

# Dockerizing ASP.NET Core and Blazor Applications on Mac

PRESENTED BY : VINCENT MAVERICK DURANO

## Table of Contents

Introduction.....	5
Background.....	5
Who is this for?.....	6
What You Will Learn.....	7
Setup the Development Environment .....	8
Prerequisites.....	8
macOS 10.12 “Sierra” and later versions .....	8
Visual Studio Community.....	8
.NET Core (optional) .....	8
Docker Engine .....	8
Valentina Studio.....	8
Download Source.....	9
Five Players, One Goal .....	10
ASP.NET Core Web API.....	10
Entity Framework Core.....	10
Blazor.....	10
SQL Server.....	11
Docker.....	11
Application Flow .....	11
Configure a Database.....	12
SQL Server for Linux Prerequisites.....	12
Connect to SQL Server Inside a Docker Container .....	17
Create an Empty Database .....	21
Create an ASP.NET Core Web API Project .....	23
First Run.....	26
Configure Data Access with Entity Framework Core .....	26
Create Models .....	28
Define a DbContext.....	29
Add Database Migration .....	29
Seed Test Data on Application Startup .....	33
Implement a Data Repository Interface .....	35
Create a Data Manager.....	36
Register IDataRepository Interface as a Service .....	37

Create an API Controller .....	38
Enable CORS .....	41
Testing the API Endpoints .....	41
Dockerize ASP.NET Core Web API Application .....	44
Enable Docker Support .....	44
Docker File.....	45
Managing Containers .....	45
Docker-Compose File .....	46
Execute the Docker Compose.....	47
Testing the Dockerized Web API .....	48
Create Your First Blazor App on Docker .....	49
Create a Runtime Docker Container for Blazor.....	50
Create a Blazor App.....	52
Add a New Component.....	56
Modify the Index.html File .....	60
Run the Blazor App .....	62
GitHub Repo .....	63
Summary .....	63
References .....	64

## About Author



**Vincent Maverick Durano** works as a Senior Software Engineer in a research and development company, focusing mainly on web and mobile technologies. He is a nine-time Microsoft MVP, three-time C# Corner MVP, CodeProject MVP, Microsoft Influencer, DZone MVB and a regular contributor at CodeProject, C# Corner, Microsoft TechNet Wiki, AspSnippets and Xamarin. He also contributes at the official Microsoft ASP.NET community site where he became one of the all-time top answerers with All-Star recognition level (the highest attainable level).

He authored a few e-books for C# Corner:

- **GridView Control Pocket Guide,**
- **ASP.NET MVC 5: A Beginner's Guide**

And is now working on a new book entitled **Understanding Game Development using Xamarin.Forms and ASP.NET.**

He runs a blog at <http://vmsdurano.com>, created a few open-source projects that is hosted on Codeplex and GitHub. He also developed VMD.RESTApiResponseWrapper.Core and VMD.RESTApiResponseWrapper.Net NuGet packages.

## Introduction

I've been exploring .NET Core since it was still vNext to ASP.NET 5 until it officially came a .NET Core / ASP.NET Core version 1. I was really happy and amazed that Microsoft came into a realization to rebuild .NET and making it cross-platform and run everywhere. To me, I think it's one of the greatest milestones that Microsoft ever did as this opens up .NET to an entirely new audience of developers and designers.

Technologies are constantly evolving and as developer, we need to cope up with what's the latest or at least popular nowadays. As a starter you might find yourself having a hard-time catching up with latest technologies because it will give you more confusion as to what sets of technologies to use and where to start. We know that there are tons of resources out there that you can use as a reference to learn but you still find it hard to connect the dots in the picture. Sometimes you might think of losing the interest to learn and give up. If you are confused and no idea how to start building an ASP.NET Core and Blazor apps and running them on Docker, then this book is for you.

## Background

I was researching about the concept of “microservices” since the past years and given that .NET Core is open-source, cross-platform and runs everywhere, I had this curiosity of running it in a new set of environments to get a feel of how great it is. And then I started out exploring .NET Core on Mac and running it on Docker when it came out version 1 and a year after, I started trying out dockerizing ASP.NET Core 2.0 on Mac environment. It was a fun experiment but at the same time hard because I am used to Windows environment. Building apps in Mac environment is bit different because we will be dealing with the new commands, tools, and the file structure. In other words, I was not that familiar with the Mac environment and putting the pieces together was really a P.I.T.A.

Early this year Microsoft announce a new experimental project from the ASP.NET team called Blazor. Blazor is an experimental web UI SPA framework based on C#, Razor, and HTML that runs in the browser via WebAssembly without JavaScript. Yes, you heard that right – without JavaScript! That being said. Blazor is still on its experimental stage and we can't guarantee anything until it will be officially released.

I personally find the framework very interesting. I think Blazor is going to be incredibly a hit and that's because I see WebAssembly as actually superseding JavaScript. Sure, JavaScript and its frameworks aren't going anywhere, but why would you teach a new programmer JavaScript when you can just teach them C#, Python, etc. and have them work with simpler tools in a more performant environment?

It was on my to-do list to try out Blazor but I just didn't get the change to make time for it because of priorities and I've been very busy at work. Fortunately, I was able to find some time this week and gave Blazor a shot. My first attempt was running it on Windows, but I also wanted to test out Blazor on Mac and running it on Docker.

It took me a few days to put the pieces together and trying to figure out how to make them work. With a lot of research, some patience and head scratching, I was able to connect the dots in the picture. So that's why here I am trying to share the fun and experience to those who are also interested.

## Who is this for?

While there are a bunch of resources on the web that demonstrate how to build and run .NET Core apps in docker, it seems to me that there are only limited resources on doing it on a Mac. This article will walk you through on building your first Blazor app with ASP.NET Core Web API, Entity Framework Core, SQL Server and running them on a Docker container.

This book is targeted for beginners to intermediate .NET developers who want to jump on ASP.NET Core, Blazor and Docker on Mac environment and get your hands dirty with practical example.

I've written this book in such a way that it's easy to follow and understand by providing step-by-step process with detail code explanation as possible. As you go along and until such time you finished following the article, you will learn the basic concepts and fundamentals of each of the technologies used for building the whole application and how each of them connects to each other.

## What You Will Learn

Here's what you can learn from this book:

- The goal of what we are trying to achieve
- Setting up the development environment
- Configuring a database using SQL Server for Linux
- Using Valentina Studio for managing database
- Creating an ASP.NET Core Web API application
- Dockerizing the Web API application
- Creating your first Blazor application
- Connecting all applications together from a Docker container

## Setup the Development Environment

Let's go ahead and install the required tools and SDKs for us to build our application in MAC environment. If you already have installed the tools mentioned in the list below, then you may skip this step but just make sure you update them to the latest version as possible.

### Prerequisites

macOS 10.12 "Sierra" and later versions

You need to have at least macOS 10.12 "Sierra" version on your MAC machine as we are going to use .NET Core 2.1 as the target framework for building the apps.

### Visual Studio Community

The Community edition of Visual Studio is a free and full-featured solution that enables developers to build applications for Android, iOS, macOS, Cloud and Web. We will be using this editor to build a Blazor and ASP.NET Core Web API app using C# code. For more information about the features of Visual Studio Community edition, see:

<https://visualstudio.microsoft.com/vs/community/>

### .NET Core (optional)

.NET Core gives us the .NET command line tools to build and run .NET Core apps without any IDE. I tagged it as optional because .NET Core SDK should be included when you install Visual Studio for MAC. For more information about .NET Core, see: <https://docs.microsoft.com/en-us/dotnet/core/>

### Docker Engine

Docker creates simple tooling and a universal packaging approach that bundles up all application dependencies inside a container. Docker Engine enables containerized applications to run anywhere consistently on any infrastructure. We will see how we can use Docker to deploy and run .NET Core apps inside a container. For more information about the Docker Engine, see: <https://www.docker.com/why-docker>

### Valentina Studio

As far as I know, Microsoft SSMS team has no plan as of this time of writing to make a cross-platform version of SQL Server Management Studio. As a result, if you are planning to use SQL Server database in your MAC and wanted to use a tool (GUI) for managing your database then you may end up using a third-party tool such as Valentina Studio, SQLPro, Navicat, TablePlus and etc.

For this demo, I'm going to use Valentina Studio because it's free and it doesn't require you to run a Windows Virtual Machine (VM) on your MAC. Adding to that, it also supports a few database engines such as PostgreSQL, MySQL and SQLite.



## Download Source

- [.NET Core for MAC](#) (The latest version as of this time of writing is 2.1.302)
- [Visual Studio Community 2017 For MAC](#) (The latest version as of this time of writing is v7.6.7)
- [Docker Community Edition for MAC](#) (The latest version as of this time of writing is 18.06.1\_ce)
- [Valentina Studio](#) (The latest version as of this time of writing is 8.6 (64-bit) )

Make sure to download and install the prerequisites before you go any further.

## Five Players, One Goal

As you can see from the prerequisites section, we are going to use various technologies to build this whole application to fulfill a goal. At this point, you should already have the needed frameworks installed in your machine.

Our main goal is to build a simple data-driven web application using cutting-edge technologies: Blazor, ASP.NET Core Web API, Entity Framework Core, SQL Server and Docker.

Before we talk about the high-level application flow on how each technology connects together, let's take a look at first the brief overview of them.

### ASP.NET Core Web API

The ASP.NET Core Web API is an extensible framework for building HTTP based services that can be accessed in different applications on different platforms. It works more or less the same way as ASP.NET Core MVC web application except that it sends data as a response instead of HTML View. It is like a webservice or WCF service, but the exception is that it only supports HTTP protocol. Here's the definition taken from the official documentation:

[https://msdn.microsoft.com/en-us/library/hh833994\(v=vs.108\).aspx](https://msdn.microsoft.com/en-us/library/hh833994(v=vs.108).aspx)

*ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices. ASP.NET Web API is an ideal platform for building RESTful applications on the .NET Framework.*

### Entity Framework Core

Entity Framework (EF) Core is a lightweight, extensible, and cross-platform version of the popular Entity Framework data access technology. EF Core is an object-relational mapper (O/RM) that enables .NET developers to work with a database using .NET objects. It eliminates the need for most of the data-access code that developers usually need to write. According to the official documentation: <https://docs.microsoft.com/en-us/ef/ef6/>

*As an O/RM, EF reduces the impedance mismatch between the relational and object-oriented worlds, enabling developers to write applications that interact with data stored in relational databases using strongly-typed .NET objects that represent the application's domain, and eliminating the need for a large portion of the data access "plumbing" code that they usually need to write.*

### Blazor

Blazor is a single-page web app (SPA) framework built on top .NET Core that runs in the browser with WebAssembly. For more information about Blazor, see:

<https://blazor.net/docs/index.html>

---

*Warning! Blazor is an unsupported experimental web framework that shouldn't be used for production workloads at this time.*

---

## SQL Server

Microsoft SQL Server is a relational database management system developed by Microsoft. As a database server, it is a software product with the primary function of storing and retrieving data as requested by other software applications (Desktop, Service, Mobile or Web) — which may run either on the same computer or on another computer across a network or internet.

## Docker

Docker provides container software that is ideal for developers and teams looking to get started and experimenting with container-based applications. For more information about Docker, see:

<https://www.docker.com/>

## Application Flow

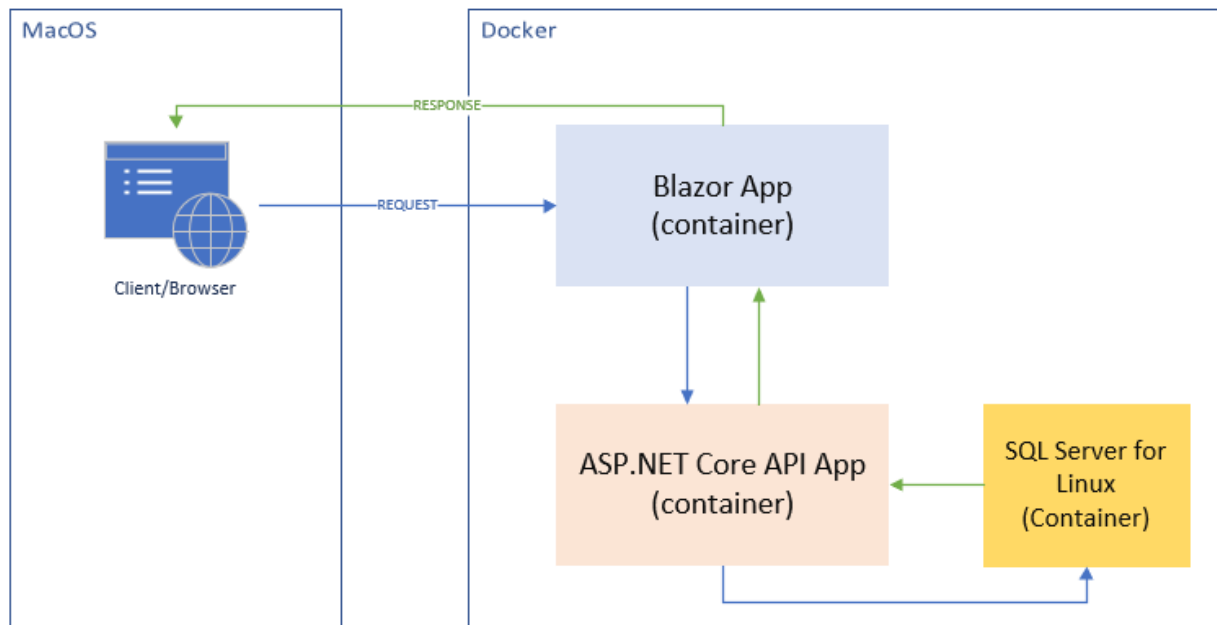


Figure 1: Application flow

The diagram above shows what we are trying to achieve in this article. We basically need three main Docker containers: a Blazor app for UI, Web API for serving JSON data and SQL Server for storing the actual data.

Let's take a look at how to achieve the goal. Let's start by setting up a database.

## Configure a Database

In order for us to work with real data, we need to have a persistent storage called "database". In Windows environment, setting up a new database is very easy and straight forward. However, in the context of macOS is different as there is no SQL Server version for MAC yet as of this time of writing.

The only way to run SQL Server on MAC is to install a Windows VM and run the it from there. However, with the release of SQL Server 2017, Microsoft has made it available for macOS and Linux environments. This enables us to use Docker container to run SQL Server which acts as if the server is running on your MAC. Thanks to Docker!

### SQL Server for Linux Prerequisites

According to the [documentation](#), SQL Server for Linux requires at least a minimum of 2 GB of disk space to run. However, Docker only allocates 2 GB by default. Therefore, we should increase this allocation to ensure that SQL Server will be able to run. To do that, follow the steps below:

1. **Open** the Docker application.
2. **Logon** using your Docker credentials.
3. On the Docker main menu, select **Preferences**.
4. Click on the **Advance** tab and adjust the Memory allocation to something like 4 GB just to be safe.

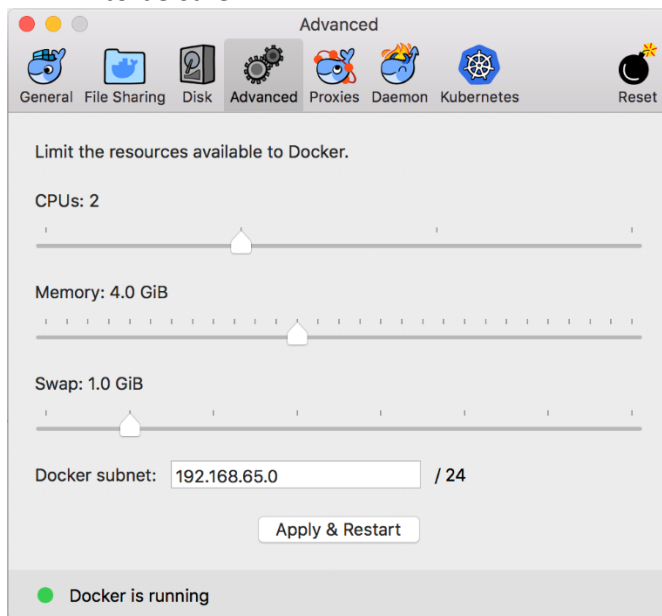


Figure 2: Docker memory allocation

## 5. Click **Apply & Restart**.

Now let's download the SQL Server for Linux Docker container images from the [Docker Hub](#) by running the following command in the Terminal console:

```
docker pull mcr.microsoft.com/mssql/server:2017-latest
```

The **docker pull** command downloads a particular image or sets of images from the registry. The figure below shows when your pull is successful:

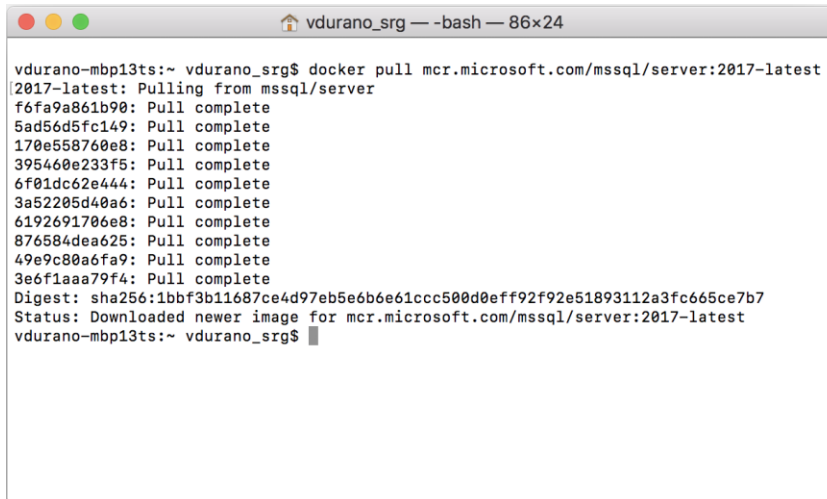
A terminal window titled 'vdurano\_srg' with a width of 86x24. The prompt is 'vdurano-mbp13ts:~ vdurano\_srg\$'. The command entered is 'docker pull mcr.microsoft.com/mssql/server:2017-latest'. The output shows the image being pulled from the registry, with a list of layers being pulled and their status as 'Pull complete'. The final output shows the digest and status: 'Digest: sha256:1bbf3b11687ce4d97eb5e6b6e61ccc500d0eff92f92e51893112a3fc665ce7b7' and 'Status: Downloaded newer image for mcr.microsoft.com/mssql/server:2017-latest'. The prompt returns to 'vdurano-mbp13ts:~ vdurano\_srg\$'.

