Getting Started with .NET Core, Docker, and RabbitMQ — Part 1

Trimble MAPS Engineering Blog Follow
Jul 12, 2019 · 6 min read

by Matthew Harper

The goal of this series is to introduce some of the technologies we use at Trimble MAPS. We're going to build a web application that will accept HTTP messages from a client, publish them to a RabbitMQ message queue, then consume those messages from the queue. We'll also use Docker to run each application in a container, and Docker-Compose to orchestrate the launch of all of our containers. If any of those terms are unfamiliar, don't worry, each one will be explained as we approach it. The first part of this tutorial will focus on .NET Core.

Part 1: .NET Core (source code can be found on Github)

.NET Core is an open source software development framework, and effectively the successor to Microsoft's .NET Framework. It's been steadily growing in popularity among developers thanks to a few significant differences from previous iterations of .NET. Foremost among these is cross-platform support — applications can be built on Windows, Linux, or MacOS, and once built, will run on any of those platforms. The lightweight nature of .NET Core also make it an ideal choice for containerization. Combining those two attributes leads to a common industry use case — building apps on Windows, but running them in Linux containers to improve performance and reduce operating costs.

Let's create two apps: a simple WebAPI to accept HTTP POST requests, and a console app to send those messages to our WebAPI. The only prerequisites for this are Visual Studio Code and the .NET Core 2.2 SDK.

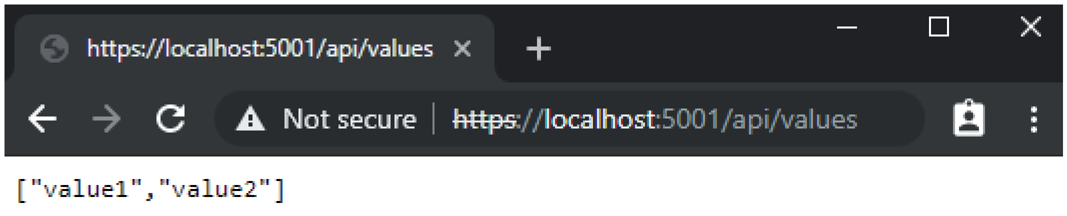
First, create a new folder to serve as the root directory of our project, and open a command prompt pointing to that directory. We'll use the dotnet CLI tool to create our solution and skeleton apps for our projects. Run the following commands:

```
dotnet new webApi -o "publisher_api"
dotnet new sln
dotnet sln add publisher_api/publisher_api.csproj
```

cmd hosted with ♡ by GitHub view raw

The first line creates a new ASP.Net Core WebAPI project in a sub-directory called publisher_api. The next two commands create a solution in our root folder, and add the publisher_api project to that solution.

Now, we already have a working web service! We can test it out by executing "dotnet run" from the publisher_api directory and then visiting https://localhost:5001/api/values in our web browser.


https://localhost:5001/api/values
Not secure
~~https~~://localhost:5001/api/values

```
["value1","value2"]
```

Output from our publisher_api project

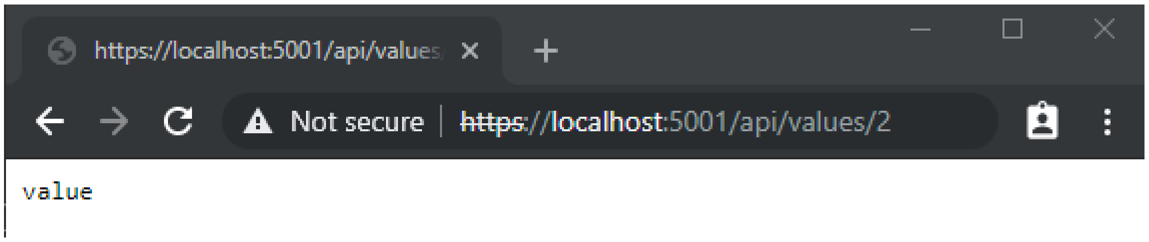
Let's open our solution in Visual Studio Code and see what we've created. In our command prompt, enter "code ." from the root directory to launch the IDE and browse our project.

