath.replace('./src',
'./ClientApp/src')
        return `webpack:///${resourcePath}?${info.loaders}`
      }
    }
  }
}
```

Once created, make sure you have enabled *Script debugging* in Visual Studio and start debugging. Set a breakpoint in the **ClientApp/src/router.js** file and reload the page, notice the breakpoint is hit in Visual Studio and that Chrome shows it is stopped by a debugger!

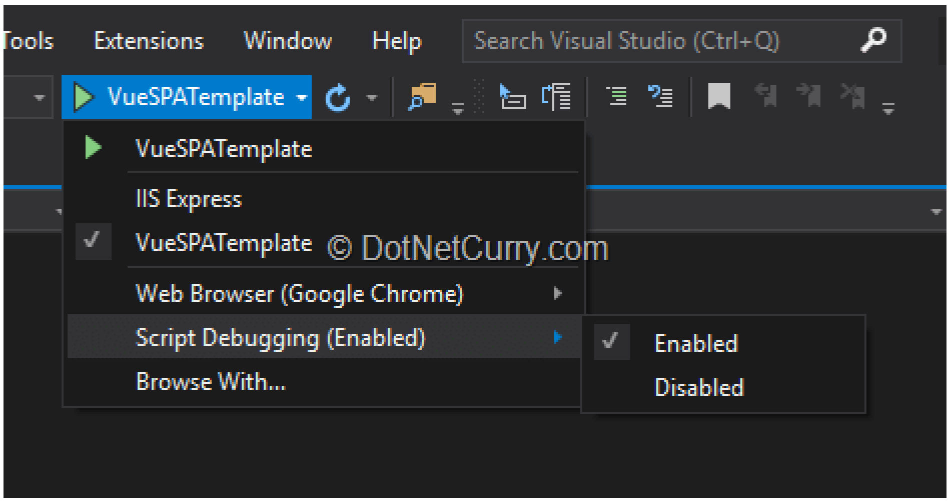
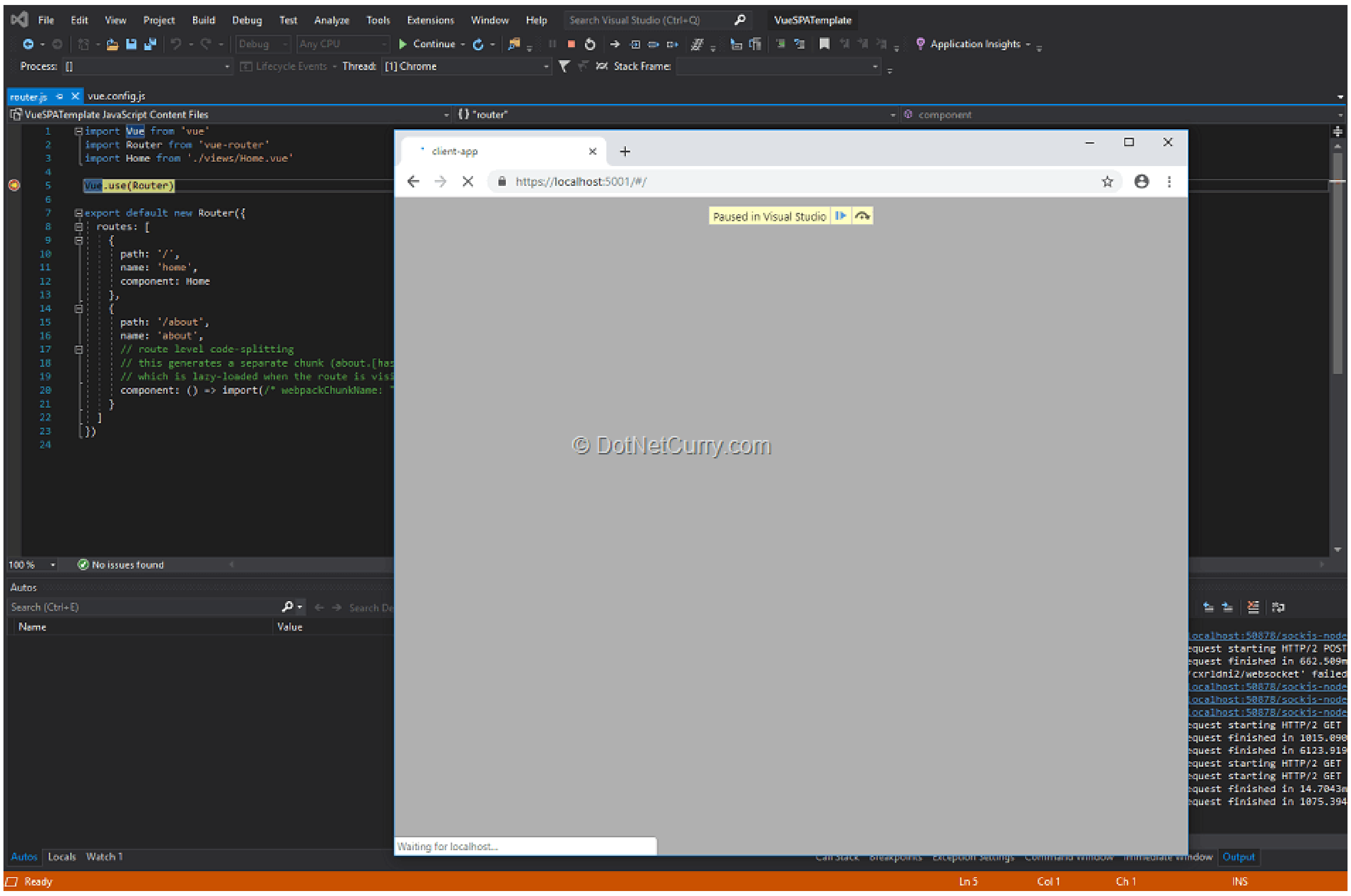


Figure 13, enabling script debugging in Visual Studio



```
import Vue from 'vue'
import Router from 'vue-router'
import Home from './views/Home.vue'

Vue.use(Router)

export default new Router({
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home
    },
    {
      path: '/about',
      name: 'about',
      // route level code-splitting
      // this generates a separate chunk (about.[has
      // which is lazy-loaded when the route is vis
      component: () => import(/* webpackChunkName: '
    }
  ]
})
```