Figure 3: Downloading the SQL Server for Linux Docker container

To confirm that the image that we pulled, run:

```
docker images
```

The command above should result to something like this:

```

vdurano_srg — -bash — 85x24
[vdurano-mbp13ts:~ vdurano_srg$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
mcr.microsoft.com/mssql/server  2017-latest        885d07287041       11 days ago
1.45GB
vdurano-mbp13ts:~ vdurano_srg$

```

Figure 4: List of Docker images

We can then start running the SQL Server container image within Docker by running the following command:

```
docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD=SuperSecret1!' -p 1433:1433 --name sql17_linux -d mcr.microsoft.com/mssql/server:2017-latest
```

Take note of the **SA\_PASSWORD** value as we are going to use those values in setting up a Connection String to enable us to communicate with the database.

The following table provides a description of the parameters in the previous *docker run* example taken from the official [documentation here](https://docs.microsoft.com/en-us/sql/linux/quickstart-install-connect-docker?view=sql-server-2017). <https://docs.microsoft.com/en-us/sql/linux/quickstart-install-connect-docker?view=sql-server-2017>

Parameter	Description
<b>-e 'ACCEPT_EULA=Y'</b>	Set the <b>ACCEPT_EULA</b> variable to any value to confirm your acceptance of the End-User Licensing Agreement. Required setting for the SQL Server image.

Parameter	Description
<b>-e</b> <b>'SA_PASSWORD=&lt;YourStrong!Passw0rd&gt;'</b>	Specify your own strong password that is at least 8 characters and meets the SQL Server password requirements. Required setting for the SQL Server image.
<b>-p 1433:1433</b>	Map a TCP port on the host environment (first value) with a TCP port in the container (second value). In this example, SQL Server is listening on TCP 1433 in the container and this is exposed to the port, 1433, on the host.
<b>--name sql17_linux</b>	Specify a custom name for the container rather than a randomly generated one. If you run more than one container, you cannot reuse this same name.
<b>microsoft/mssql-server-linux:2017-latest</b>	The SQL Server 2017 Linux container image.

To verify that the container is running, run:

```
docker ps
```

and it should give you the following result:

```
vdurano-srg — -bash — 90x22
vdurano-mbp13ts:~ vdurano_srg$ docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD=SuperSecret1!' -p 1433:1433 --name sql17_linux -d mcr.microsoft.com/mssql/server:2017-latest
026a7605178be4aac85f2e11fc7f455749d60562074180dc0299355077165fae
vdurano-mbp13ts:~ vdurano_srg$ docker ps
CONTAINER ID        IMAGE                                     COMMAND
CREATED            STATUS              PORTS              NAMES
026a7605178b       mcr.microsoft.com/mssql/server:2017-latest  "/opt/mssql/bin/sqls..."
About a minute ago Up About a minute   0.0.0.0:1433->1433/tcp sql17_linux
vdurano-mbp13ts:~ vdurano_srg$
```

Figure 5: List of Docker containers

If the SQL Server for Linux Docker container exits, you can run it again using the following command:

```
docker run -e 'SA_PASSWORD=SuperSecret1!' -p 1433:1433 -d
mcr.microsoft.com/mssql/server:2017-latest
```



## Connect to SQL Server Inside a Docker Container

Open Valentina Studio. You may be prompted to enter a key the very first time you open it as shown in the figure below:

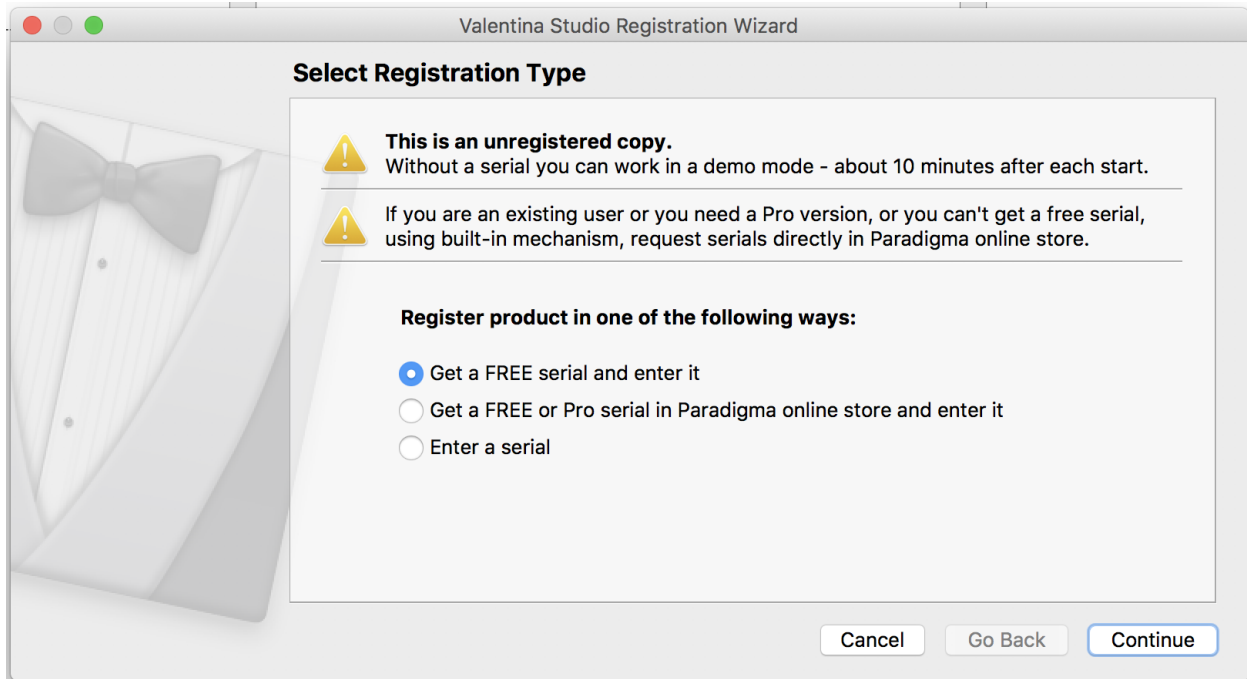
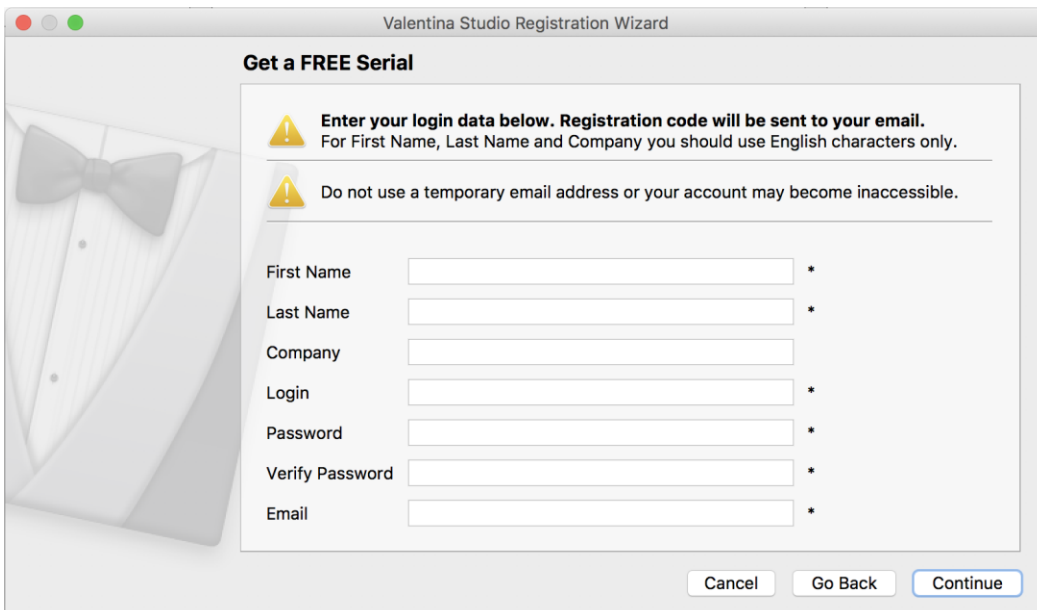


Figure 6: Valentina Studio registration - screen 1

Select **Get a FREE serial and enter it**, then click **Continue** and it should take you to the next screen below:



The image shows the 'Valentina Studio Registration Wizard' window, screen 2. The title bar says 'Valentina Studio Registration Wizard'. The main window has a light gray background with a faint image of a white tuxedo with a bow tie on the left. The title of the wizard is 'Get a FREE Serial'. There are two warning icons with exclamation marks. The first warning says: 'Enter your login data below. Registration code will be sent to your email. For First Name, Last Name and Company you should use English characters only.' The second warning says: 'Do not use a temporary email address or your account may become inaccessible.' Below the warnings, there are seven input fields with labels: 'First Name', 'Last Name', 'Company', 'Login', 'Password', 'Verify Password', and 'Email'. Each field has a small asterisk to its right. At the bottom right, there are three buttons: 'Cancel', 'Go Back', and 'Continue'.

Figure 7: Valentina Studio registration - screen 2

Enter all the required information and then click **Continue**.

When your registration is successful, you should be able to receive an email from Valentina support containing the free serial keys for Mac, Linux and Windows.

After you've configured Valentina to use a free serial key, you should be able to see something like this:

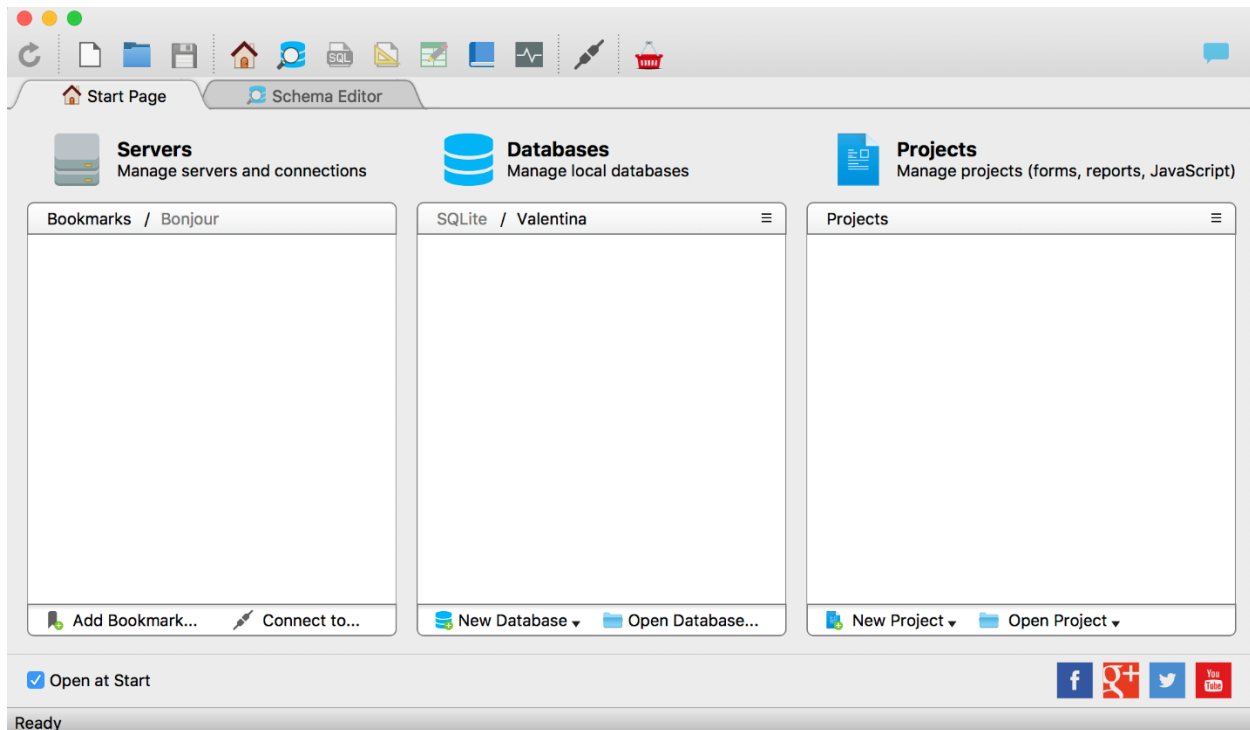
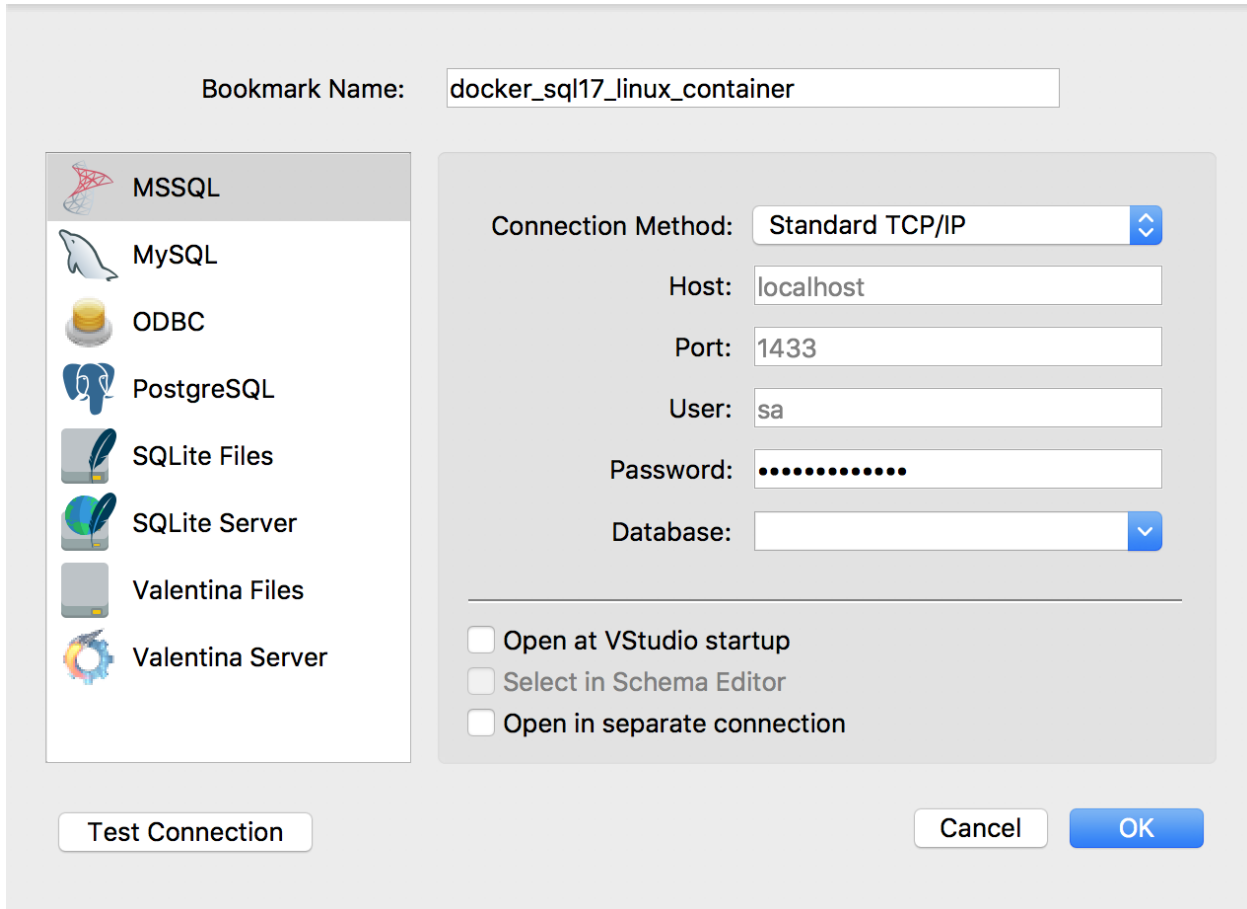










Figure 8: Valentina Studio start page

At the bottom of the “**Servers**” panel, click **Add Bookmark** and it should present you the following screen:



Bookmark Name:

 MSSQL  
 MySQL  
 ODBC  
 PostgreSQL  
 SQLite Files  
 SQLite Server  
 Valentina Files  
 Valentina Server

Connection Method:

Host:

Port:

User:

Password:

Database:

☐ Open at VStudio startup  
☐ Select in Schema Editor  
☐ Open in separate connection

Figure 9: Valentina Studio connecting to SQL Server for Linux container

Enter a **Bookmark Name**, **Port** and the **Password** you used for connecting the SQL Server for Linux hosted in Docker container in previous step. Its good practice to click “Test Connection” first to make sure you entered your information correctly, and if the test succeeds, click **OK** to save the bookmark.