Startup.cs and Program.cs contain the boilerplate necessary to run our small web service. They are important, but their details are beyond the scope of this tutorial. Instead, let's look inside the Controllers folder and examine ValuesController.cs. In ASP.Net Core, controllers are responsible for handling incoming HTTP requests. When we entered localhost:5001/api/values in our browser, we sent an HTTP GET request to the Values controller. The response we saw in the browser is returned from the designated GET method in that controller:

```csharp
public class ValuesController : ControllerBase
{
    // GET api/values
    [HttpGet]
    public ActionResult<IEnumerable<string>> Get()
    {
        return new string[] { "value1", "value2" };
    }
```

ValuesController.cs hosted with ♡ by GitHub view raw

Just below the default Get method, you'll see another Get function defined, but this one takes an int as a parameter. If we enter https://localhost:5001/api/values/2 in our browser, our request will be routed to that function (and '2' will be the value of the 'id' parameter). We'll also see a different response in the browser.


https://localhost:5001/api/values
Not secure
https://localhost:5001/api/values/2

```
value
```

• • •

Let's add another project to our solution — the console app that will post messages to our web service. The dotnet CLI commands to create the console app and add it to our existing solution are below. Make sure you return to the root directory of our solution before you execute them.

```
dotnet new console -o "worker"
dotnet sln add worker/worker.csproj
```

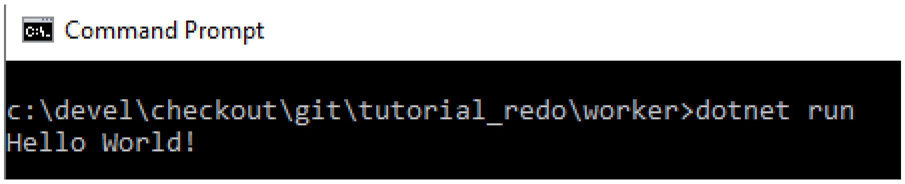
cmd hosted with ♡ by GitHub view raw

To test the console app, navigate to the worker directory in your command prompt, and execute the following:

```
dotnet run
```

cmd hosted with ♡ by GitHub view raw

You'll see the program output "Hello World."



```
c:\devel\checkout\git\tutorial_redo\worker>dotnet run
Hello World!
```

Our console app will depend on a third party library to assist with JSON serialization, so let's add that now using the dotnet CLI command below, again executed from the worker directory:

```
dotnet add package Newtonsoft.Json
```

cmd hosted with ♡ by GitHub view raw

This will modify the worker.csproj file to include the package reference:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="12.0.2" />
  </ItemGroup>

</Project>
```

worker.csproj hosted with ♡ by GitHub view raw

Tab over to VSCode, and you will be prompted to ‘restore’. The restore will download the new dependency.

. . .

We have one more bit of housekeeping before we get to the code. Because we have multiple projects in our solution, we need to tweak to the build and launch configurations for our workflow.

In VS Code, open tasks.json; this is where our build task is defined. Modify it to build the entire solution by changing the argument we pass to ‘build’ to from a specific project to “${workspaceFolder}”. The result should look like this:

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "build",
            "command": "dotnet",
            "type": "process",
            "args": [
                "build",
                "${workspaceFolder}"
            ],
            "problemMatcher": "$msCompile"
        }
    ]
}
```

tasks.json hosted with ♡ by GitHub view raw

Next, open launch.json; this is where our debug and run configurations are stored. We need to add a launch configuration for the console app. There is a wizard in VS Code to assist with this, or you can simply open launch.json and paste the below snippet within the array of configurations:

```
{
            "name": ".NET Core Launch (console)",
            "type": "coreclr",
            "request": "launch",
            "preLaunchTask": "build",
            "program": "${workspaceFolder}/worker/bin/Debug/netcoreapp2.2/worker.dll",
            "args": [],
            "cwd": "${workspaceFolder}/worker",
            "stopAtEntry": false,
            "console": "internalConsole"
        },
```

launch.json hosted with ♡ by GitHub view raw

OK — time to write code!

· · ·

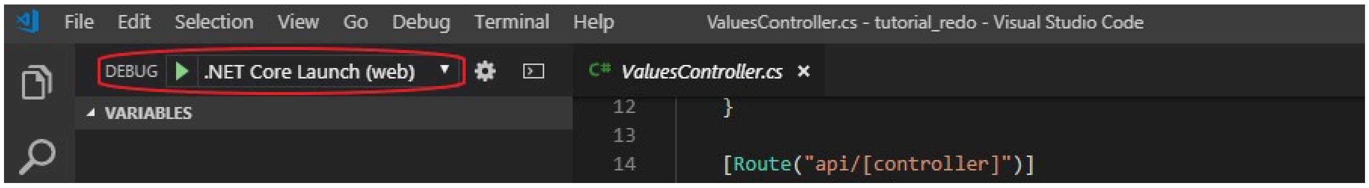
Let's go back to ValuesController.cs. Besides the GET methods we examined earlier, you'll find methods to handle POST, PUT, and DELETE requests. Replace the existing POST method with the code below:

```
[HttpPost]
public IActionResult Post([FromBody] string payload)
{
  return Ok("{\"success\": \"true\"}");
}
```