```
localhost:50878/sockjs-node
equest starting HTTP/2 POST
equest finished in 662.509m
/cxr1dni2/websocket' failed
localhost:50878/sockjs-node
localhost:50878/sockjs-node
localhost:50878/sockjs-node
equest starting HTTP/2 GET
equest finished in 1015.898
equest finished in 6123.919
equest starting HTTP/2 GET
equest starting HTTP/2 GET
equest finished in 14.7043m
equest finished in 1075.394
```

Figure 14, Debugging Vue’s JavaScript code from Visual Studio on F5

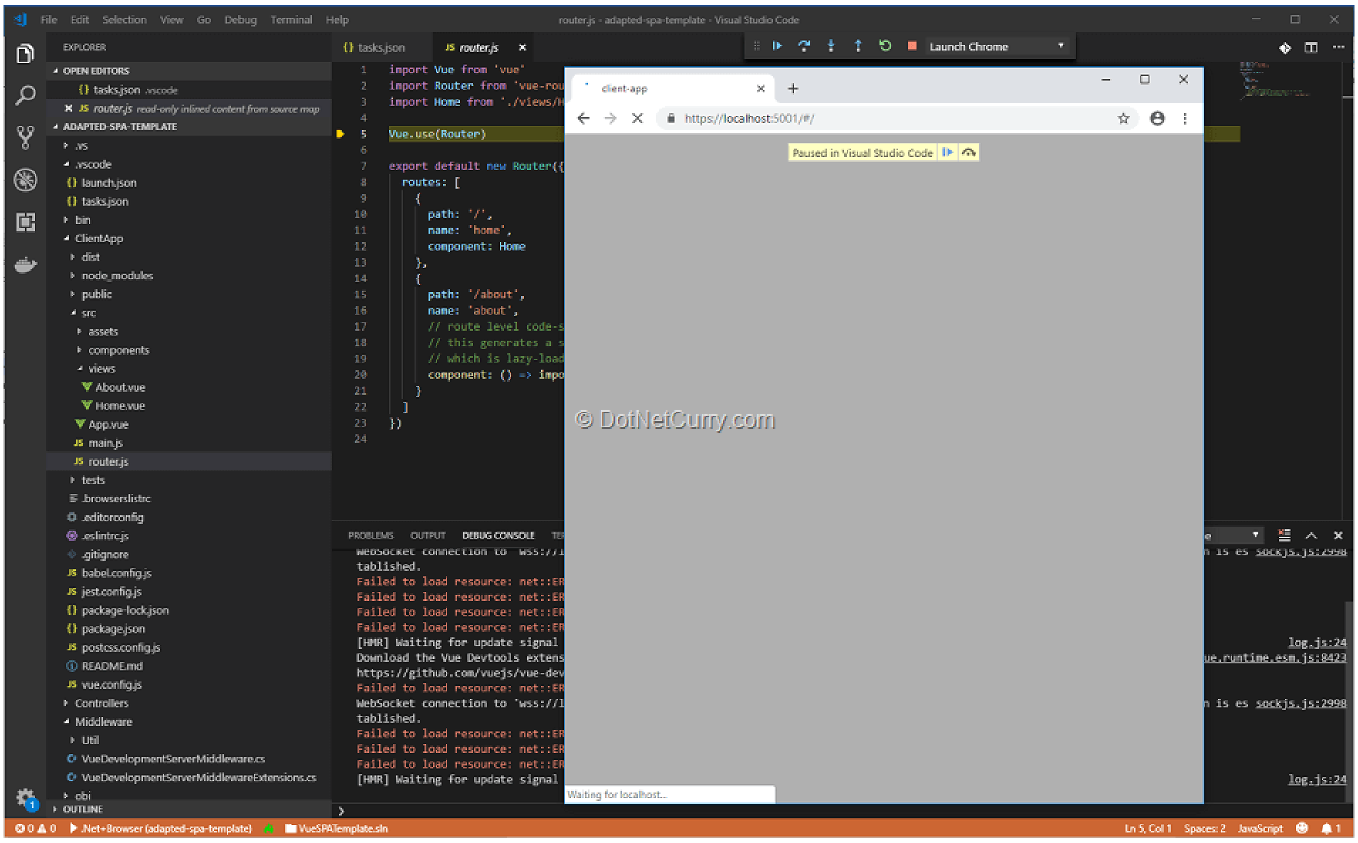
Debugging in Visual Studio Code requires you to install the C# extension (https://marketplace.visualstudio.com/items?itemName=ms-vscode.csharp) as well as the Debugger for Chrome extension (https://scotch.io/tutorials/debugging-javascript-in-google-chrome-and-visual-studio-code). When opening the project for the first time after adding the extensions, accept the suggestion to add common .NET Core tasks, which should add a tasks.json file.

Then add launch configurations to start the ASP.NET Core application (without starting the browser) and another to launch the Chrome debugger (this one starts the browser). Finally add a compound task to launch both.

The tasks should look similar to this:

```
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?
linkid=830387
  "version": "0.2.0",
  "compounds": [
    {
        "name": ".NET+Browser",
        "configurations": [ ".NET Core Launch (console)", "Launch Chrome" ]
    }
  ],
  "configurations": [
    {
      "type": "chrome",
      "request": "launch",
      "name": "Launch Chrome",
      "url": "http://localhost:5000",
      "webRoot": "${workspaceFolder}/ClientApp/src",
      "sourceMaps": true,
    },
    {
      "name": ".NET Core Launch (console)",
      "type": "coreclr",
      "request": "launch",
      "preLaunchTask": "build",
      "program":
"${workspaceFolder}/bin/Debug/netcoreapp3.0/VueSPATemplate.dll",
      "args": [],
      "cwd": "${workspaceFolder}",
      "stopAtEntry": false,
      "launchBrowser": {
        "enabled": false
      },
      "env": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
    },
    {
      "name": ".NET Core Launch (web)",
      "type": "coreclr",
      "request": "launch",
      "preLaunchTask": "build",
      "program":
"${workspaceFolder}/bin/Debug/netcoreapp3.0/VueSPATemplate.dll",
      "args": [],
      "cwd": "${workspaceFolder}",
      "stopAtEntry": false,
      "launchBrowser": {
        "enabled": true
      },
      "env": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "sourceFileMap": {
        "/Views": "${workspaceFolder}/Views"
      }
    },
    {
      "name": ".NET Core Attach",
      "type": "coreclr",
      "request": "attach",
      "processId": "${command:pickProcess}"
    }
  ]
}
```

You should now be able to launch the debugger selecting the ".NET+Browser" option, and setup breakpoints in both .NET and JavaScript code:



```
import Vue from 'vue'
import Router from 'vue-rou
import Home from './views/H

Vue.use(Router)

export default new Router({
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home
    },
    {
      path: '/about',
      name: 'about',
      // route level code-s
      // this generates a s
      // which is lazy-load
      component: () => impo
    }
  ]
})
```

```
WebSocket connection to 'wss://l
tablished.
Failed to load resource: net::ER
Failed to load resource: net::ER
Failed to load resource: net::ER
Failed to load resource: net::ER
[HMR] Waiting for update signal
Download the Vue Devtools extens
https://github.com/vuejs/vue-dev
Failed to load resource: net::ER
WebSocket connection to 'wss://l
tablished.
Failed to load resource: net::ER
Failed to load resource: net::ER
Failed to load resource: net::ER
[HMR] Waiting for update signal
```

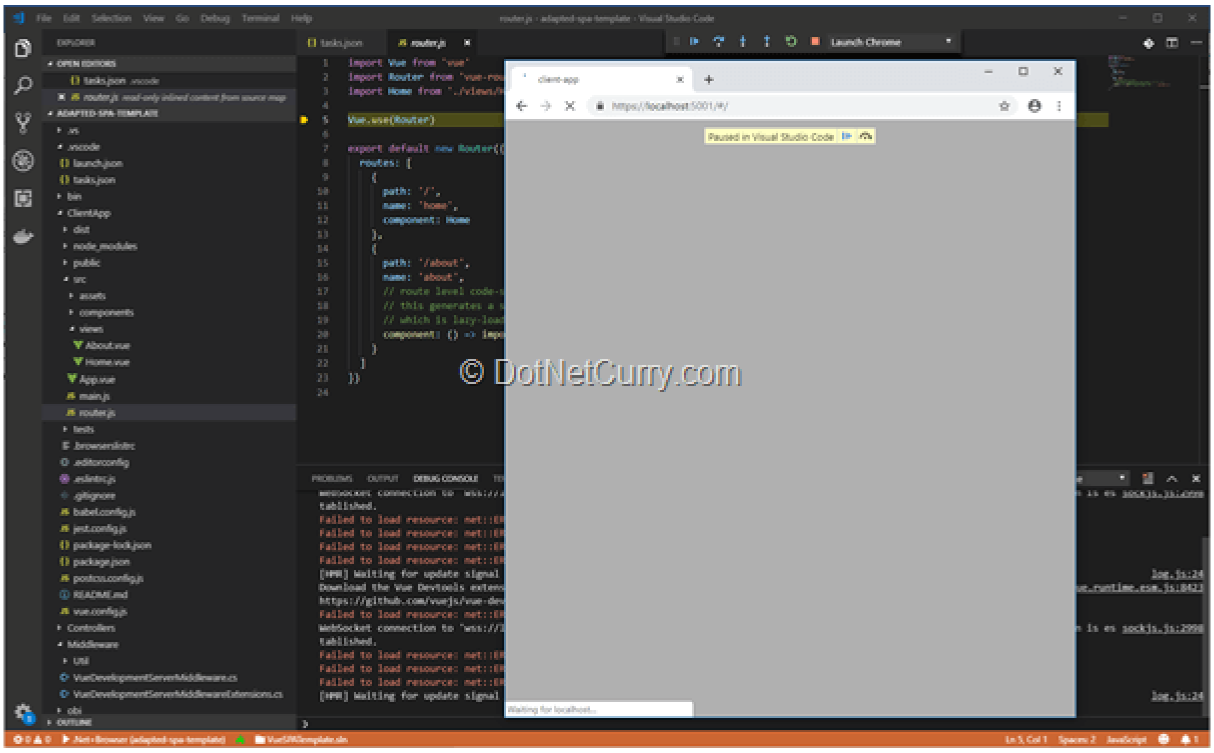


Figure 15, Debugging Vue's JavaScript code from Visual Studio Code

It is important to raise a big caveat when debugging JavaScript in Visual Studio (but not in Visual Studio Code). This is the fact that Visual Studio doesn't understand .vue files, forcing you to keep separate js/ts files if you really want to debug Vue components from Visual Studio.

As said this limitation doesn't apply to Visual Studio Code as long as you install the Vetur (https://github.com/vuejs/vetur) extension.

Although in my opinion you are better off debugging in Chrome with the Vue dev tools extension, using the right tool for the job!

Step 4. Including the built Vue.js bundles during publish

When the Vue.js application bundles are generated with npm run build, these are generated into the **ClientApp/dist** folder. This is different from where the React template expects the generated bundles, so we will need to adapt the template for the bundles to be included in the published output and for the ASP.NET Core server to be able to serve them when deployed.

First update the RootPath property in the call to AddSpaStaticFiles found in the ConfigureServices method of the Startup class:

```
// In production, the Vue files will be served from this directory
services.AddSpaStaticFiles(configuration =>
{
    configuration.RootPath = "ClientApp/dist";
});
```

Next update the csproj files, so the section that copies the generated bundles into the output looks in the ClientApp/dist folder:

```
<DistFiles Include="$(SpaRoot)dist\**" />
```

That's all, you will now correctly generate the Vue.js bundles and include them into the published output inside the ClientApp/dist folder, which the ASP.NET Core server is configured to serve as static files.

Using the community published template

As mentioned at the beginning of the section, Software Ateliers has already published a template (https://github.com/SoftwareAteliers/asp-net-core-vue-starter) following this approach. You can just install their template and use it when generating a new project, rather than manually going through the steps described above! (Although it is good for you to understand what's going on under the covers.)

The only thing you might want to add is the **vue.config.js** file with the source map rewrite that enables debugging from Visual Studio and Visual Studio Code. Although in my opinion, and as already mentioned earlier, you are better off debugging in Chrome with the Vue.js devtools extension, using the right tool for the job!

The next section discusses a variant of this template that inverts the proxying role between the ASP.NET Core and Vue.js development servers.

A different SPA template approach

All of the official SPA templates plus the Vue.js SPA community template has taken the same approach. The ASP.NET Core application acts as the main server and proxies requests to a React/Angular/Vue.js development server launched during application startup.

We can easily modify the middleware we created in the previous section so we invert the roles of the servers.

We can keep launching everything when starting the ASP.NET Core application, launching the npm script during its Startup. However, we can then redirect the browser to the Vue.js development server and have this one proxying API requests to the ASP.NET Core application.

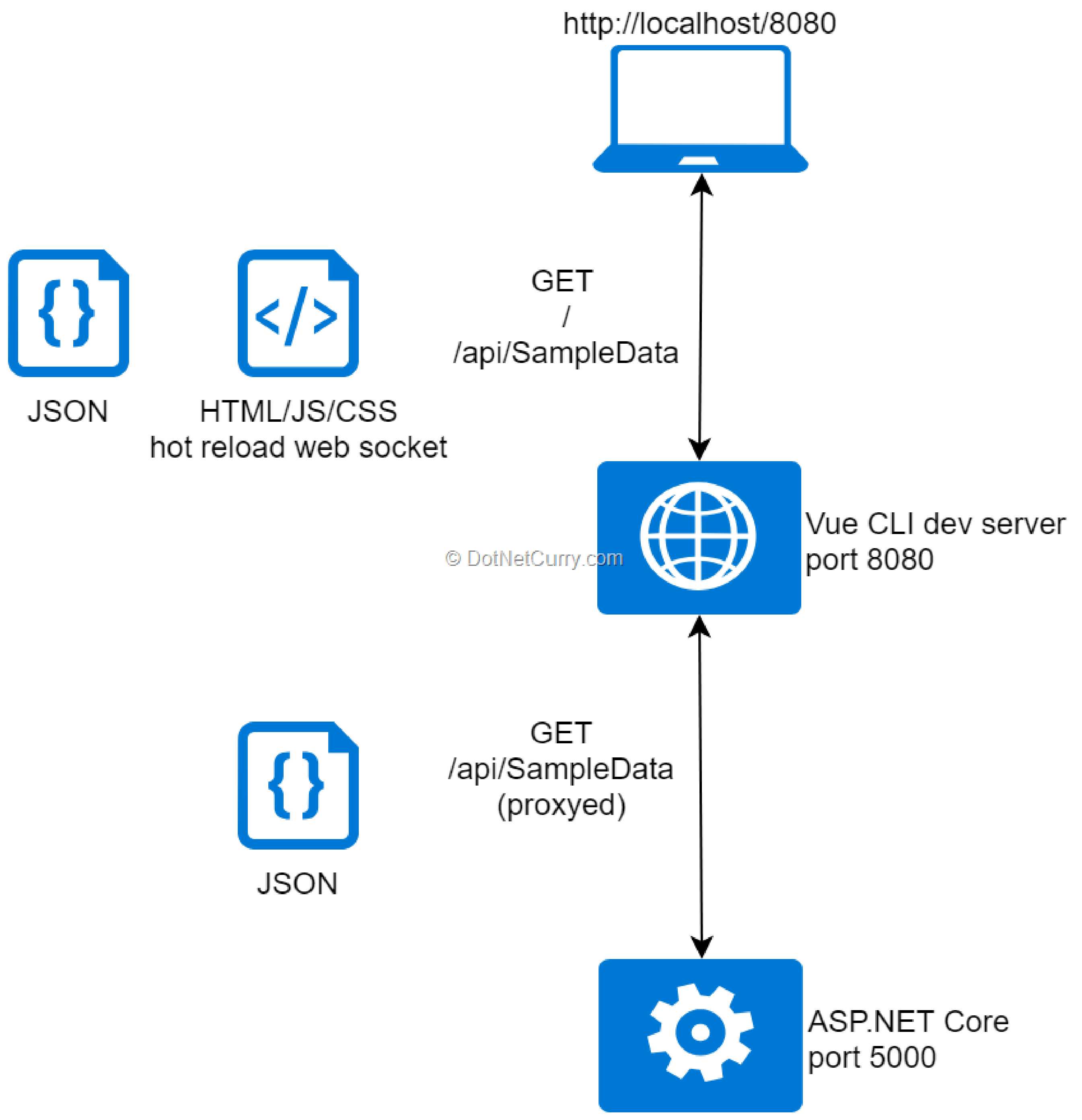


Figure 16, Inverting the server roles during development. Vue.js development server now proxies to ASP.NET Core

This will align every server closely to the role they actually play in the system. Keeping the Vue.js development server closer to the browser is a great idea for avoiding issues with its hot reload functionality.

Instead of ASP.NET Core setting up a SPA proxy to the Vue.js development server, we will setup the Vue.js development server proxy (https://cli.vuejs.org/config/#devserver-proxy) to the ASP.NET Core application.

We will also request the browser to initially load the root page "/" from Vue.js development server.

This way the browser will initially load the Vue.js home page from the Vue.js development server, while any requests for an /api endpoint will be sent to the ASP.NET Core server by the Vue.js development server proxy.

We will use the previous template as the starting point, since most of the changes we made to the official React template are still needed. If you are starting from scratch, it will be useful to go through the **Manually adapting the template** section of the earlier template, bearing in mind most of the changes will be located in the VueDevelopmentServerMiddleware.

Feel free to **download the code** from GitHub (https://github.com/DaniJG/ASPCoreVueCLITemplates) if that helps.

Setting up the Vue development server as the main server

The first change we will make to the previous template is located in the **vue.config.js** file. We will use its dev server proxy (https://cli.vuejs.org/config/#devserver-proxy) option to send any requests towards the ASP.NET Core server that it cannot solve itself.

Rather than hardcoding a URL, we will read its value from an environment variable ASPNET_URL that will be set by the VueDevelopmentServerMiddleware before starting the Vue dev server:

```
module.exports = {
  configureWebpack: {
    // The URL where the .NET Core app will be listening.
    // Read the ASPNET_URL environment variable, injected by
VueDevelopmentServerMiddleware
    devServer: {
      // When running in IISExpress, the env variable won't be provided.
      // Hardcode a fallback here based on your launchSettings.json
      proxy: process.env.ASPNET_URL || 'https://localhost:44345'
    },
    // … devtool and output are same as in the earlier template …
  }
}
```

It is important to note the hardcoded fallback in case you run the project with IISExpress. The startup code does not know the public URL used by IISExpress and can't inject the environment variable, so you need to hardcode the value here based on the contents of your ***launchSettings.json****!*

Next, we will modify the StartVueDevServerAsync function of the VueDevelopmentServerMiddleware so it injects the ASPNET_URL environment variable. The value can be found by getting IServerAddressesFeature service:

```
private static async Task<int> StartVueDevServerAsync(
    IApplicationBuilder appBuilder,
    string sourcePath,
    string npmScriptName,
    ILogger logger)
{
    var portNumber = TcpPortFinder.FindAvailablePort();
    logger.LogInformation($"Starting Vue dev server on port {portNumber}...");