Once you’ve saved your bookmark, you will see a new item under “**Servers**” panel which allows you to directly access your Docker SQL instance as shown in the figure below:

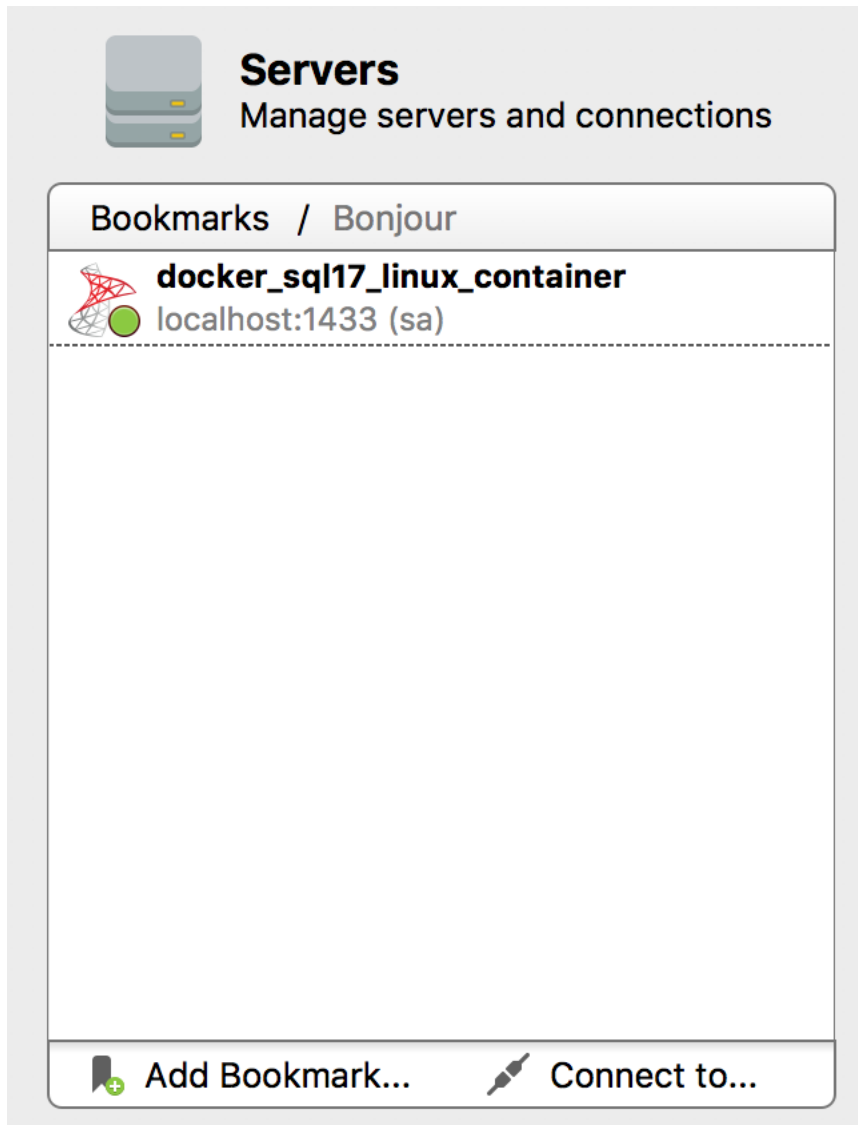


Figure 10: Valentina Studio SQL Server for Linux running

---

*Note that you can also use the SQL Server command-line tool, `sqlcmd`, inside the container to connect to SQL Server. For more information, see: <https://docs.microsoft.com/en-us/sql/linux/quickstart-install-connect-docker?view=sql-server-2017>*

---

## Create an Empty Database

Now it's time for us to create a new database. Double click the database instance that we've created earlier. It should display something like this:

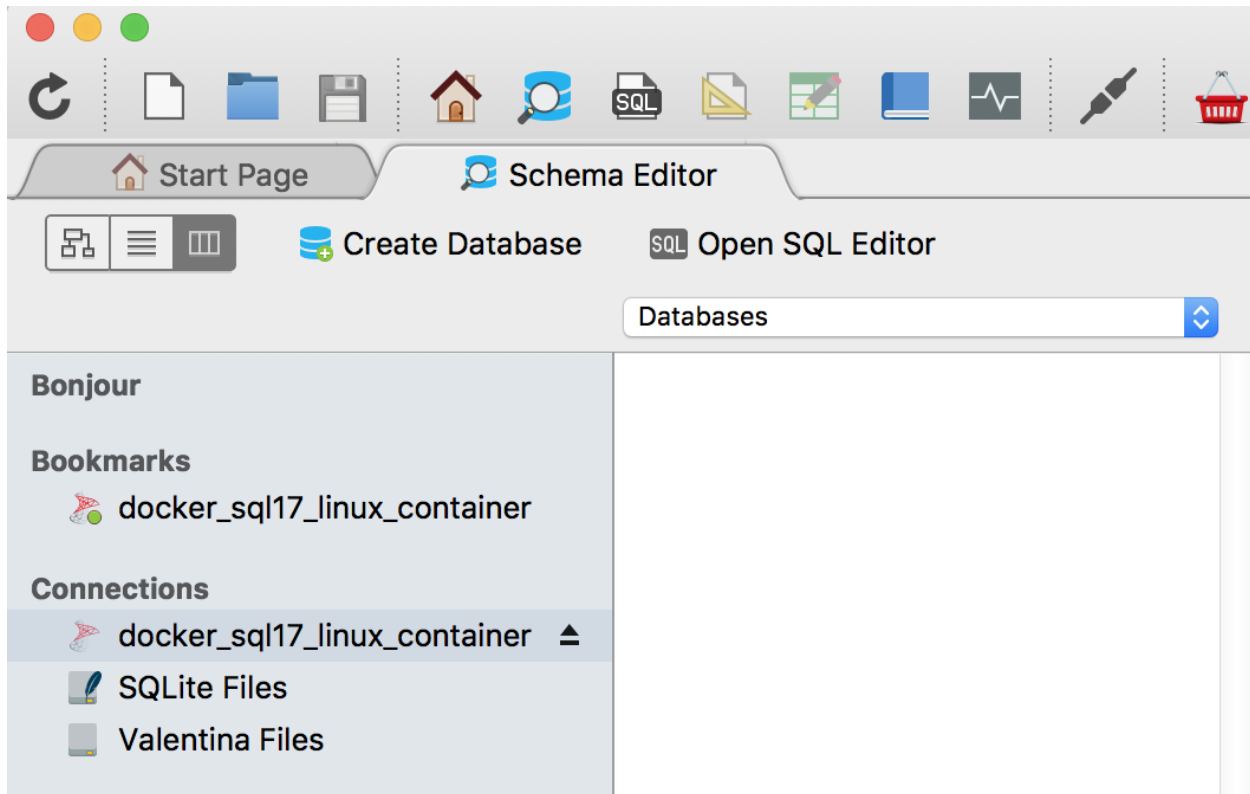


Figure 11: Valentina Studio SQL Server for Linux running

Click the **Open SQL Editor** and then execute the following script below:

```
CREATE DATABASE Band;
```

The script above should create a blank database called "**Band**" as shown in the figure below:

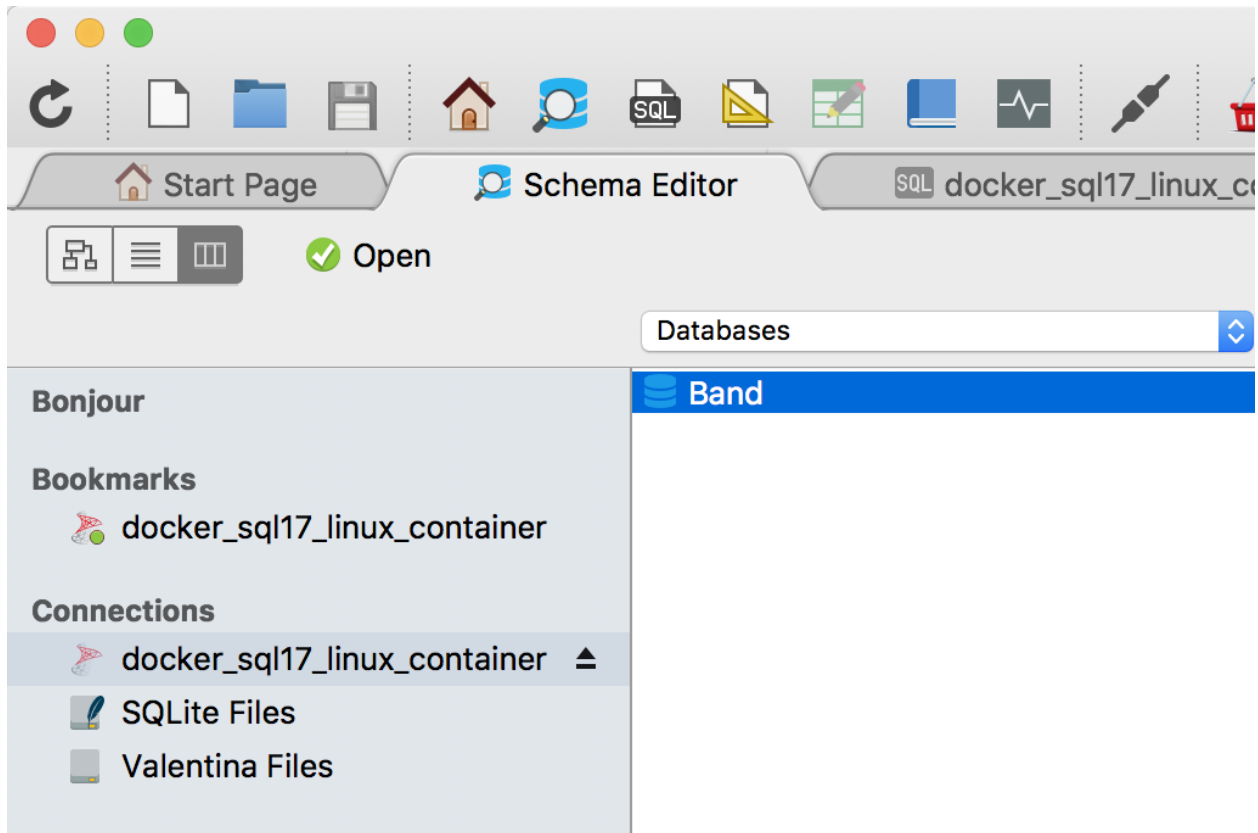


Figure 12: Created the Band database

Note that you can also use the Valentina Studio UI to create and manage database by clicking on the **Create Database** button.

At this point, we are now ready implement a basic Data Access.

## Create an ASP.NET Core Web API Project

Open Visual Studio for MAC and then create a **New Project**. Under **.NET Core > App**, select **ASP.NET Core Web API** just like in the figure below:

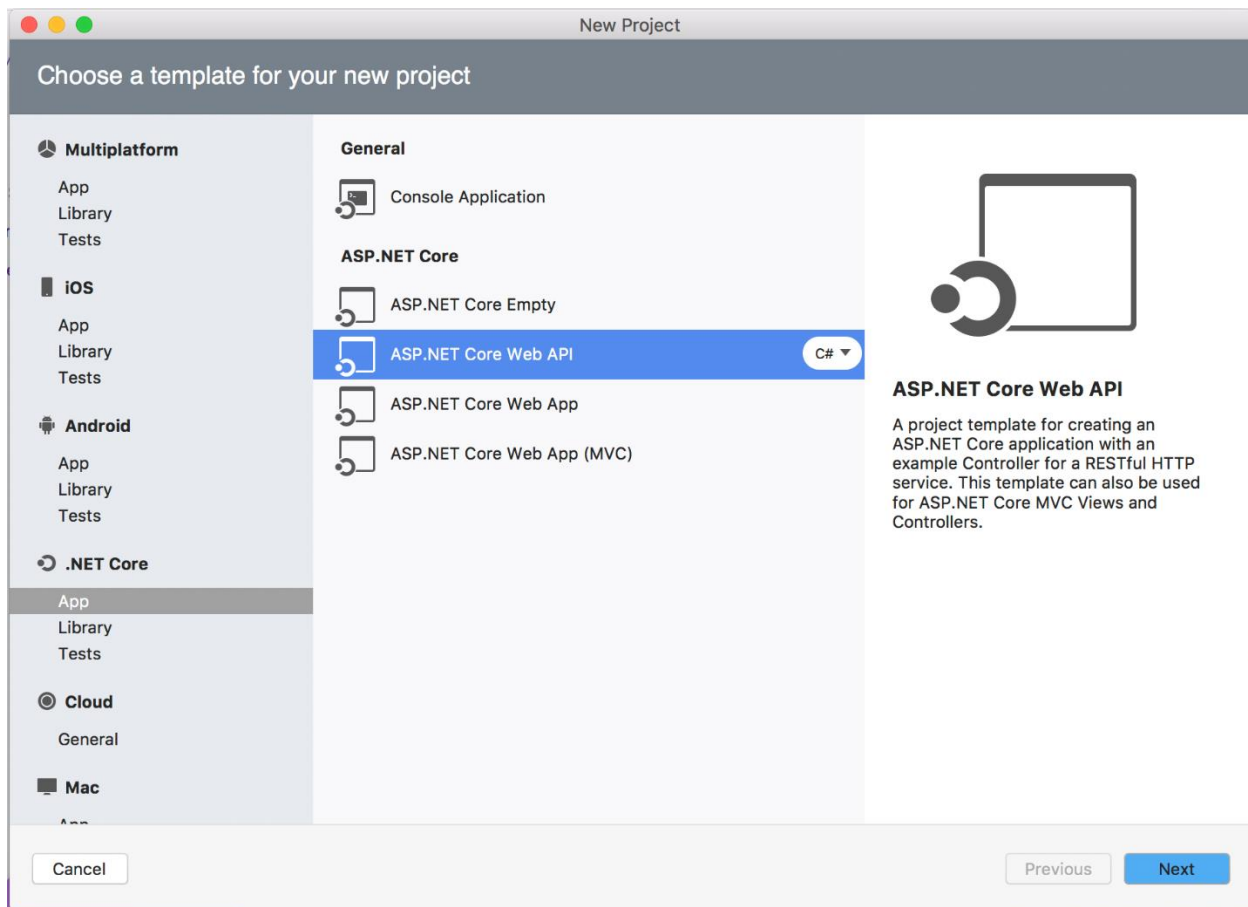


Figure 13: New ASP.NET Core Web API project

Click **Next** and it should present you the following screen below:

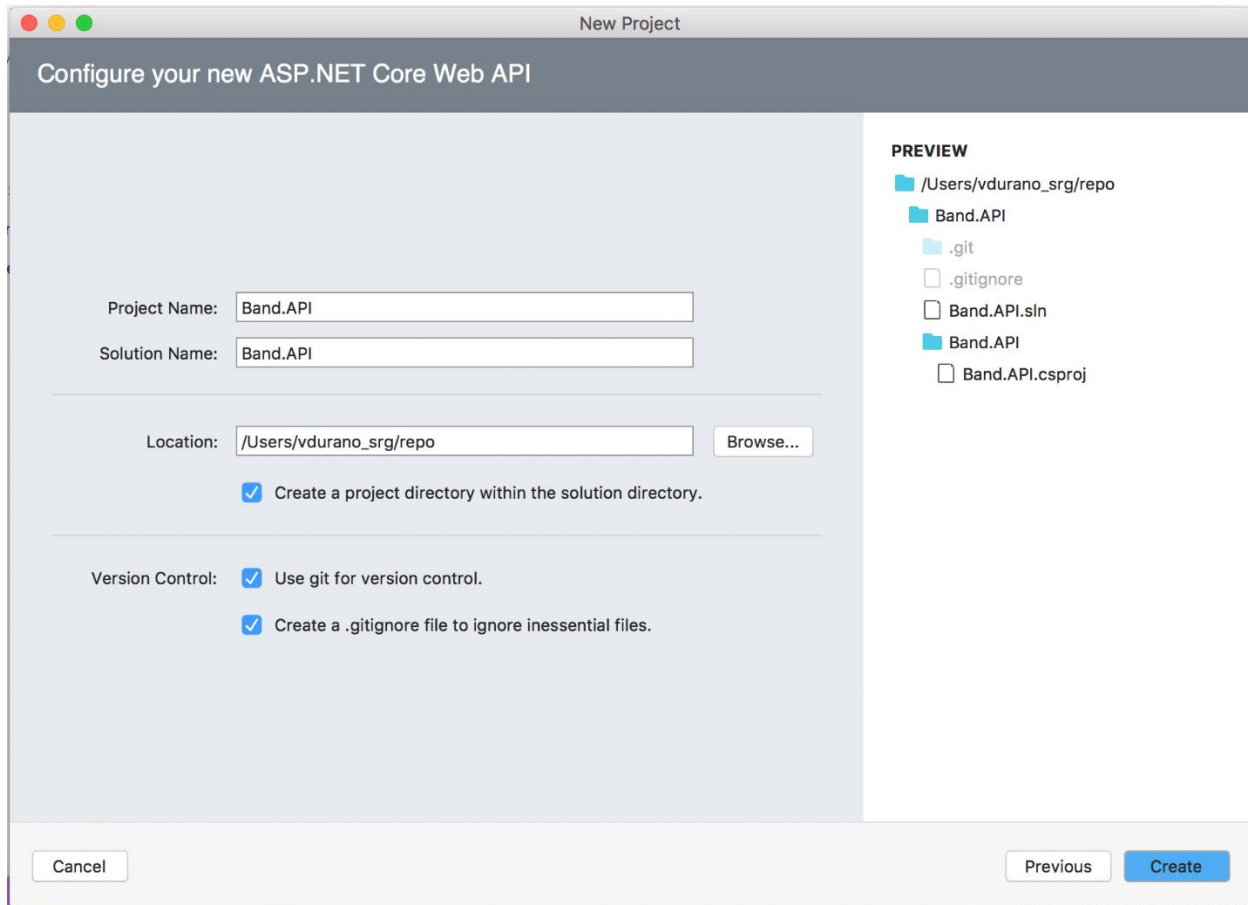


Figure 14: New ASP.NET Core Web API project

Enter the **Project Name**, **Solution Name** and **Location**. You can also configure version control but it's optional. Click **Create** to let Visual Studio generates the default files needed for our application. Here's what the Solution looks like:



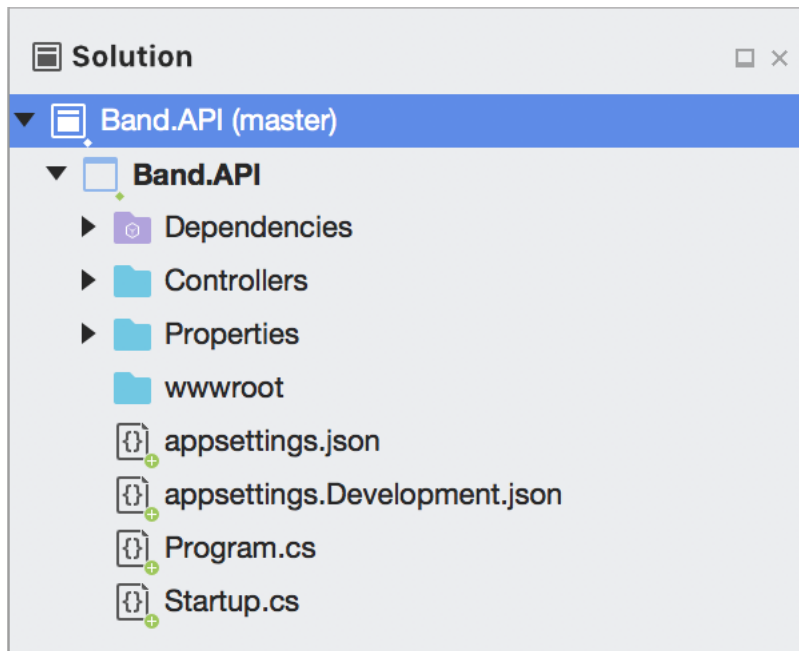


Figure 15: ASP.NET Core Web API default generated files

Let's take a quick overview about each file generated.

If you already know the core significant changes of ASP.NET Core then you may skip this part, but if you are new to ASP.NET Core then I would like to highlight some of those changes. If you have worked with previous versions of ASP.NET before then you will notice that the new project structure is totally different. The project now includes these files:

- **Dependencies:** contains both NuGet and SDK dependencies needed for the application.
- **Controllers:** this is where you put all your Web API or MVC controller classes. Note that the Web API project will by default generate the `ValuesController.cs` file.
- **Properties:** contains the **launchSettings.json** file which manages application configuration settings associated with each debug profile such as IIS, IIS Express and the application itself. This is where you define profile-specific configuration (Compilation and Debug profiles) for frameworks used in the application.
- **wwwroot:** is a folder in which all your static files will be placed. These are the assets that the web app will serve directly to the clients, including HTML, CSS, Image and JavaScript files.
- **appsettings.json:** contains application settings.
- **Startup.cs:** this is where you put your startup and configuration code.
- **Program.cs:** this is where you initialize all services needed for your application.

## First Run

To ensure that we got everything we need for our Web API project, let's try to build and run the project. We can do this by pressing the Command + Return keys or by clicking the Play button located at the Visual Studio menu toolbar. This will compile, build and automatically launches the browser window. If you are getting the following result below, then we can safely assume that our project is good to go.

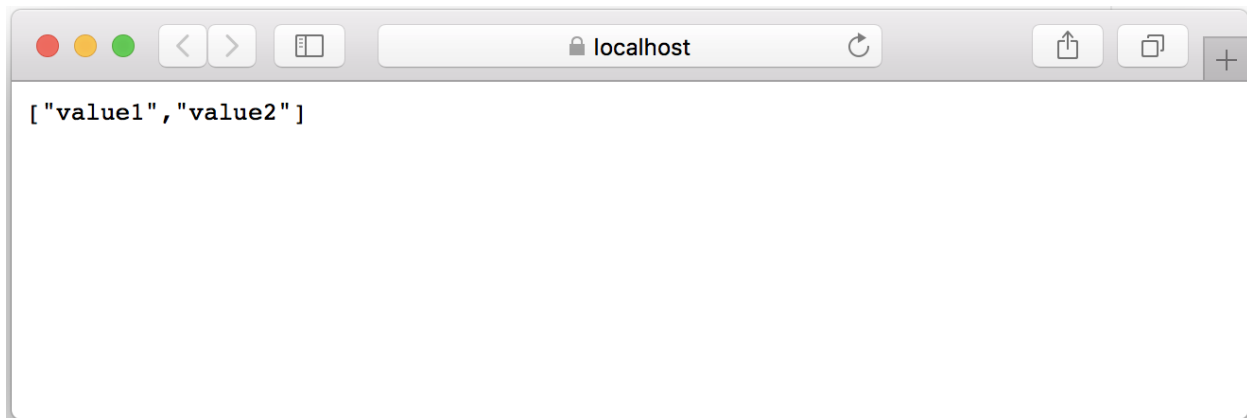


Figure 16: First run

Cool! Now let's move on to the next step.

## Configure Data Access with Entity Framework Core

Let's install Entity Framework Core packages into our project. From Visual Studio menu, go to **Project > Add NuGet Packages**. Alternatively, you can right-click on the **Dependencies** folder of the project and select **Add Packages**.

The figure below shows the NuGet Package Manager window:

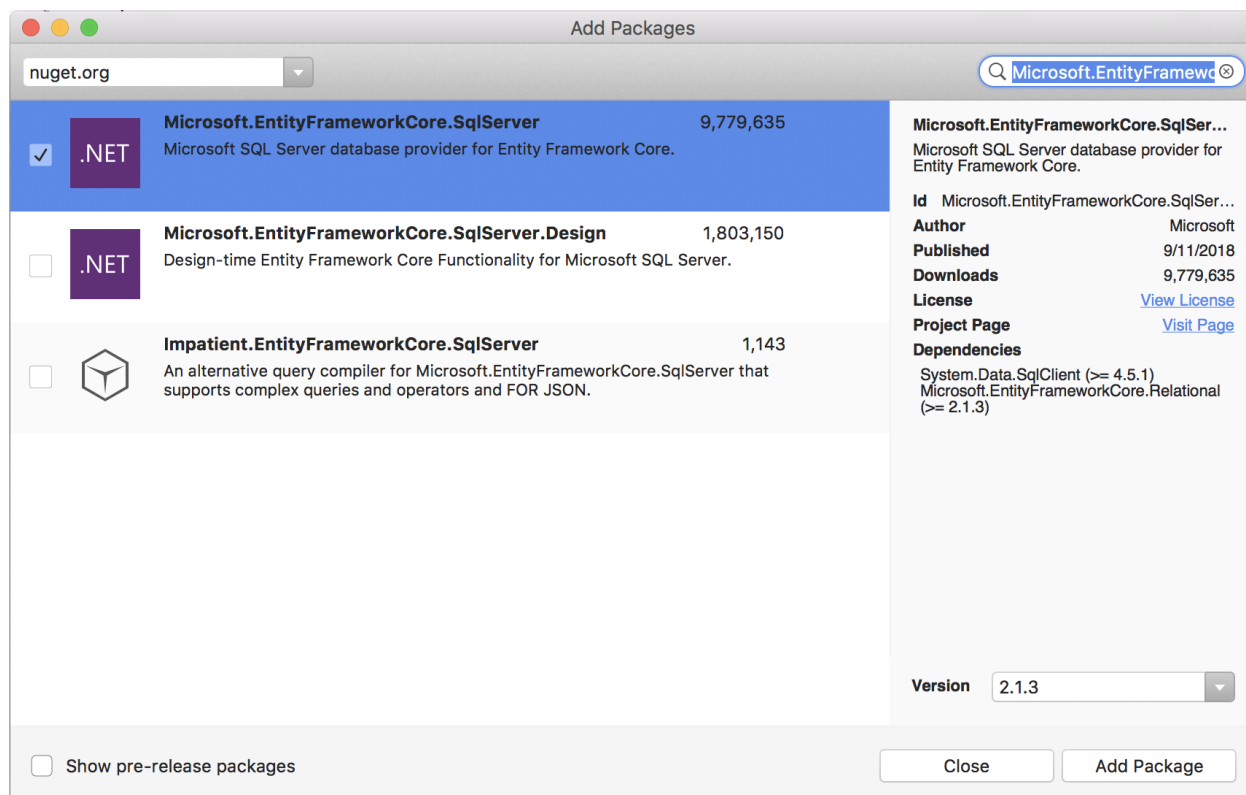


Figure 17: NuGet Package Manager window

On the search bar type in “**Microsoft.EntityFrameworkCore.SqlServer**” and then click **Add Package**.

Later in this demo we will use some Entity Framework Tools to maintain the database. So install the following tools package as well:

- **Microsoft.EntityFrameworkCore.Tools**

The versions of **Microsoft.EntityFrameworkCore.SqlServer** and **Microsoft.EntityFrameworkCore.Tools** as of this time of writing is **2.1.3**.

After successfully installing the packages above, make sure to check your project NuGet dependencies to ensure that we got them in the project. See the figure below:

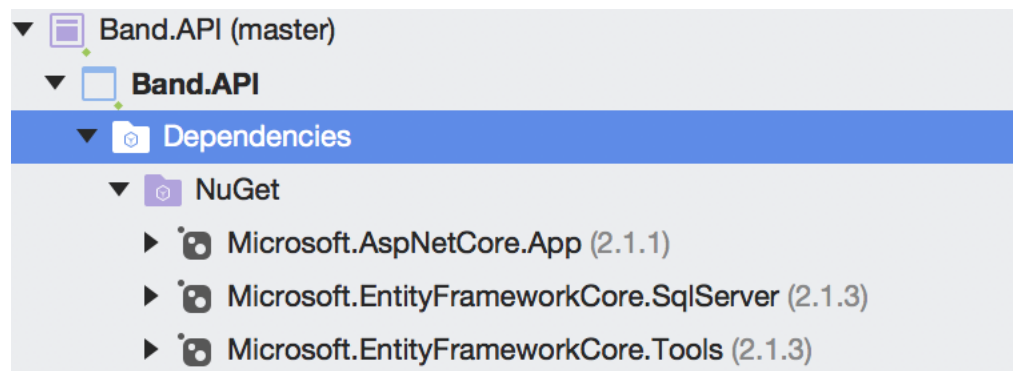


Figure 18: Nuget Package dependencies

These packages are used to provide Entity Framework Core migration. Migration enables us to generate database, sync and update tables based on our Entity Models.

For this demo, we are going to use Code-First approach, that is we create classes (POCO Entities) and generate a new database out from it. If you are new to Entity Framework Core, read my article other article here: <https://www.codeproject.com/Articles/1218427/Getting-Started-with-Entity-Framework-Core-Buildin>

## Create Models

Create a new folder in your Application and name it as “**Models**”. Under that folder, create a new class and name it as “**Band**” and then copy the following code below:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Band.API.Models
{
    public class Band
    {
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public int Id { get; set; }
        public string Name { get; set; }
        public string Genre { get; set; }
    }
}
```

The **Band** class is nothing but just a simple class that houses some properties. You may have noticed that we decorated the Id property with **[Key]** and **[DatabaseGenerated]** attributes. This is because we will be converting this class into a database table and the **Id** property will serve as

our primary key with auto-incremented identity. These attributes resides within **System.ComponentModel.DataAnnotations**. For more details about Data Annotations see: <https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/mvc-music-store/mvc-music-store-part-6>

## Define a DbContext

Entity Framework Core Code-First development approach requires us to create a data access context class that inherits from the **DbContext** class. Now let's add a new class under the **"Models"** folder. Name the class as **"BandContext"** and then copy the following code below:

using Microsoft.EntityFrameworkCore;

```
namespace Band.API.Models
{
    public class BandContext
    {
        public class ApplicationContext : DbContext
        {
            public ApplicationContext(DbContextOptions opts) : base(opts)
            {
            }

            public DbSet<Band> Bands { get; set; }
            protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
            {
            }
        }
    }
}
```

The code above defines a context and an entity class that makes up our model. A **DbContext** must have an instance of **DbContextOptions** in order to execute. This can be configured by overriding **OnConfiguring()**, or supplied externally via a constructor argument. In our example above, we opt to choose **constructor argument** for configuring our **DbContext**.

## Add Database Migration

Our next step is to add Code-First migrations. Migrations automates the creation of database based from our Model. We will be using EF Core tools to scaffold our migration and updating the database. We'll be using the command line (dotnet CLI) to run our migrations.

### *Setup a ConnectionString*

Let's go ahead and define the following database connection string under **appsettings.json** file:

```
"ConnectionString": "Server=127.0.0.1; Database=Band; User Id=sa;Password=SuperSecret1!;"
```

The **ConnectionString** above defines a set of attributes to connect the application to the database we configured previously. These attributes are:

- **Server:** We used an IP address to connect to the server using 127.0.0.1 which is equivalent to localhost. Please note that putting the 1433 port is not necessary as that's the default port number for connecting to SQL Server for Linux instance.
- **Database:** is the name of the database that you want to connect. In this case the Band database.
- **User Id:** The user id for connecting to SQL server. In this case "SA"
- **Password:** The password for connecting to SQL Server for Linux in Docker container. In this case, it's "SuperSecret1!"

### *Register DBContext as a Service*

Next, we need to register the **BandContext** as a Service and enable it to use SQL Server. To do that, follow the steps below:

(1) Open **Startup.cs** and declare the following namespaces below:

```
using Microsoft.EntityFrameworkCore;  
using BandModel = Band.API.Models;
```

(2) Next, add the following code within **ConfigureServices()** method to register the BandContext as a service:

```
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddDbContext<BandModel.BandContext>(opts => opts.UseSqlServer(Configuration["DbContextSettings:ConnectionString"]));  
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);  
}
```

(3) Build the application to ensure there's no error.

## Execute Database Migrations

Now open the Terminal console to the location where your project Solution is located and then do:

### **dotnet ef migrations add InitialMigration**

The ***dotnet ef migration*** is the command to add migrations. When the migration is successful, you should be able to see a new folder named "**Migrations**" that contains the migration files as shown in the figure below.

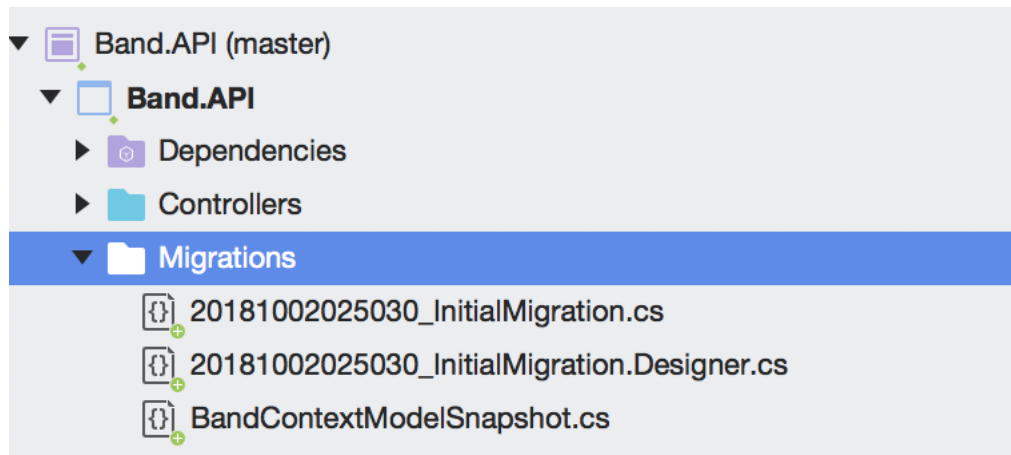


Figure 19: EF Migration files

At this point our Entity Model (**Bands DbSet**) isn't added to the database yet. We need to apply the migration to the database to create the schema by running the following command below in the Terminal console:

### **dotnet ef database update**

The command above should sync the Entity Model to the database. You can confirm that in Valentina Studio as shown in the figure below:

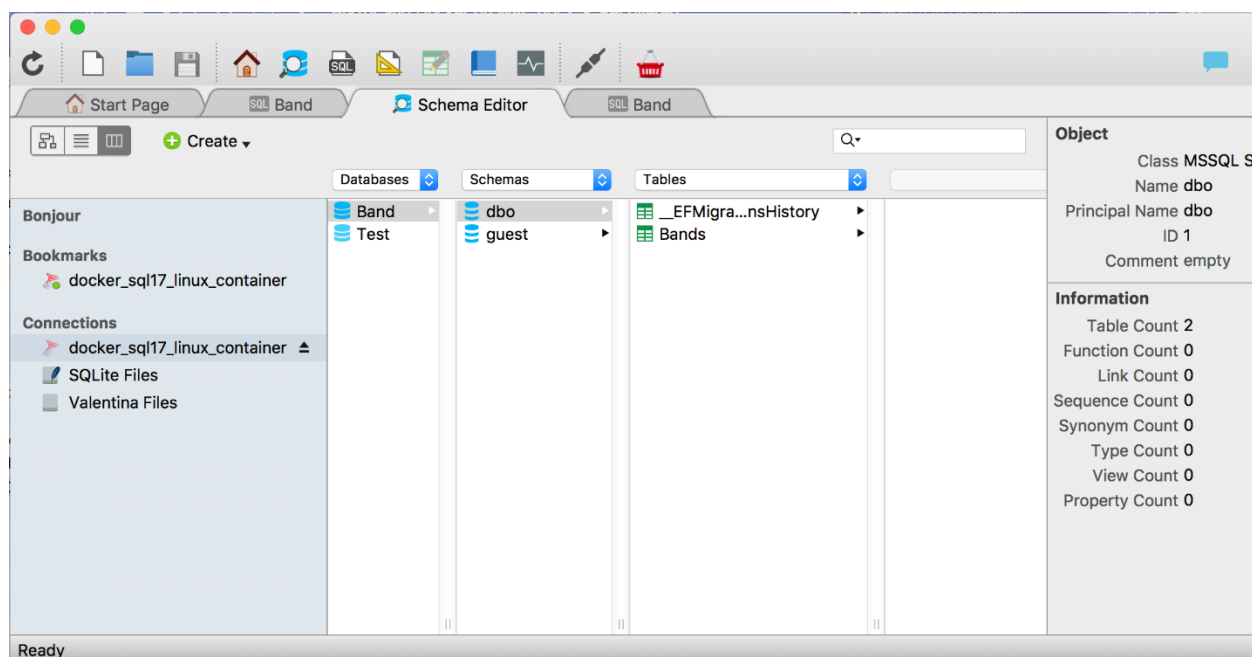


Figure 20: Generated Database schema from EF migration process

You can also use the **SQLCMD** command tool to verify if the migration is reflected inside the SQL Server for Linux in Docker container by running the following command in the Terminal console:

```
docker exec -it sql17_linux /opt/mssql-tools/bin/sqlcmd -S localhost -U sa -P SuperSecret1!
```

And then running the following command:

```
1> SELECT Name FROM sys.Databases
2> GO
```

This should result to this:

```
Band.API — docker exec -it s
[1> SELECT Name FROM sys.Databases
[2> GO
Name
-----
master
tempdb
model
msdb
Band
(5 rows affected)
```



Figure 21: Using SQLCMD

You can then drill-down to the schema by running the following:

```
1> USE band
2> GO
1> SELECT * FROM Bands
2> GO
```

It should display the Columns that we've migrated but of course it should yield an empty result for now.

### Seed Test Data on Application Startup

There are certain scenarios where you don't want to manually generate the database and schema after adding a migration. This can be accomplished by applying migrations at runtime. To do that let's setup a helper class for generating default test data.

Go ahead and create a new folder under the "**Models**" folder and create a new class called "**DBSeeder**" and replace the default generated code with the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace Band.API.Models.DataManager
{
    public class DBSeeder
    {
        public static void Seed(IApplicationBuilder appBuilder)
        {
            using (var serviceScope = appBuilder.ApplicationServices.CreateScope())
            {
                var context = serviceScope.ServiceProvider.GetService<BandContext>();

                using (context)
                {
                    context.Database.Migrate();

                    if (!context.Bands.Any())
```

```
{
    var bands = new List<Band>()
    {
        new Band(){
            Name = "Alice In Chains",
            Genre="Heavy Metal"
        }
    };

    context.Bands.AddRange(bands);
    context.SaveChanges();
}
}
```

The **Database.Migrate()** method piece is responsible for two things:

- The creation of database in SQL Server if it does not exist yet.
- Migrating the database schemas to the latest version.

The final step is to apply the migration when the application starts up. To do that, you can call the **DBSeeder.Seed()** method to the **Configure()** method in the **Startup** class of the Web API project:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseHsts();
    }

    DBSeeder.Seed(app);
}
```

```
app.UseHttpsRedirection();  
app.UseMvc();  
}
```

*Caution, this approach isn't for everyone. While it's great for apps with a local database, most applications will require more robust deployment strategy like generating SQL scripts.*

## Implement a Data Repository Interface

We don't want our API Controller to access directly our **BandContext** service, instead we will let other service handle the communication between our **BandContext** and API Controller. Having that said, we are going to implement a basic generic repository for handling data access within our application. Add a new folder under “**Models**” and name it as “**Repository**”. Create a new interface and name it as “**IDataRepository**”. Update the code within that file so it would look similar to the following code below:

```
using System.Collections.Generic;  
  
namespace Band.API.Models.Repository  
{  
    public interface IDataRepository<TEntity, U> where TEntity : class  
    {  
        IEnumerable<TEntity> GetAll();  
        TEntity Get(U id);  
        int Add(TEntity b);  
        int Update(U id, TEntity b);  
        int Delete(U id);  
    }  
}
```

The code above defines our **IDataRepository** interface. An interface is just a skeleton of a method without the actual implementation. This interface will be injected into our API Controller so we will only need to talk to the interface rather than the actual implementation of our repository. One of the main benefits of using interface is to make our code reusable and easy to maintain.

## Create a Data Manager

Next is we are going to create a concrete class that implements the **IDataRepository** interface. Add a new folder under Models and name it as **"DataManager"**. Create a new class and name it as **"BandManager"**. Update the code within that file so it would look similar to the following code below:

```
using Band.API.Models.Repository;
using System.Collections.Generic;
using System.Linq;

namespace Band.API.Models.DataManager
{
    public class BandManager: IDataRepository<Band, int>
    {
        BandContext _context;
        public BandManager(BandContext context)
        {
            _context = context;
        }

        public Band Get(int id)
        {
            return _context.Bands.FirstOrDefault(b => b.Id == id);
        }

        public IEnumerable<Band> GetAll()
        {
            return _context.Bands.ToList();
        }

        public int Add(Band band)
        {
            _context.Bands.Add(band);
            int id = _context.SaveChanges();
            return id;
        }

        public int Delete(int id)
        {
            var band = _context.Bands.FirstOrDefault(b => b.Id == id);
            if (band != null)
            {
```

```
        _context.Bands.Remove(band);
        _context.SaveChanges();
    }
    return id;
}

public int Update(int id, Band item)
{
    var band = _context.Bands.Find(id);
    if (band != null)
    {
        band.Name = item.Name;
        band.Genre = item.Genre;

        _context.SaveChanges();
    }
    return id;
}
}
```

The **DataManager** class handles all database operations for our application. The purpose of this class is to separate the actual data operation logic from our API Controller, and to have a central class for handling create, update, fetch and delete (CRUD) operations.

At the moment, the **DataManager** class contains five methods: The **Get()** method gets a specific student record from the database by passing an Id. It uses a LINQ syntax to query the data and returns a **Band** object. The **GetAll()** method gets all band records from the database as you could probably guess based on the method name. The **Add()** method creates a new band record in the database. The **Delete()** method removes a specific band record from database based on the Id. Finally, the **Update()** method updates a specific band record in the database. All methods above use LINQ to query the data from database.

### Register IDataRepository Interface as a Service

Next, we will register the **IDataRepository** as a service. This service will be registered with **Dependency Injection** (DI) during application startup. This would enable our API Controller or other services gain access to the **BandContext** via constructor parameters or properties.

In order for other services to make use of the **BandContext**, we will register it as a service. To enable the service, do following steps:

1. Open Startup.cs and declare the following namespaces below:

```
using Band.API.Models.Repository;  
using Band.API.Models.DataManager;
```

2. Add the following code within ConfigureServices method between **services.AddDbContext()** and **services.AddMvc()** method:

```
services.AddScoped(typeof(IDataRepository<BandModel.Band, int>), typeof(BandManager));
```

## Create an API Controller

Now that our **DataManager** is all set and enabled Dependency Injection for **IDataRepository**, it's time for us to create the API Controller and expose some endpoints for serving data to the client.

Go ahead and right-click on the Controllers folder and then select Add > New File > ASP.NET Core > Web API Controller Class as shown in the figure below:

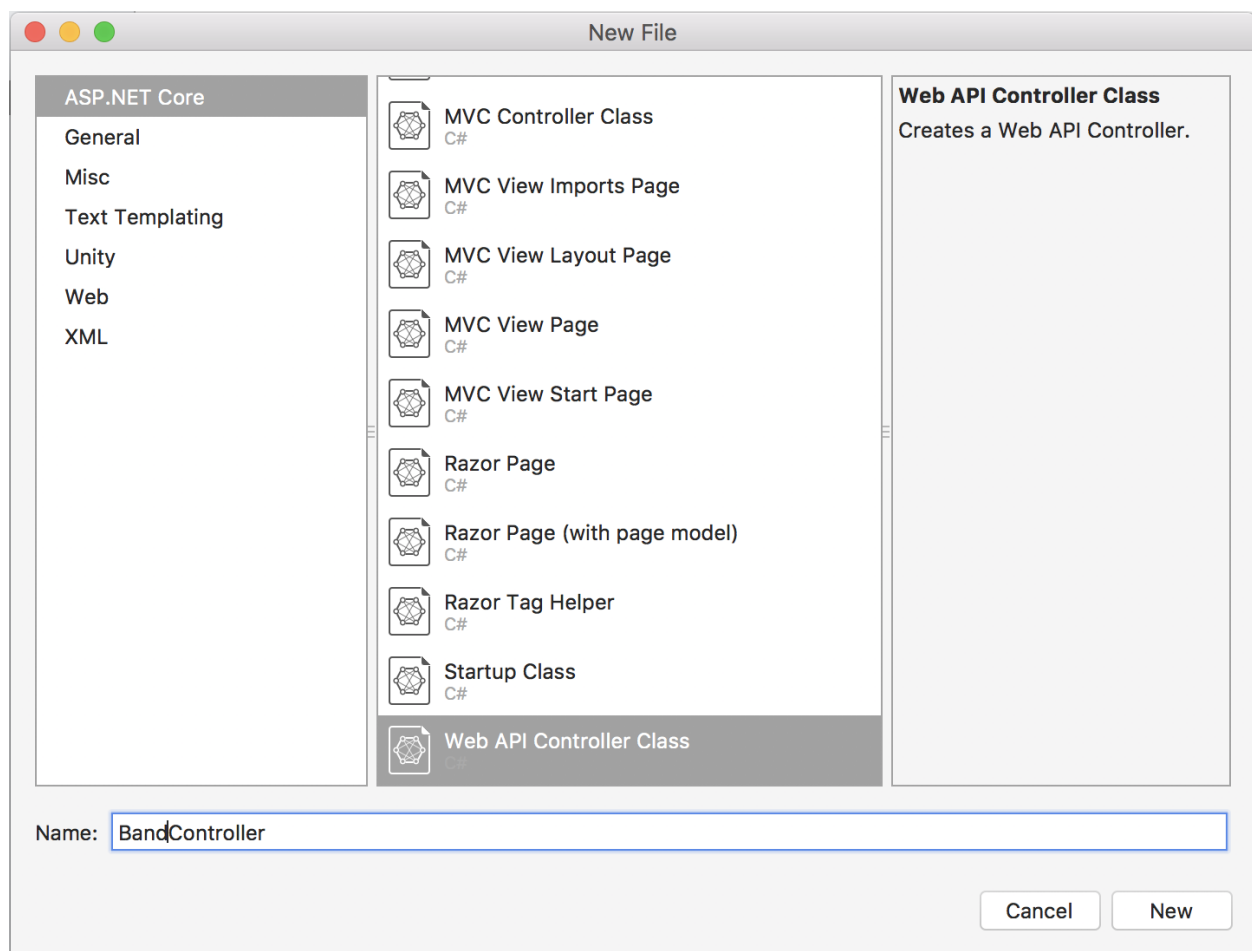


Figure 22: New Web API Controller

Name the class as "**BandsController**" and click **New**.

Now replace the default generated code so it would look similar to the following code below:

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Band.API.Models.Repository;
using BandModel = Band.API.Models;

namespace Band.API.Controllers
{
    [Route("api/[controller]")]
    public class BandsController : Controller
    {
        private IRepository<BandModel.Band, int> _iRepo;
        public BandsController(IRepository<BandModel.Band, int> repo)
        {
            _iRepo = repo;
        }

        // GET: api/bands
        [HttpGet]
        public IEnumerable<BandModel.Band> Get()
        {
            return _iRepo.GetAll();
        }

        // GET api/bands/5
        [HttpGet("{id}")]
        public BandModel.Band Get(int id)
        {
            return _iRepo.Get(id);
        }

        // POST api/bands
        [HttpPost]
        public long Post([FromBody]BandModel.Band band)
        {
            return _iRepo.Add(band);
        }
    }
}
```

```
// POST api/bands
[HttpPut]
public int Put([FromBody]BandModel.Band band)
{
    return _iRepo.Update(band.Id, band);
}

// DELETE api/bands/5
[HttpDelete("{id}")]
public long Delete(int id)
{
    return _iRepo.Delete(id);
}
}
```

Let's see what we just did there.

The BandController derives from the Controller base and by decorating it with the Route attribute enables this class to become a Web API Controller.

If you notice, we have used Controller Injection to subscribed to the IRepository interface. So instead of taking directly to the StudentManager class, we just simply talk to the interface and make use of the methods that are available.

The BandController has five (5) main methods/endpoints. The first Get() method class the GetAll() method from IRepository interface and basically returns all the list of Student from the database. The second Get() method returns a specific Student object based on the Id. Notice that the second Get() method is decorated with [HttpGet("{id}")] , which means append the "id" to the route template. The "{id}" is a placeholder variable for the ID of the Student object. When the second Get() method is invoked, it assigns the value of "{id}" in the URL to the method's id parameter. The Post() method adds a new record to the database. The Put() method updates a specific record from the database. Notice that both Post() and Put() uses a [FromBody] tag. When a parameter has [FromBody], Web API uses the Content-Type header to select a formatter. Finally, the Delete() method removes a specific record from the database.



## Enable CORS

Now that we have our API ready, the final step that we are going to do on this project is to enable Cross-Origin Resource Sharing (a.k.a CORS). We need this configuration so other clients that is hosted in a different domain/ports can access the API endpoints.

To enable CORS in ASP.NET Core Web API, follow the steps below:

- (1) Open **Startup.cs** file and on **ConfigureServices()** method, add the following code below before **services.AddMvc()**:

```
services.AddCors(opts =>{
    opts.AddPolicy("AllowAll", builder =>
    {
        builder.AllowAnyOrigin()
            .AllowAnyMethod()
            .AllowAnyHeader()
            .AllowCredentials();
    });
});
```

- (2) Then on **Configure()** method, copy the following code below before **app.UseMvc()**:

```
app.UseCors("AllowAll");
```

That's it. Note that just for the purpose of this demo, we allow all clients to access the API. In real-world applications, it's recommended to set the allowable origins, methods, headers and credentials for your APIs. For more information about CORS, see: <https://docs.microsoft.com/en-us/aspnet/core/security/cors?view=aspnetcore-2.1>

## Testing the API Endpoints

To ensure that our API endpoints are working, it's always recommended to do an initial test at the early stage of development. You can download Postman and use that as a tool to test out APIs.