ValuesController.cs hosted with ♡ by GitHub view raw

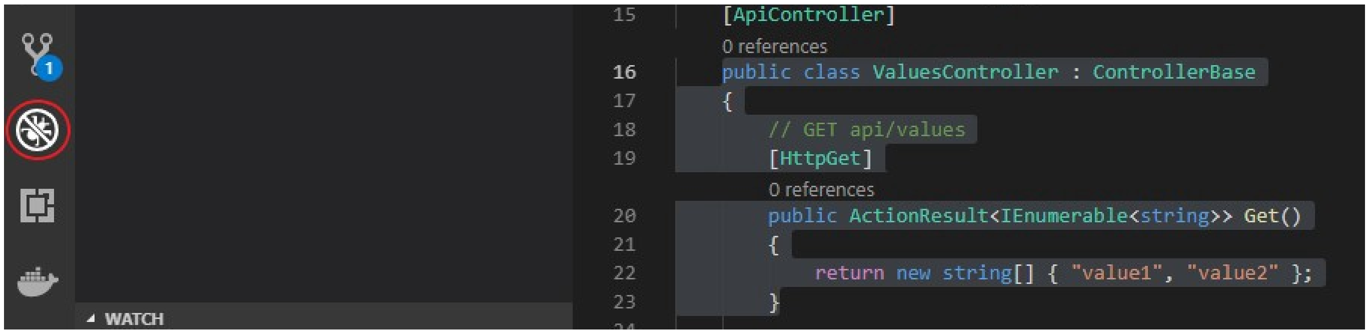
Let's walk through the differences. We've changed the return type to IActionResult, which allows our response to represent various HTTP status codes (200, 403, etc). We also modified the function to accept a string from the body of the HTTP message. For now, our method will always return a 200 (OK) along with a simple JSON response.

Let's test this out. Switch to the debug pane in VS Code (Ctrl+Shift+D), select the .NET Core Launch (Web) configuration, and start debugging.



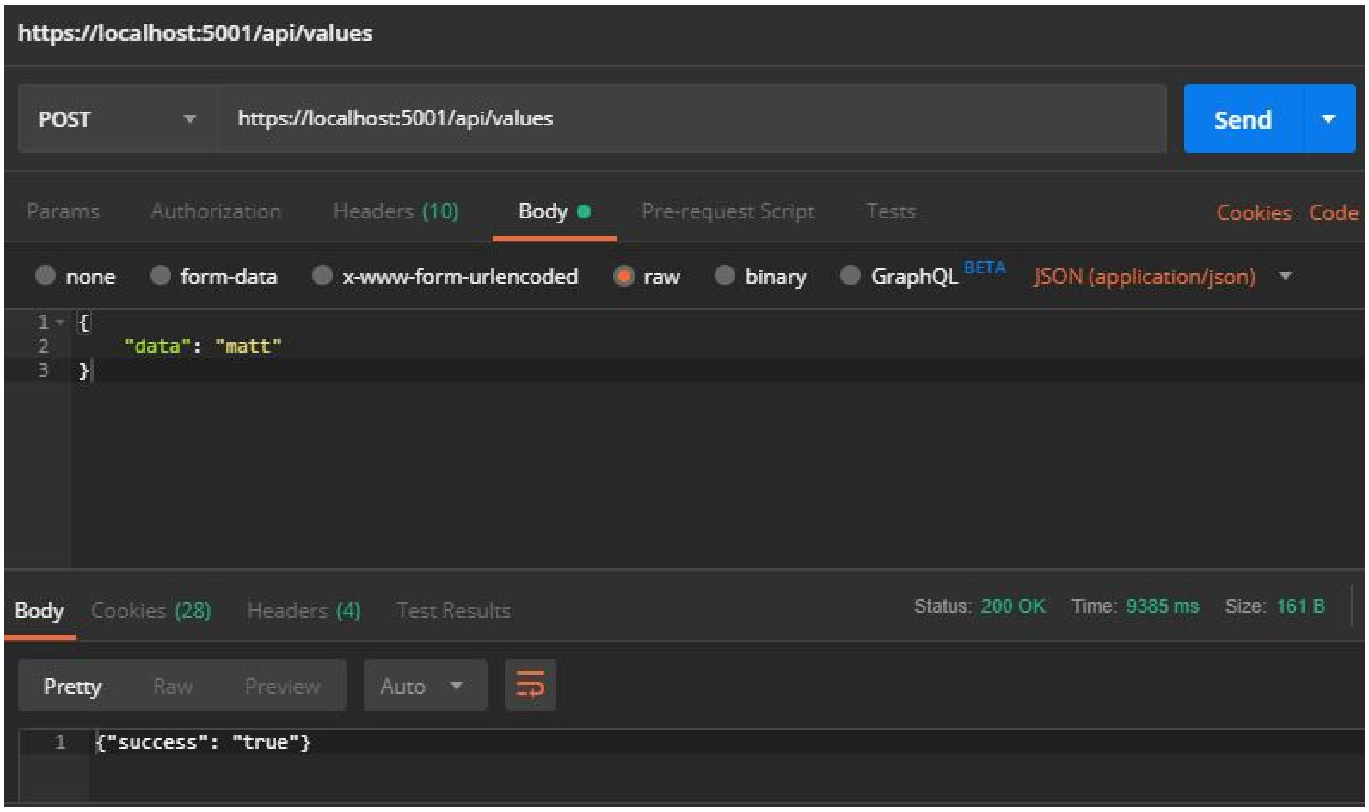
```
    }

    [Route("api/[controller]")]
```



```
[ApiController]
0 references
public class ValuesController : ControllerBase
{
    // GET api/values
    [HttpGet]
    0 references
    public ActionResult<IEnumerable<string>> Get()
    {
        return new string[] { "value1", "value2" };
    }
```