    // Inject address of .NET app as the ASPNET_URL env variable
    // which will be read it in vue.config.js from process.env
    // NOTE: When running with IISExpress this will be empty,
    // so you need to hardcode the URL in IISExpress as a fallback
    var addresses = appBuilder.ServerFeatures.Get<IServerAddressesFeature>
().Addresses;
    var envVars = new Dictionary<string, string>
    {
        { "ASPNET_URL", addresses.Count > 0 ? addresses.First() : "" },
    };
    var npmScriptRunner = new NpmScriptRunner(
        sourcePath, npmScriptName, $"--port {portNumber} --host localhost",
envVars);
    npmScriptRunner.AttachToLogger(logger);
    // the rest of the method remains unchanged, waiting to see "DONE"
    // in the script output and returning the portNumber

}
```

The biggest change will be made in the Attach method. Instead of calling SpaProxyingExtensions.UseProxyToSpaDevelopmentServer we will add middleware to:

- Wait for the Vue.js development server to be started

- Redirect requests to the root page "/" to the root of the Vue.js development server

The Attach method will then look like:

```
public static void Attach(
    IApplicationBuilder appBuilder,
    string sourcePath,
    string npmScriptName)
{
    if (string.IsNullOrEmpty(sourcePath))
    {
        throw new ArgumentException("Cannot be null or empty",
nameof(sourcePath));
    }

    if (string.IsNullOrEmpty(npmScriptName))
    {
        throw new ArgumentException("Cannot be null or empty",
nameof(npmScriptName));
    }

    var logger = LoggerFinder.GetOrCreateLogger(appBuilder, LogCategoryName);

    // Start Vue development server
    var portTask = StartVueDevServerAsync(appBuilder, sourcePath,
npmScriptName, logger);
    var targetUriTask = portTask.ContinueWith(
        task => new UriBuilder("http", "localhost",
task.Result).Uri);

    // Add middleware that waits for the Vue development server to start
    // before calling the next middleware on the chain
    appBuilder.Use(async (context, next) =>
    {
        // On each request gets its own timeout. That way, even if
        // the first request times out, subsequent requests could still work.
        var timeout = TimeSpan.FromSeconds(30);
        await targetUriTask.WithTimeout(timeout,
            $"The vue development server did not start listening for requests
" +
            $"within the timeout period of {timeout.Seconds} seconds. " +
            $"Check the log output for error information.");

        await next();
    });

    // Redirect all requests for root towards the Vue development server,
    // using the resolved targetUriTask
    appBuilder.Use(async (context, next) =>
    {
        if (context.Request.Path == "/")
        {
            var devServerUri = await targetUriTask;
            context.Response.Redirect(devServerUri.ToString());
        } else
        {
            await next();
        }
    });
}
```

Now we need to update the UseVueDevelopmentServer method of the extension class. We no longer need to pass an ISpaBuilder, just pass both the IApplicationBuilder and sourcePath as parameters:

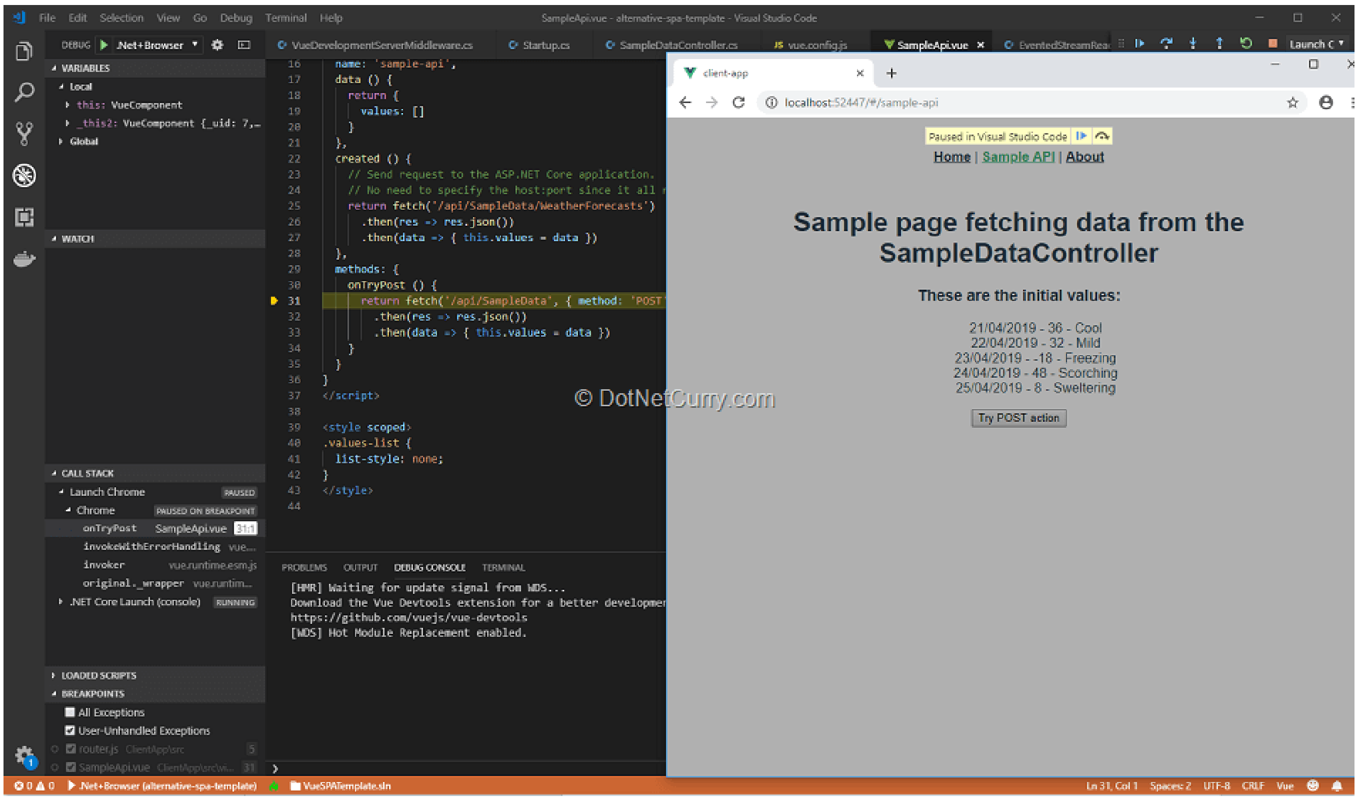
```
public static void UseVueDevelopmentServer(
     this IApplicationBuilder appBuilder,
     string sourcePath = "ClientApp",
     string npmScript = "serve")
{
     if (appBuilder == null)
     {
          throw new ArgumentNullException(nameof(appBuilder));
     }