Here are a few test I did in Postman:

POST cURL:

```
curl -X POST \
  https://localhost:5001/api/bands \
  -H 'Cache-Control: no-cache' \
  -H 'Content-Type: application/json' \
```

```
-H 'Postman-Token: a025f464-3501-4fbc-a556-51ed1f4abe0c' \  
-d '{  
    "Name": "Alice in Chains",  
    "Genre": "Heavy Metal"  
'
```

Output:

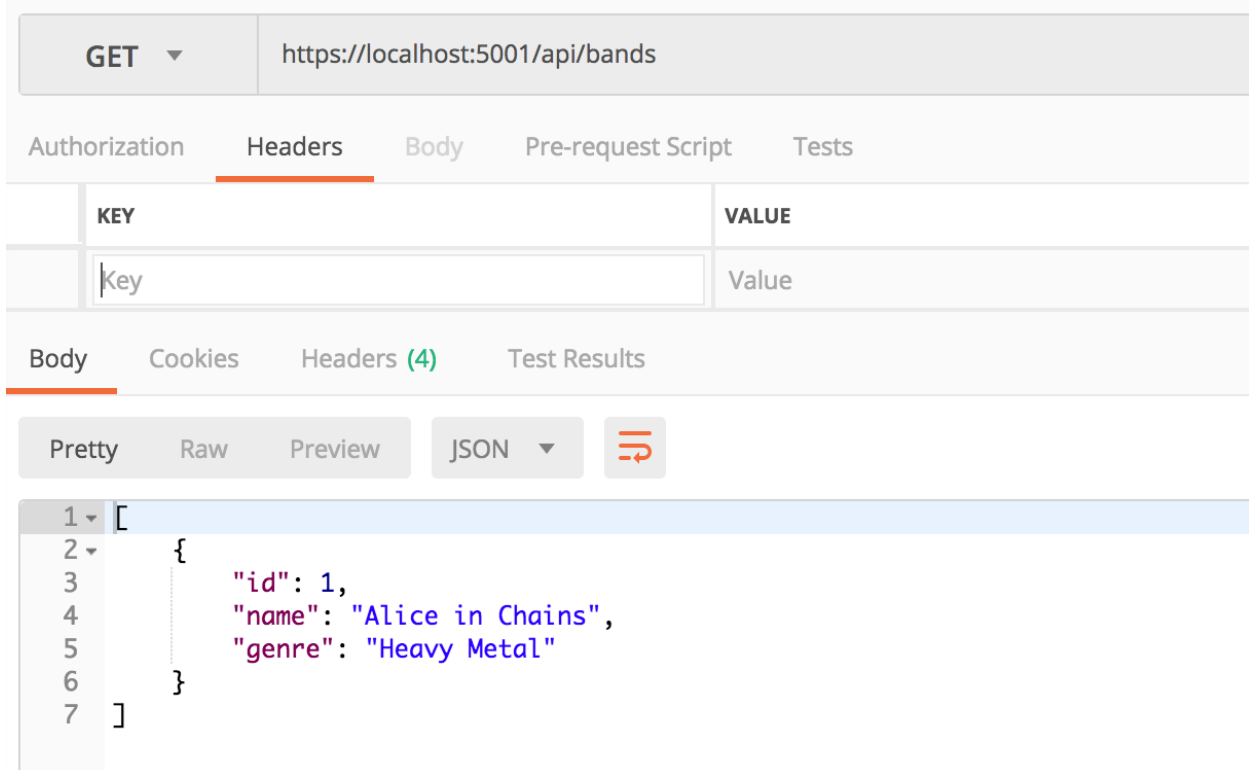


Figure 23: POST result

GET cURL:

```
curl -X GET \  
https://localhost:5001/api/bands \  
-H 'Cache-Control: no-cache' \  
-H 'Postman-Token: 40bbcadd-c5fe-43da-8c4f-fb89bc63f9b8'
```

## Output



The screenshot shows a REST client interface. At the top, the method is set to **GET** and the URL is `https://localhost:5001/api/bands`. Below this, there are tabs for **Authorization**, **Headers**, **Body**, **Pre-request Script**, and **Tests**. The **Headers** tab is selected, showing a table with two columns: **KEY** and **VALUE**. The first row has the key `key` and the value `Value`. Below the headers, there are tabs for **Body**, **Cookies**, **Headers (4)**, and **Test Results**. The **Body** tab is selected, showing a **Pretty** view of the JSON response. The response is a JSON array containing one object: `[{"id": 1, "name": "Alice in Chains", "genre": "Heavy Metal"}]`. The JSON is formatted with line numbers 1 through 7 on the left.

```
1 [
2   {
3     "id": 1,
4     "name": "Alice in Chains",
5     "genre": "Heavy Metal"
6   }
7 ]
```

Figure 24: GET result

## Dockerize ASP.NET Core Web API Application

At this point, the Web API application is still running on the local machine. Our goal is to deploy it on a Docker container so other client application can access to it.

### Enable Docker Support

To enable docker support, right-click on the Web API project and then select **Add > Add Docker Support** as shown in the figure below:

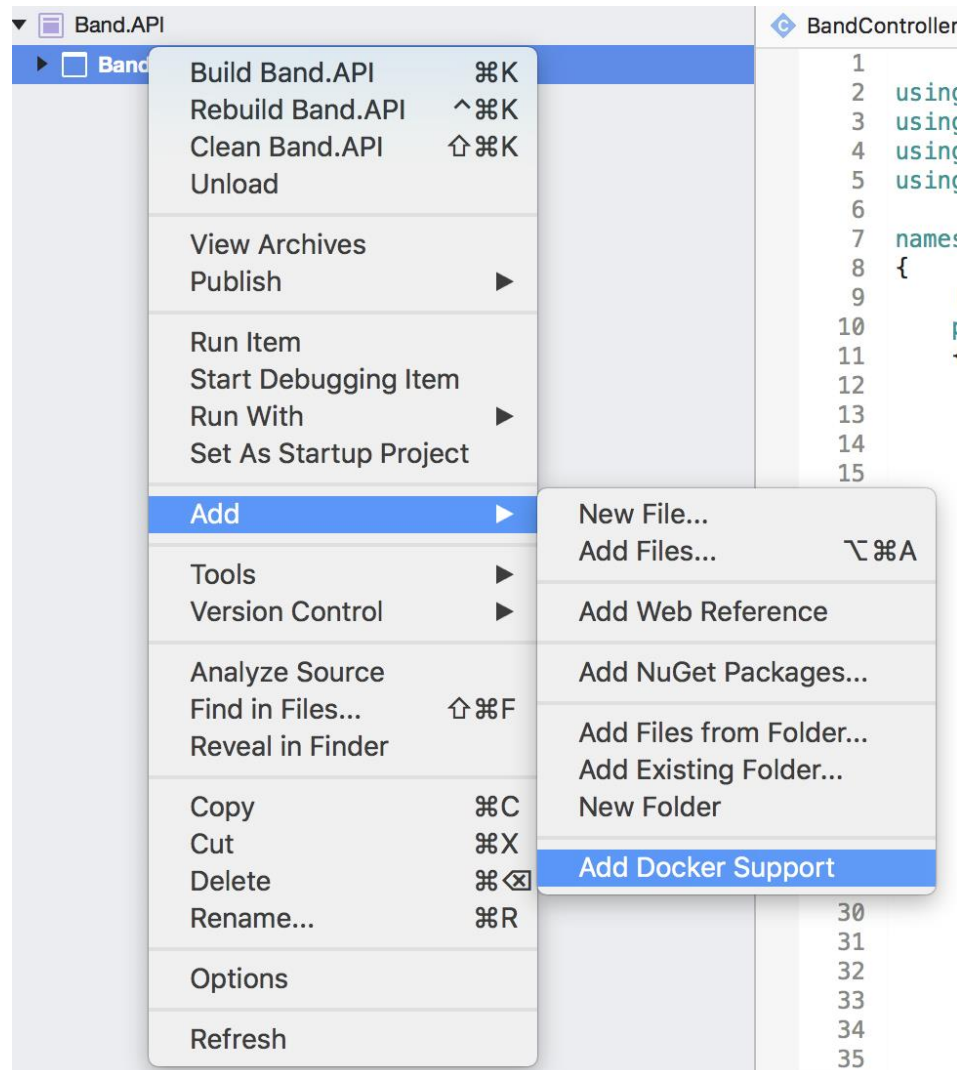


Figure 25: Add Docker support

That should add the following file to the project:

- **Dockerfile** - contains the instruction to build the docker image. For more information, see: <https://docs.docker.com/engine/reference/builder/>

- **Docker-Compose** - tool for defining and running multi-container Docker applications.  
For more information, see: <https://docs.docker.com/compose/>

#### Docker File

Here's what the Docker file looks like for this demo:

```
FROM microsoft/dotnet:2.1-aspnetcore-runtime AS base
WORKDIR /app
EXPOSE 5555
```

```
FROM microsoft/dotnet:2.1-sdk AS build
WORKDIR /src
COPY Band.API/Band.API.csproj Band.API/
RUN dotnet restore Band.API/Band.API.csproj
COPY . .
WORKDIR /src/Band.API
RUN dotnet build Band.API.csproj -c Release -o /app
```

```
FROM build AS publish
RUN dotnet publish Band.API.csproj -c Release -o /app
```

```
FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "Band.API.dll"]
```

The instruction above contains the recipe for creating a final Docker image. What it does is it initializes multi-stage build images and sets the base image for subsequent commands. In this case, it tells Docker to base our project on aspnetcore-runtime image and run from there by exposing port 5555. The subsequent instructions tells the Docker to pull the required dependencies needed for this app to run, build it and publish as a container . For more information about the commands within it, see: <https://docs.docker.com/engine/reference/builder/#usage>

## Managing Containers

Remember that our SQL Server database is running on separate container. What we need right now is to enable the Web API app to connect with the database when they are both inside Docker containers. Unfortunately, connecting to the already running container using the **ConnectionString** Server value of **127.0.0.0,1433** won't work.

While it may be possible to link an existing container and run it in conjunction with your Web API container as demonstrated on my previous article [here](#), still it's not an ideal way to handle it especially if you are working with multi-container applications which depends on each other.

## Docker-Compose File

Thankfully, Visual Studio 2017 versions 15.7 or earlier support Docker Compose as the sole container orchestration solution. The Docker Compose artifacts are automatically added for us when we enabled Docker Support as described in previous section.

The Docker Compose gives us a more convenient way to manage multiple Docker containers in one place. Here's the **docker-compose.yml** file looks like for this app:

```
version: '3.4'

services:
  band.api:
    image: ${DOCKER_REGISTRY}band
    container_name: band.api
    ports:
      - "5555:80"
    build:
      context: .
      dockerfile: Band.API/Dockerfile
    environment:
      - ConnectionString=Server=band.data;Database=Band;User Id=sa;Password=SuperSecret1!
    depends_on:
      - band.data

  band.data:
    image: mcr.microsoft.com/mssql/server:2017-latest
    container_name: band.data
    environment:
      - SA_PASSWORD=SuperSecret1!
      - ACCEPT_EULA=Y
    ports:
      - "1433:1433"
```

The root key in this file is “**services**”. Under that key you define the services you want to deploy and run when you execute the docker-compose using a command or when you deploy from Visual Studio by using this docker-compose.yml file. In this case, the docker-compose.yml file has multiple services defined, as described in the following list.

- **band.api** – defines the container for the ASP.NET Core Web API project. The **ports** attribute exposes **5555** for accessing the app externally and exposes **80** for accessing the app internally inside docker. In other words, it provides the network plumbing so we can talk to services running in containers by mapping to ports on the host. The **build** attribute contains the path to the **dockerfile** to use as well as the context docker should use to build the container from. The **environment** attribute defines variable named **ConnectionString** with the connection string to be used by Entity Framework to access the SQL Server instance inside Docker container. This configuration overrides the **ConnectionString** value we defined in the **appsettings.json** in the Web API project. Also notice that it uses the value **band.data** as the server value instead of the IP address. This is the beauty of the docker compose as it enables us to connect between containers without having to worry about internal IP and ports used to accessing between containers. Finally, the **depends\_on** attribute defines a dependency to other containers. In this case the **band.data**. This tells Docker to run the **band.data** service first before running the **band.api** service.
- **band.data** – defines how to run SQL Server for Linux docker container. This docker compose configuration is the same as running the image using the docker run command that we used previously using this command:

```
docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD=SuperSecret1!' -p 1433:1433 --name sql17_linux -d mcr.microsoft.com/mssql/server:2017-latest
```

## Execute the Docker Compose

To build and deploy the containers defined in docker-compose.yaml file, do follow the following steps:

1. **Clean** and the **Build** the Web API project to **release mode**
2. **Open** the **Terminal console** and make sure you are running it in the location where your Solution file is located.
3. **Stop** the existing running container for SQL Server for Linux using the command: **docker stop sql17\_linux**
4. Then run: **docker ps** to ensure that the container named “**sql17\_linux**” will be stopped.
5. Next, run: **docker-compose up -d**

You should be able to see something like in the figure below when the build is successful.

```

vdurano-mbp13ts:band.api vdurano_srg$ docker stop sql17_linux
[sql17_linux]
vdurano-mbp13ts:band.api vdurano_srg$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
vdurano-mbp13ts:band.api vdurano_srg$ docker-compose up -d
WARNING: The DOCKER_REGISTRY variable is not set. Defaulting to a blank string.
Starting band.data ... done
Creating band.api ... done
vdurano-mbp13ts:band.api vdurano_srg$

```

Figure 26: Composing containers

## Testing the Dockerized Web API

At this point, the Web API app and database should be up and running inside the Docker container. We can access it by using the external port **5555** that we have defined in the **dockerfile** and **dockerfile -compose.yaml**. Fire off a browser or Postman and then access the following URL:

- <http://localhost:5555/api/bands>

That should output the following data:

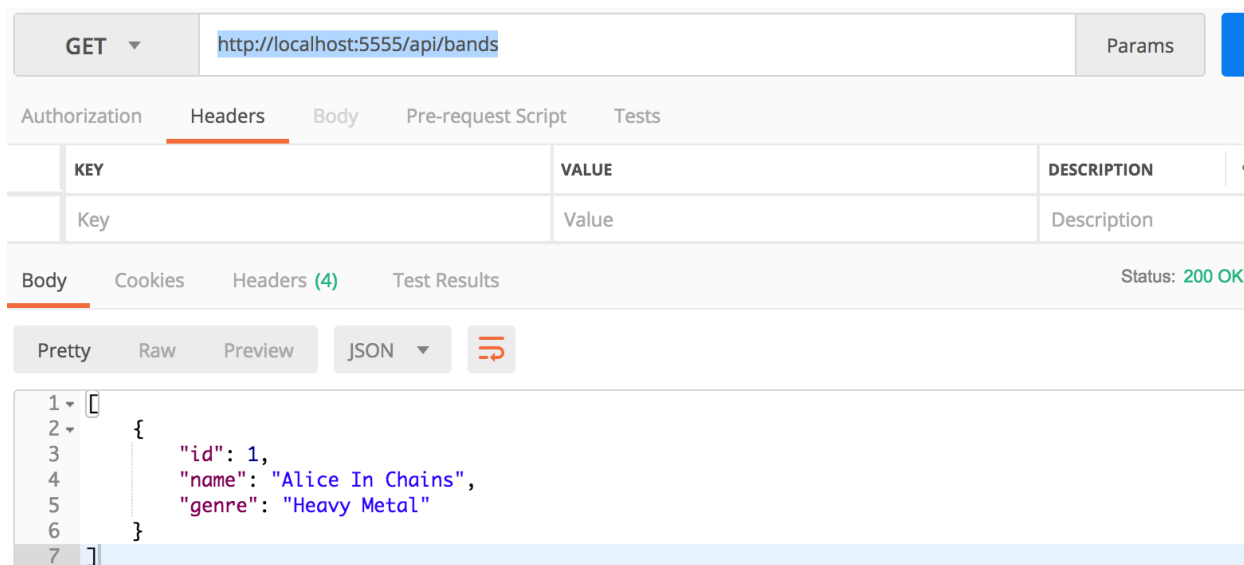


Figure 27: Dockerize Web API output

If you've made this far, Congrats! You've just ran an ASP.NET Core and SQL Server inside Docker on your MAC machine. Pretty cool eh?

But we're not done yet! We are still going to need a UI to consume the API using a server-side SPA framework – Blazor!