Place a break point in the Post method, and then fire up Postman (or your API testing tool of choice). Send a POST to https://localhost:5001/api/values, and we should hit our break point.



```
{
    "data": "matt"
}
```

Status: 200 OK Time: 9385 ms Size: 161 B

```
{"success": "true"}
```

Testing our API with Postman

. . .

Console App — the client for our API

It's time to modify our console app to communicate with our webAPI. Open worker/Program.cs. We'll need to import types from other namespaces, so add the following lines to the top of this file:

```
using System.Net.Http;
using System.Threading.Tasks;
using System.Net.Http.Headers;
using System.Text;
using Newtonsoft.Json;
```

worker.cs hosted with ♡ by GitHub view raw

Define a new method PostMessage, just above Main(), to send messages to our web service.

```
public static async Task PostMessage(string postData)
{
    var json = JsonConvert.SerializeObject(postData);
    var content = new StringContent(json, UnicodeEncoding.UTF8, "application/json");

    using (var httpClientHandler = new HttpClientHandler())
    {
        httpClientHandler.ServerCertificateCustomValidationCallback = (message, cert, chain, erro
        using (var client = new HttpClient(httpClientHandler))
        {
            var result = await client.PostAsync("https://localhost:5001/api/values", content);
            string resultContent = await result.Content.ReadAsStringAsync();
            Console.WriteLine("Server returned: " + resultContent);
        }
    }
```

worker.cs hosted with ♡ by GitHub view raw

Let's talk through that function. It's defined as 'async' which means that it can be executed asynchronously, in the background, without blocking the currently executing thread. This is important for functions that may wait on external resources (like our webAPI).

We use JsonConvert (from the NewtonSoft package) to serialize the object we're going to post, and a StringContent object to format out text for HTTP communication. We create an HttpClientHandler; the 'using' statement ensure's that its disposed of, even if an exception is thrown. Overriding the HttpClientHandlers's certification validation allows us to use https locally without jumping through hoops.

Finally, PostAsync sends the message to our webAPI, and ReadAsStringAsync returns the server's response. The 'await' keyword specifies that, within this function, we want to wait for those Async methods to return before continuing to the next statement.

The last thing we need to do is modify our Main() method to invoke PostMessage(), and wait for it to return.

```
static void Main(string[] args)
{
  Console.WriteLine("Posting a message!");
  PostMessage("test message").Wait();
}
```

worker.cs hosted with ♡ by GitHub view raw

From the VS Code Debug Pane, launch the web app first, then launch the console app. Add some breakpoints to trace the request from the console ap sends it, to the point where the webAPI receives it, to when the response is returned to the client.

· · ·

Stay tuned for Part 2 of this tutorial, where we'll learn about Docker so that we can execute both of these apps within containers.

Docker Web Development Dotnet Core Rabbitmq Visual Studio Code