     VueDevelopmentServerMiddleware.Attach(appBuilder, sourcePath, npmScript);
}
```

Finally we replace the call to app.UseSpa at the end of the Configure method of the Startup class with this simple call:

```
if (env.IsDevelopment())
{
     app.UseVueDevelopmentServer();
}
```

Once you are done with these changes, you should now be able to start and debug the application the same way as before, including JavaScript code. Make sure you completed steps 1, 3 and 4 of the **Manually adapting the template** section since they are still relevant!



File Edit Selection View Go Debug Terminal Help
SampleApi.vue - alternative-spa-template - Visual Studio Code
DEBUG .Net+Browser
VueDevelopmentServerMiddleware.cs
Startup.cs
SampleDataController.cs
vue.config.js
SampleApi.vue
EventedStreamRea
Launch C
VARIABLES
Local
this: VueComponent
_this2: VueComponent {_uid: 7,…
Global
WATCH
CALL STACK
Launch Chrome PAUSED
Chrome PAUSED ON BREAKPOINT
onTryPost SampleApi.vue 31:1
invokeWithErrorHandling vue…
invoker vue.runtime.esm.js
original._wrapper vue.runtim…
.NET Core Launch (console) RUNNING
LOADED SCRIPTS
BREAKPOINTS
All Exceptions
User-Unhandled Exceptions
router.js ClientApp\src 5
SampleApi.vue ClientApp\src\vi… 31
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
0 0 .Net+Browser (alternative-spa-template) VueSPATemplate.sln
Ln 31, Col 1 Spaces: 2 UTF-8 CRLF Vue
client-app
localhost:52447/#/sample-api
Paused in Visual Studio Code
Home | Sample API | About
Try POST action
© DotNetCurry.com

```
  name: 'sample-api',
  data () {
    return {
      values: []
    }
  },
  created () {
    // Send request to the ASP.NET Core application.
    // No need to specify the host:port since it all r
    return fetch('/api/SampleData/WeatherForecasts')
      .then(res => res.json())
      .then(data => { this.values = data })
  },
  methods: {
    onTryPost () {
      return fetch('/api/SampleData', { method: 'POST'
        .then(res => res.json())
        .then(data => { this.values = data })
    }
  }
}
</script>

<style scoped>
.values-list {
  list-style: none;
}
</style>
```

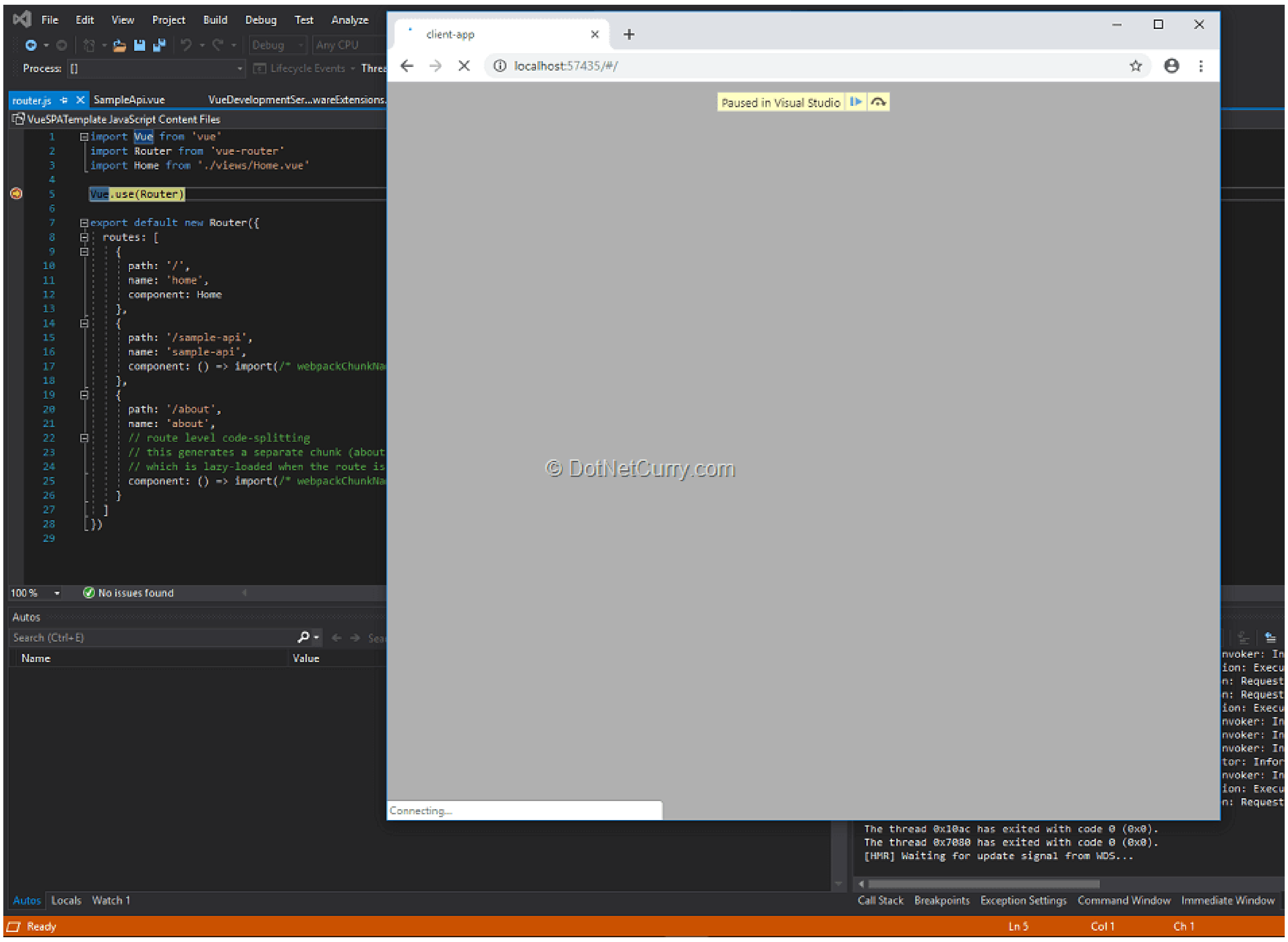
```
[HMR] Waiting for update signal from WDS...
Download the Vue Devtools extension for a better developmen
https://github.com/vuejs/vue-devtools
[WDS] Hot Module Replacement enabled.
```

Sample page fetching data from the SampleDataController

These are the initial values:

21/04/2019 - 36 - Cool
22/04/2019 - 32 - Mild
23/04/2019 - -18 - Freezing
24/04/2019 - 48 - Scorching
25/04/2019 - 8 - Sweltering

Figure 17, Debugging with the alternate template from Visual Studio Code