## Create Your First Blazor App on Docker

*This demo is intended for Blazor 0.6.0 as this is the latest version of as of this time of writing. Any code, file structure and syntax may change on future versions. Check out the official repo for future release updates here: <https://github.com/aspnet/Blazor>*

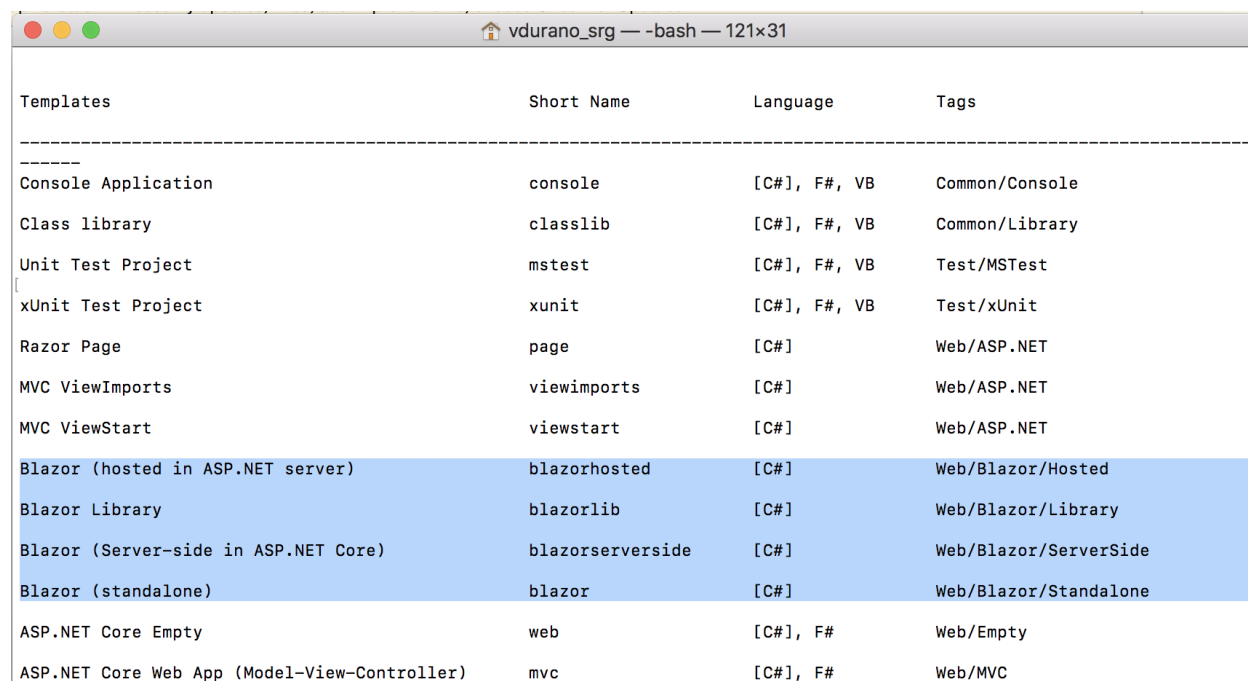
Without further ado, let's get cracking. As of this time of writing, Visual Studio 2017 Community v7.6 for MAC doesn't include Blazor templates so you need to manually install them using the dotnet new CLI just like in the following:

```
dotnet new -i Microsoft.AspNetCore.Blazor.Templates::*
```

The command above should install the templates for building and running Blazor apps. You can verify if the installation is successful by running the following command:

```
dotnet new --list
```

which should give you the following results:



Templates	Short Name	Language	Tags
-----			
Console Application	console	[C#], F#, VB	Common/Console
Class library	classlib	[C#], F#, VB	Common/Library
Unit Test Project	mstest	[C#], F#, VB	Test/MSTest
xUnit Test Project	xunit	[C#], F#, VB	Test/xUnit
Razor Page	page	[C#]	Web/ASP.NET
MVC ViewImports	viewimports	[C#]	Web/ASP.NET
MVC ViewStart	viewstart	[C#]	Web/ASP.NET
Blazor (hosted in ASP.NET server)	blazorhosted	[C#]	Web/Blazor/Hosted
Blazor Library	blazorlib	[C#]	Web/Blazor/Library
Blazor (Server-side in ASP.NET Core)	blazorserverside	[C#]	Web/Blazor/ServerSide
Blazor (standalone)	blazor	[C#]	Web/Blazor/Standalone
ASP.NET Core Empty	web	[C#], F#	Web/Empty
ASP.NET Core Web App (Model-View-Controller)	mvc	[C#], F#	Web/MVC

Figure 28: ASP.NET Core templates with Blazor

Note that the Blazor templates will not be available in the list of .NET Core Apps when you create a new project in Visual Studio for MAC even if you've added them from the command line (CLI). This means that in order for us to create a new Blazor project, we need to use the command line too.

## Create a Runtime Docker Container for Blazor

Since our goal is to run Blazor on a Docker container, then we are going to take a different route. This simply means that we are going to build a Docker Container runtimes for Blazor first since there's no existing one that I am aware of as of this moment.

In order to build a Docker container runtime for Blazor, we need to make sure the exact supported version of .NET Core SDK that corresponds to the Blazor version that you are using. In this example, we are going to use **Blazor 0.6.0** which requires **.NET Core 2.1 SDK (2.1.402 or later)**. Once we've identified that head over to the official repository of .NET Core Docker containers here: <https://hub.docker.com/r/microsoft/dotnet/>

The link should take you to page where it lists all the available official images for .NET Core and ASP.NET Core for Linux and Windows Nano Server. The figure below shows the latest versions of common tags:

### Full Description

### Latest Version of Common Tags

The following tags are the latest stable versions of the most commonly used images. The complete set of tags is listed further down.

- `2.1-sdk`
- `2.1-aspnetcore-runtime`
- `2.1-runtime`

Figure 29: .NET Core docker images

Click on **2.1-sdk** and it should take you to the GitHub repo for the corresponding **Dockerfile**.

**Copy** the content of the **Dockerfile** from the repo. Create a new **Dockerfile** on your local drive and paste the content there. Here's what my **Dockerfile** looks like:

```
FROM buildpack-deps:stretch-scm

# Install .NET CLI dependencies
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
        libc6 \
        libgcc1 \
        libgssapi-krb5-2 \
        libicu57 \
```

```
liblttng-ust0 \  
libssl1.0.2 \  
libstdc++6 \  
zlib1g \  
&& rm -rf /var/lib/apt/lists/*  
  
# Install .NET Core SDK  
ENV DOTNET_SDK_VERSION 2.1.403  
  
RUN curl -SL --  
output dotnet.tar.gz https://dotnetcli.blob.core.windows.net/dotnet/Sdk/$DOTNET_SDK_VERSION/dotnet-sdk-$DOTNET_SDK_VERSION-linux-x64.tar.gz \  
&& dotnet_sha512='903a8a633aea9211ba36232a2decb3b34a59bb62bc145a0e7a90ca46dd37bb6c2da02bcbe2c50c17e08cdf8e48605c0f990786faf1f06be1ea4a4d373beb8a9' \  
&& sha512sum dotnet.tar.gz \  
&& echo "$dotnet_sha512 dotnet.tar.gz" | sha512sum -c - \  
&& mkdir -p /usr/share/dotnet \  
&& tar -xzf dotnet.tar.gz -C /usr/share/dotnet \  
&& rm dotnet.tar.gz \  
&& ln -s /usr/share/dotnet/dotnet /usr/bin/dotnet \  
&& dotnet new -i Microsoft.AspNetCore.Blazor.Templates  
  
# Configure Kestrel web server to bind to port 80 when present  
ENV ASPNETCORE_URLS=http://+:80 \  
# Enable detection of running in a container  
DOTNET_RUNNING_IN_CONTAINER=true \  
# Enable correct mode for dotnet watch (only mode supported in a container)  
DOTNET_USE_POLLING_FILE_WATCHER=true \  
# Skip extraction of XML docs - generally not useful within an image/container - helps performance  
NUGET_XMLDOC_MODE=skip  
  
# Trigger first run experience by running arbitrary cmd to populate local package cache  
RUN dotnet help
```

The script above contains the instructions to install and configure .NET Core within Docker container. One important line there is I added the (dotnet new -i Microsoft.AspNetCore.Blazor.Templates) command to get the Blazor templates so we don't need to worry about manually installing them and instead just focus directly on building Blazor apps.

Now, save the Dockerfile and then open the Terminal console where you put the Docker file and then run the following command:

```
docker build -t blazor060:core2.1.402 .
```

The command above creates a local Docker image named **blazor060:core2.1.402** . Note that the name is configurable using the following format **<Your Image Name> : < Tag >** .

Once the build is successful, then you can run the command docker images to verify if it's created on your local machine.

Note: If you don't want to build and create your own Docker runtime container for Blazor, you can also use the existing Docker image that've created for this demo here:

<https://hub.docker.com/r/proudmonkey30/blazor060/>

You can then do:

```
docker pull proudmonkey30/blazor060
```

## Create a Blazor App

At this point, we are now ready to create the Blazor app. Go ahead and create a new directory in your local drive to where you would want to store your Blazor app. In this example, I created a folder on my Mac at **/<User>/repo/Blazor.Spa**.

After that, open your Terminal console and run the following command:

```
docker run -it --rm --name blazorapp -p 5600:80 -v  
/Users/vdurano_srg/repo/Blazor.Spa:/BlazorApp -w /BlazorApp blazor060:core2.1.402
```

The command above runs the Docker container in interactive mode. The **--rm** command tells Docker to remove the container after exiting. The **--name** specifies the container name. The **-p** command sets the port for the running container, in this case we set an external port **5600** and internal port **80**. The **-v** command attaches a volume to the container, in this case we set the volume to the local directory where we dump Blazor related files and then maps it to the internal Docker container location named **"/BlazorApp"**. The **-w** command specifies a working directory which in this case, we want to automatically set the working directory to **"/Blazor"** after executing the command. Finally, we based the container on the image we've created earlier to download and restore the dependencies.

Now open a new Terminal console window and run the command `docker ps`, and it should result to something like this:

```

vdurano_srg — -bash — 139x14
Last login: Fri Oct 5 13:52:53 on ttys004
vdurano-mbp13ts:~ vdurano_srg$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
94b947142a6f       blazor060:core2.1.402  "bash"             About a minute ago  Up About a minute  0.0.0.0:5600->80/tcp  blazorapp
vdurano-mbp13ts:~ vdurano_srg$

```

Figure 30: List of containers

Go back to the first Terminal console window where we execute the docker run command and then do:

dotnet new blazor -o .

The command above should pull the Blazor templates and create the default files to the local directory that we set as volume as shown in the figure below:

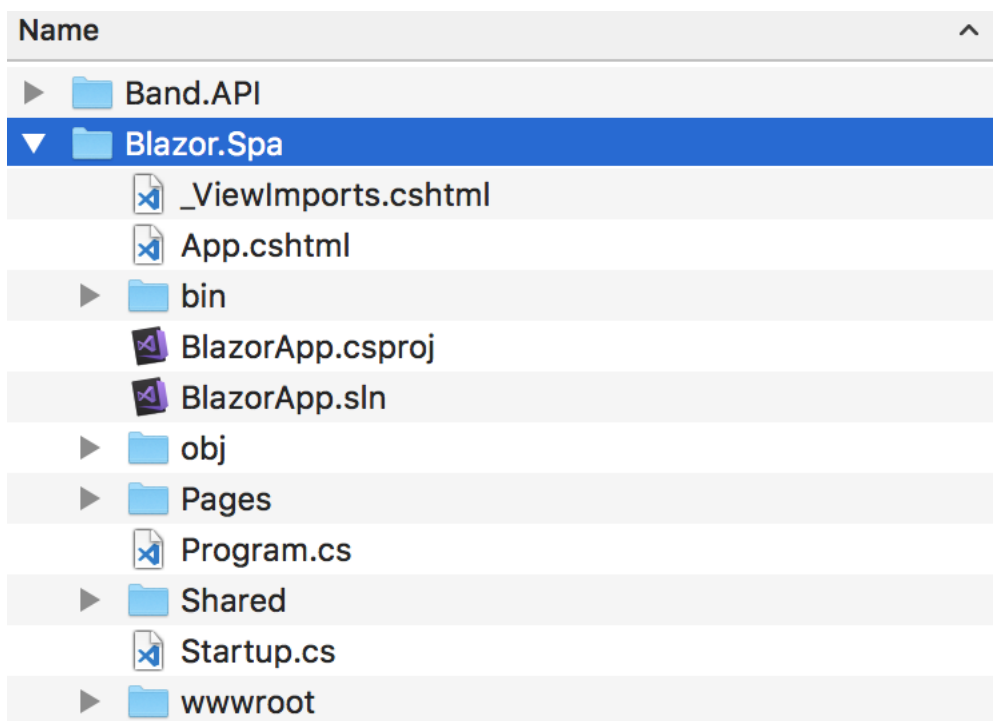


Figure 31: Blazor app directory

Now, let's try to run the application by running:

dotnet run

When successful, it should show something like this in the console:

```
BlazorApp — docker run -it --rm --name blazorapp -p 5600:80 -v ~/repo/Blazc
Installing Microsoft.AspNetCore.SpaServices.Extensions 2.1.0.
Installing Microsoft.AspNetCore.Routing 2.1.0.
Installing Microsoft.AspNetCore.SignalR 1.0.0.
Installing Microsoft.AspNetCore.Server.IISIntegration 2.1.0.
Installing Microsoft.AspNetCore.SignalR.Protocols.MessagePack 1.0.0.
Installing Microsoft.AspNetCore.StaticFiles 2.1.0.
Installing Microsoft.AspNetCore.Server.Kestrel.Https 2.1.0.
Installing Microsoft.Extensions.FileProviders.Physical 2.1.0.
Installing Microsoft.AspNetCore.Server.Kestrel 2.1.0.
Installing Mono.Cecil 0.10.0-beta7.
Installing Microsoft.AspNetCore.Blazor 0.6.0.
Installing Microsoft.AspNetCore.Blazor.Browser 0.6.0.
Restore completed in 11.16 sec for /BlazorApp/BlazorApp.csproj.
Installing Microsoft.AspNetCore.Blazor.Analyzers 0.6.0.
Installing Microsoft.AspNetCore.Blazor.Build 0.6.0.
Generating MSBuild file /BlazorApp/obj/BlazorApp.csproj.nuget.g.props.
Generating MSBuild file /BlazorApp/obj/BlazorApp.csproj.nuget.g.targets.
Restore completed in 13.58 sec for /BlazorApp/BlazorApp.csproj.

Restore succeeded.

[root@70756dc8b14e:/BlazorApp# dotnet run
Hosting environment: Production
Content root path: /BlazorApp
Now listening on: http://[::]:80
Application started. Press Ctrl+C to shut down.
```

Figure 32: Running Blazor app inside docker

Remember that we are running the application within Docker container that's why you see that's it's listening to port **80**. However, we can test that out externally using port **5600**. To see that action, go ahead and fire up a chrome browser and the navigate to <http://localhost:5600>. It should display something like this in your browser:

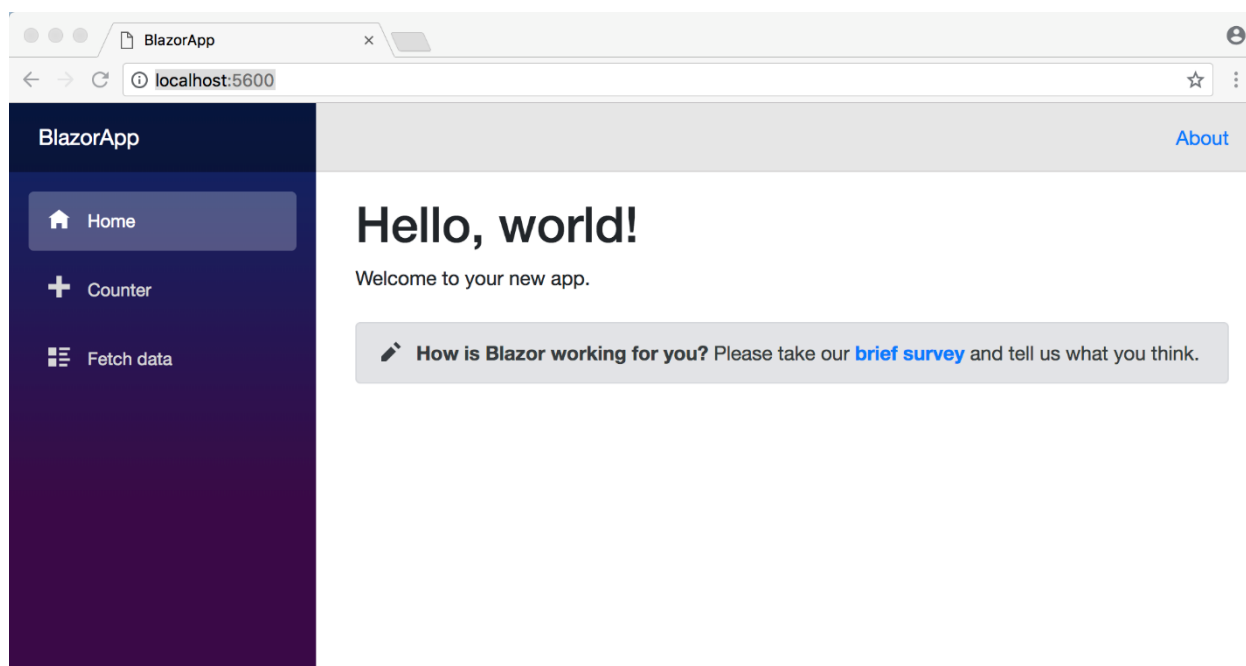


Figure 33: Blazor first run

Congrats! You just had your first Blazor app running on MAC! Pretty cool, huh? Now let's have some fun and modify the app to work with real data from our Dockerized Web API.

Now go back to the Terminal console and press **CTRL + C** to stop the running application.