```
import Vue from 'vue'
import Router from 'vue-router'
import Home from './views/Home.vue'

Vue.use(Router)

export default new Router({
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home
    },
    {
      path: '/sample-api',
      name: 'sample-api',
      component: () => import(/* webpackChunkNa
    },
    {
      path: '/about',
      name: 'about',
      // route level code-splitting
      // this generates a separate chunk (about
      // which is lazy-loaded when the route is
      component: () => import(/* webpackChunkNa
    }
  ]
})
```

```
The thread 0x10ac has exited with code 0 (0x0).
The thread 0x7080 has exited with code 0 (0x0).
[HMR] Waiting for update signal from WDS...
```

Figure 18, Debugging the alternate template with Visual Studio

Conclusion

Vue.js is one of the fastest growing and most loved web frameworks. Just to name a couple of recent events, it was chosen both as the second most loved and second most wanted web framework in the 2019 Stack Overflow survey (https://insights.stackoverflow.com/survey/2019?utm_source=so-owned&utm_medium=blog&utm_campaign=dev-survey-2019&utm_content=launch-blog#technology-_-most-loved-dreaded-and-wanted-web-frameworks), and surpassed React in the number of GitHub stars during 2018.

In my personal experience, most of the developers who have worked with Vue.js find it very attractive, intuitive and a pleasure to work with. Tooling like the Vue CLI (https://cli.vuejs.org), the Chrome Dev tools (https://github.com/vuejs/vue-devtools) or the VS Code plugin Vetur (https://github.com/vuejs/vetur) make it even easier to work with it and develop web applications, while its plugin model and library ecosystem, makes it a very extensible and adaptable framework.

It is a shame that it isn't one of the frameworks with an official SPA template out of the box in ASP.NET Core. As of today, only React and Angular are officially supported.

While things might change in the future, for now it is up to the community to fill the gap.

In this tutorial, we have seen several approaches in which you can **integrate Vue.js with ASP.NET Core**, all of them using Vue.js applications generated by the Vue CLI.

While the first approach is the obvious one of treating the Vue.js and ASP.NET Core applications as separate projects, we have also seen two other approaches in which they can be more tightly integrated for those who prefer that approach. One of these tightly integrated ways is a straight port of the React template and is already published (https://github.com/SoftwareAteliers/asp-net-core-vue-starter) as a NuGet package by Software Ateliers.

The lack of an official template is unfortunate, but I hope this article gives you enough information to fill in the missing piece!

Download the entire source code from GitHub (https://github.com/DaniJG/ASPCoreVueCLITemplates).

***This article was technically reviewed by Damir Arh** (https://www.dotnetcurry.com/author/damir-arh).*

This article has been editorially reviewed by Suprotim Agarwal. (https://www.dotnetcurry.com/author/suprotim-agarwal)

WHAT OTHERS ARE READING!

Role Based Security in an ASP.NET Core Application (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1505)

Zero Downtime Deployment for ASP.NET applications (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1503)

Developing Web Applications in .NET (Different Approaches and Current State) (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1501)

Using ASP.NET Core SignalR with Vue.js (to create a mini Stack Overflow rip-off) (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1480)

Facade Design Pattern: Still relevant in ASP.NET Core? (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1468)

End-to-End (E2E) Testing of ASP.NET Core Applications using Selenium and Nightwatch.js (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1433)

Lookup and Verify Global Name, Address, Phone, IP Location. (http://www.dotnetcurry.org/r/dnc-melissa-jul18)

Was this article worth reading? Share it with fellow developers too. Thanks!

Share on Facebook (https://facebook.com/sharer/sharer.php?u=https%3A%2F%2Fwww.dotnetcurry.com%2Faspnet-core%2F1500%2Faspnet-core-vuejs-template)

Share on Twitter (https://twitter.com/intent/tweet/?text=ASP.NET%20Core%20Vue%20CLI%20Templates%20%7C%20DotNetCurry&url=https%3A%2F%2Fwww.dotnetcurry.com%2Faspnet-core%2F1500%2Faspnet-core-vuejs-template)

Share on LinkedIn (https://www.linkedin.com/shareArticle?mini=true&amp;url=https%3A%2F%2Fwww.dotnetcurry.com%2Faspnet-core%2F1500%2Faspnet-core-vuejs-template&title=ASP.NET%20Core%20Vue%20CLI%20Templates%20%7C%20DotNetCurry)

Share on Google+ (https://plus.google.com/share?url=https%3A%2F%2Fwww.dotnetcurry.com%2Faspnet-core%2F1500%2Faspnet-core-vuejs-template)

AUTHOR

Daniel Jimenez Garcia is a passionate software developer with 10+ years of experience. He started as a Microsoft developer and learned to love C# in general and ASP.NET MVC in particular. In the latter half of his career he worked on a broader set of technologies and platforms while these days he is particularly interested in .Net Core and Node.js. He is always looking for better practices and can be seen answering questions on Stack Overflow.

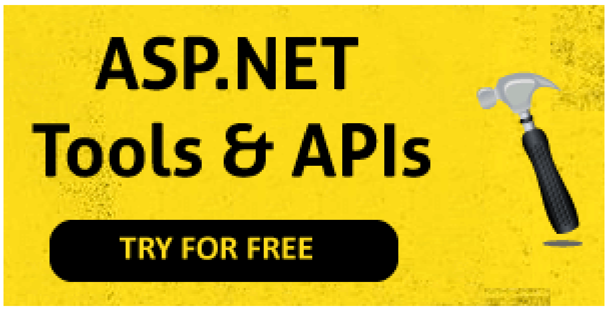
FEEDBACK - LEAVE US SOME ADULATION, CRITICISM AND EVERYTHING IN BETWEEN!

Click here to post your Comments

FEATURED TOOLS

(http://www.dotnetcurry.org/r/dnc-arc-300-150)

(http://www.dotnetcurry.org/r/dnc-csharpbk-web-300x150)



(http://www.dotnetcurry.net/s/dnc-products)

CATEGORIES

- .NET Web
- .NET Framework, Visual Studio and C#
- Patterns & Practices
- Cloud and Mobile
- JavaScript
- .NET Desktop
- Interview Questions & Product Reviews

JOIN OUR COMMUNITY

51,659
fans
(https://www.facebook.com/dotnetcurry)

7,923
followers
(https://www.twitter.com/dotnetcurry)

106,812
subscribers
(https://www.dotnetcurry.com/magazine/)

POPULAR ARTICLES

New C# 8 Features in Visual Studio 2019 (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1489)

Parallel workflow with the .NET Task Parallel Library (TPL) DataFlow (C#) (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1483)

Azure DevOps to build and deploy ReactJS App (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1488)

Use REST APIs to access Azure DevOps (formerly VSTS) (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1485)

React.js - Parent Child Component Communication and Event Handling (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1484)

Global State in C# Applications - Part 1 (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1491)

Working with Barcodes in Xamarin.Forms (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1490)

The History of ASP.NET – Part I (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1492)

Developing Web Applications in .NET (Different Approaches and Current State) (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1501)

The History of ASP.NET – Part II (Covers ASP.NET MVC) (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1493)

Reactive Azure Service Bus Messaging with Azure Event Grid (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1498)

Using Azure DevOps for Build and Deployment of NodeJS application (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1486)

Shipping Pseudocode to Production (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1497)

ASP.NET Core Vue CLI Templates (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1500)

.NET Core Global Tools - (What are Global Tools, How to Create and Use them) (http://www.dotnetcurry.com/ShowArticle.aspx?ID=1495)

C# .NET BOOK

STRENGTHEN YOUR
C# CONCEPTS
COVERS C# v6, 7, 8 AND .NET Core
THE ABSOLUTELY AWESOME
BOOK ON
C#
AND
.NET
CRACK YOUR NEXT
.NET INTERVIEW
.NET Framework, CLR, .NET Core, C# (v6, 7, 8) Parallel/Async programming, Generics and Collections, LINQ, and much more...
ORDER NOW
dotnetcurry.com

(http://www.dotnetcurry.org/r/dnc-csharpbk-web-300x600x2-green)

Tags

- ASP.NET MVC (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/ASPNET-MVC)
- ASP.NET CORE (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/ASPNET-CORE)
- ASP.NET (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/ASPNET)
- SHAREPOINT (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/SHAREPOINT)
- DESIGN PATTERNS (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/PATTERNS-PRACTICES)
- C# (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/CSHARP)
- LINQ (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/LINQ)
- WPF (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/WPF)
- WCF (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/WCF)
- VISUAL STUDIO (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/VISUALSTUDIO)

VSTS & TFS (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/VSTS-TFS)

AZURE (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/WINDOWS-AZURE)

ENTITY FRAMEWORK (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/ENTITYFRAMEWORK)

ANGULAR.JS (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/ANGULARJS)

REACT.JS (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/REACTJS)

JQUERY (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/JQUERY-ASPNET)

JAVASCRIPT (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/HTML5-JAVASCRIPT)

HTML5 (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/HTML5-JAVASCRIPT)

.NET CORE (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/DOTNET-STANDARD-CORE)

.NET FRAMEWORK (HTTPS://WWW.DOTNETCURRY.COM/TUTORIALS/DOTNETFRAMEWORK)

(https://www.dotnetcurry.com)

SERVER-SIDE

ASP.NET (https://www.dotnetcurry.com/tutorials/aspnet)

ASP.NET Core (https://www.dotnetcurry.com/tutorials/aspnet-core)

ASP.NET MVC (https://www.dotnetcurry.com/tutorials/aspnet-mvc)

WCF (https://www.dotnetcurry.com/tutorials/wcf)

SharePoint (https://www.dotnetcurry.com/tutorials/sharepoint)

CLIENT-SIDE

Angular.js (https://www.dotnetcurry.com/tutorials/angularjs)

React.js (https://www.dotnetcurry.com/tutorials/reactjs)

jQuery (https://www.dotnetcurry.com/tutorials/jquery-aspnet)

Backbone.js (https://www.dotnetcurry.com/tutorials/backbonejs)

HTML5 (https://www.dotnetcurry.com/tutorials/html5-javascript)

CSS (https://www.dotnetcurry.com/tutorials/bootstrap-css)

.NET

C# (https://www.dotnetcurry.com/tutorials/csharp)

Visual Studio (https://www.dotnetcurry.com/tutorials/visualstudio)

VSTS & TFS (https://www.dotnetcurry.com/tutorials/vsts-tfs)

LINQ (https://www.dotnetcurry.com/tutorials/linq)

Entity Framework (https://www.dotnetcurry.com/tutorials/entityframework)

.NET Framework (https://www.dotnetcurry.com/tutorials/dotnetframework)

.NET Standard & .NET Core (https://www.dotnetcurry.com/tutorials/dotnet-standard-core)

WPF (https://www.dotnetcurry.com/tutorials/wpf)

WinForms (https://www.dotnetcurry.com/tutorials/winforms)

CLOUD AND MOBILE

Microsoft Azure (https://www.dotnetcurry.com/tutorials/windows-azure)

DevOps (https://www.dotnetcurry.com/tutorials/devops)

Xamarin (https://www.dotnetcurry.com/tutorials/xamarin)

Powershell (https://www.dotnetcurry.com/tutorials/powershell)

Machine Learning & AI (https://www.dotnetcurry.com/tutorials/machine-learning-ai)

UWP & Windows Store (https://www.dotnetcurry.com/tutorials/windows-store)

Windows Phone (https://www.dotnetcurry.com/tutorials/windowsphone)

SKILL UP

Design Patterns (https://www.dotnetcurry.com/tutorials/patterns-practices)

Software Gardening (https://www.dotnetcurry.com/tutorials/software-gardening)

.NET Interview Q&A (https://www.dotnetcurry.com/tutorials/dotnetinterview)

Magazines (https://www.dotnetcurry.com/magazine/)

Books (http://www.jquerycookbook.com/)

Product Reviews (https://www.dotnetcurry.com/tutorials/product-articles-review)

FOLLOW US

- Facebook (https://www.facebook.com/dotnetcurry)
- Twitter (https://www.twitter.com/dotnetcurry)
- Github (https://github.com/dotnetcurry)

Contact Us (https://www.dotnetcurry.com/Contact.aspx) Write For Us (https://www.dotnetcurry.com/WriteForUs.aspx)

Privacy (https://www.dotnetcurry.com/PrivacyPolicy.aspx)