Just to give you a quick heads-up that I'm not going to cover the details on how the app is implemented in Blazor here. I'm going to cover that in a separate article. The main goal of this article is to see how we can run, build and deploy apps in Docker containers and connect between them.

Okay let's keep moving, open up the Blazor app that we've created previously. To open multiple instance of Visual Studio on Mac, see:

<https://visualstudio.microsoft.com/vs/support/mac/open-multiple-solutions-instances-visual-studio-mac/>

## Add a New Component

Let's add a new **component** by right-clicking on "**Pages**" folder and select **Add > New Item**. Select **ASP.NET Core** from the left panel, then select "**Razor Page**" from templates panel as shown in the figure below:

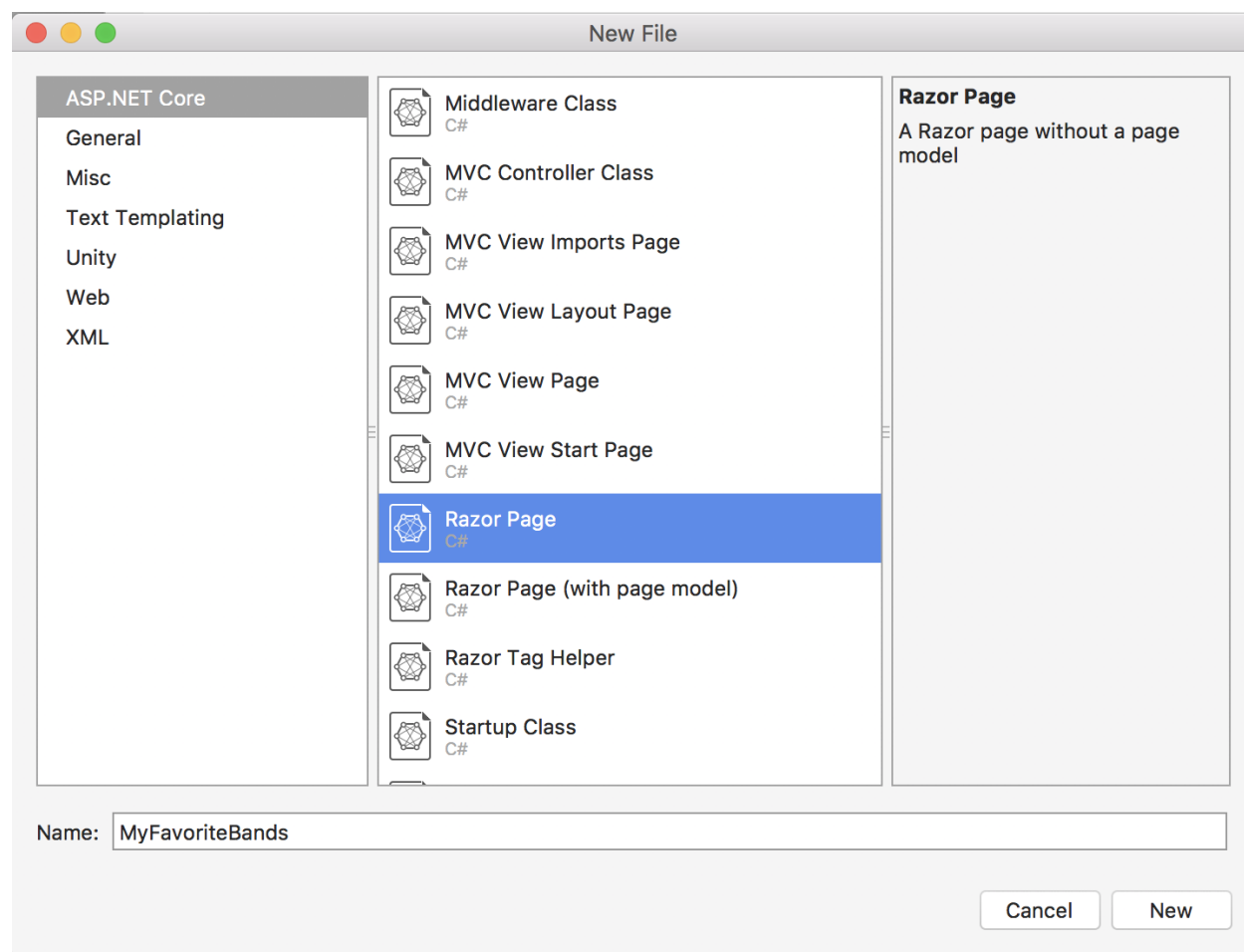


Figure 34: Create new Blazor component

Name the file as "**MyFavoriteBands**" and then click New. Copy the following code below:

```
@page "/bands"
@using Microsoft.AspNetCore.Blazor;
@using Microsoft.JSInterop;
@using System.Threading.Tasks;
@inject HttpClient Http
```

```
<h1>All-time Favorite Bands</h1>
```



```

<div>
  <div class="row">
    <div class="col-sm-1">
      <p>Name:</p>
    </div>
    <div class="col-sm-4">
      <input id="txtBandName" placeholder="Band Name" bind="@_bandName" />
    </div>
  </div>
<br/>
<div class="row">
  <div class="col-sm-1">
    <p>Genre:</p>
  </div>
  <div class="col-sm-4">
    <input id="txtBandGenre" placeholder="Band Genre" bind="@_bandGenre" />
  </div>
</div>
<br/>
<div class="row">
  <div class="col-sm-1">
    <button class="btn btn-
info" id="btnAdd" onclick=@(async () => await Add())>Add</button>
  </div>
</div>
<br/>
</div>

@if (bands == null)
{
  <p><em>Loading your favorite bands of all time...</em></p>
}
else
{
  @if (bands.Count > 0)
  {
    <table class='table table-striped table-bordered table-hover table-
condensed' style="width:80%;">
      <thead>
        <tr>
          <th style="width: 40%">Band Name</th>
          <th style="width: 20%">Band Genre</th>
          <th style="width: 20%">Edit</th>
          <th style="width: 20%">Delete</th>

```

```

        </tr>
    </thead>
    <tbody>
        @foreach (var band in bands)
        {
            <tr>
                <td>
                    <span id="spnName_@band.Id">@band.Name</span>
                    <input id="txtName_@band.Id" bind="@_bandNameUpdate" style="display:none;"></input>
                </td>
                <td>
                    <span id="spnGenre_@band.Id">@band.Genre</span>
                    <input id="txtGenre_@band.Id" bind="@_bandGenreUpdate" style="display:none;"></input>
                </td>
                <td>
                    <button id="btnEdit_@band.Id" class="btn btn-primary" onclick=@(async() => await Edit(band.Id, band.Name, band.Genre))>Edit</button>
                    <button id="btnUpdate_@band.Id" style="display:none;" class="btn btn-success" onclick=@(async () => await Update(band.Id))>Update</button>
                    <button id="btnCancel_@band.Id" style="display:none;" class="btn btn-primary" onclick=@(async () => await Cancel(band.Id))>Cancel</button>
                </td>
                <td><button class="btn btn-danger" onclick=@(async () => await Delete(band.Id))>Delete</button></td>
            </tr>
        }
    </tbody>
</table>
}
}

```

@functions {

```

public class BandDTO
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Genre { get; set; }
}

```

```

string _bandAPIUri = "http://localhost:5555/api/bands";

```

```
string _bandName;
string _bandGenre;
string _bandNameUpdate;
string _bandGenreUpdate;

IList<BandDTO> bands = new List<BandDTO>();

protected override async Task OnInitAsync()
{
    await RefreshView();
}

private async Task RefreshView()
{
    bands = await Http.GetJsonAsync<BandDTO[]>(_bandAPIUri);
    StateHasChanged();
}

public async Task Add()
{
    if (!string.IsNullOrEmpty(_bandName))
    {
        await Http.SendJsonAsync(HttpMethod.Post, _bandAPIUri, new BandDTO
        {
            Name = _bandName,
            Genre = _bandGenre
        });

        _bandName = string.Empty;
        _bandGenre = string.Empty;

        await RefreshView();
    }
}

public async Task Update(int id)
{
    if (!string.IsNullOrEmpty(_bandNameUpdate))
    {
        await Http.SendJsonAsync(HttpMethod.Put, _bandAPIUri, new BandDTO
        {
            Id = id,
            Name = _bandNameUpdate,
            Genre = _bandGenreUpdate
        });
    }
}
```

```

});

        await RefreshView();
        await JSRuntime.Current.InvokeAsync<bool>("toggleUIView", new object[] { id.ToString()
, "", "", false });

    }
}

public async Task Delete(int id)
{
    await Http.DeleteAsync($"{_bandAPIUri}/{id}");
    await RefreshView();
}

public async Task Edit(int id, string bandName, string bandGenre)
{
    await JSRuntime.Current.InvokeAsync<bool>("blazorAppJS.toggleUIView", new object[] { id
.ToString(), bandName, bandGenre, true });
}

public async Task Cancel(int id)
{
    await JSRuntime.Current.InvokeAsync<bool>("blazorAppJS.toggleUIView", new object[] { id
.ToString(), "", "", false });
}

}

```

The component above defines the UI and the corresponding UI code logic using Razor and C# syntax to perform a basic CRUD operations in the page by utilizing REST API's. Notice that the **bandAPIUri** contains the external/public facing URL endpoint of the Web API project that we deployed in Docker.

### Modify the Index.html File

Next, navigate to **wwwroot > index.html** and then copy the following JavaScript function after **blazor.webassembly.js** script reference:

```

<script>
    window.blazorAppJS = {
        toggleUIView: function(id, name, genre, show){
            if(show){
                var txtName = document.getElementById("txtName_" + id);

```

```
document.getElementById("spnName_" + id).style.display = "none";
txtName.style.display = "";
txtName.value = name;
txtName.focus();

var txtGenre = document.getElementById("txtGenre_" + id);
document.getElementById("spnGenre_" + id).style.display = "none";
txtGenre.style.display = "";
txtGenre.value = genre;

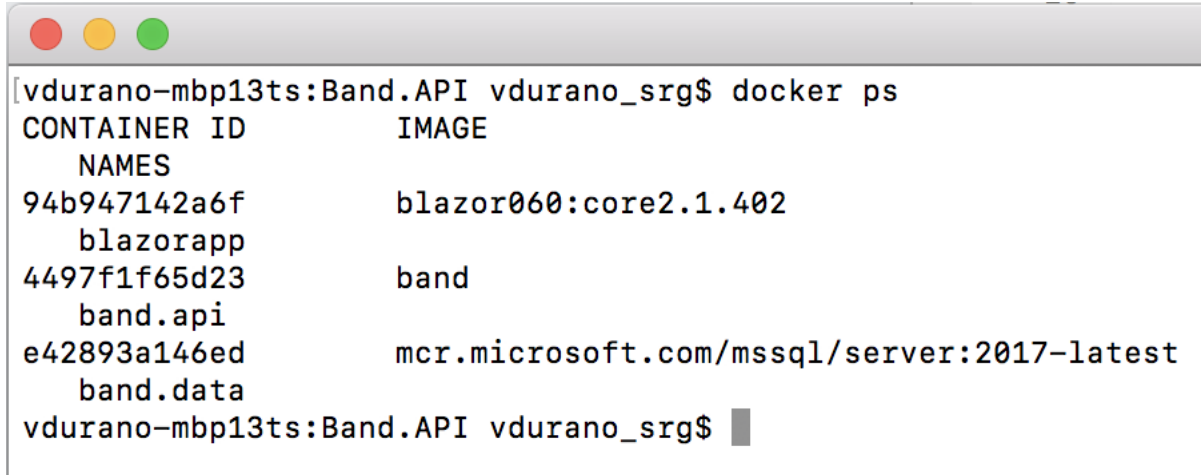
document.getElementById("btnEdit_" + id).style.display = "none";
document.getElementById("btnUpdate_" + id).style.display = "";
document.getElementById("btnCancel_" + id).style.display = "";
}
else {
    document.getElementById("spnName_" + id).style.display = "";
    document.getElementById("txtName_" + id).style.display = "none";
    document.getElementById("spnGenre_" + id).style.display = "";
    document.getElementById("txtGenre_" + id).style.display = "none";
    document.getElementById("btnEdit_" + id).style.display = "";
    document.getElementById("btnUpdate_" + id).style.display = "none";
    document.getElementById("btnCancel_" + id).style.display = "none";
}
}
};
</script>
```

The **toggleUIView** function contains the logic for toggling the Edit, Update and Cancel buttons.

For more information about Blazor, see: <https://learn-blazor.com/getting-started/>

## Run the Blazor App

Before we start the test, be sure to run the command `docker ps` to ensure that all the containers we need are up and running. You should be able to see something like this:



```
[vdurano-mbp13ts:Band.API vdurano_srg$ docker ps
CONTAINER ID        IMAGE
NAMES
94b947142a6f        blazor060:core2.1.402
blazorapp
4497f1f65d23        band
band.api
e42893a146ed        mcr.microsoft.com/mssql/server:2017-latest
band.data
vdurano-mbp13ts:Band.API vdurano_srg$
```

Figure 35: List of Docker containers

As you can see, the `blazorapp`, **band.api** and **band.data** Docker containers are up and running. It's safe for us to test the application.

Now, open the Terminal console to the location where your BlazorApp Solution project is located and then do the following:

- (1) `dotnet build`
- (2) `dotnet run`

Here's a screenshot of Blazor app running on Docker Container on my Mac:

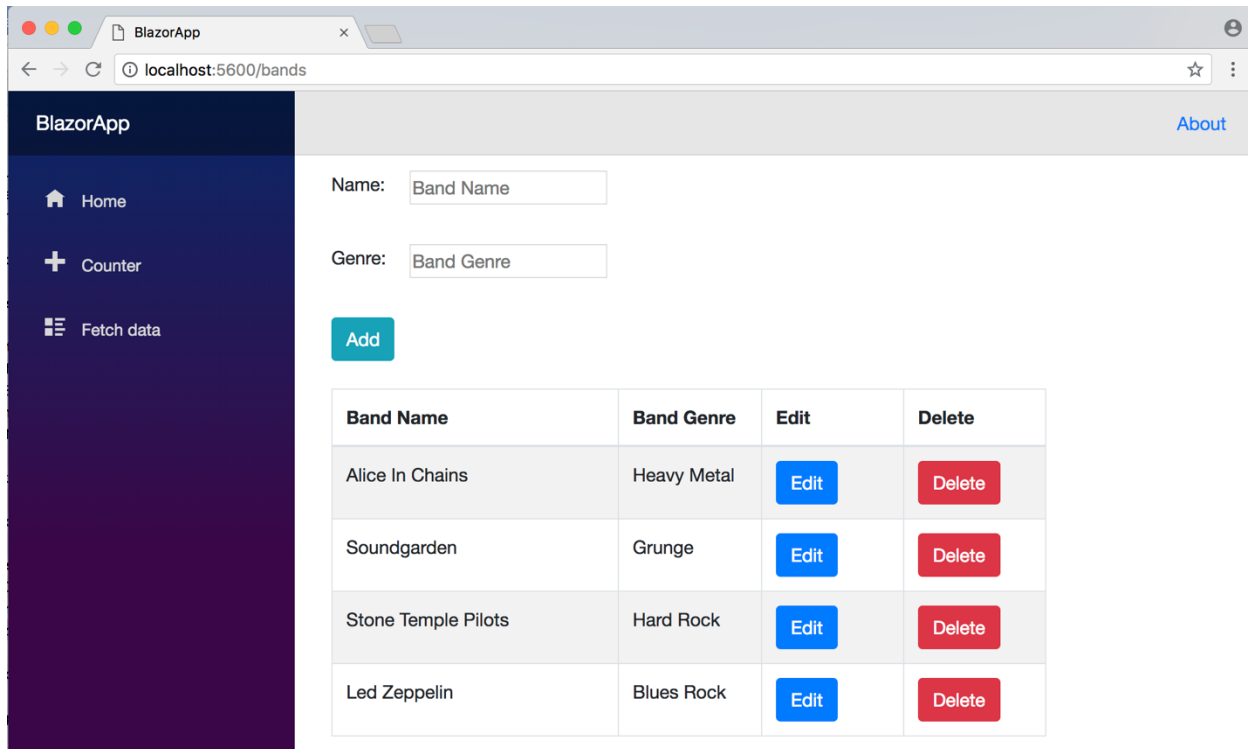


Figure 36: Blazor page output

That's it! goals achieved! woot!

## GitHub Repo

- <https://github.com/proudmonkey/ASP.Net-Core-and-Blazor-on-MAC>

## Summary

In this book we've learned a lot of things, starting from setting up the development environment, creating REST APIs and data-driven SPA app from scratch, down to deploying ASP.NET Core Web API and Blazor apps on a Docker containers. To summarize, here's what we've learned:

- The goal of what we are trying to achieve
- Setting up the development environment
- Configuring a database using SQL Server for Linux
- Using Valentina Studio for managing database
- Creating an ASP.NET Core Web API application
- Dockerizing the Web API application
- Creating your first Blazor application
- Connecting all applications together from a Docker container

## References

<https://docs.microsoft.com/en-us/sql/linux/quickstart-install-connect-docker?view=sql-server-2017>

<https://docs.docker.com/engine/reference/commandline/docker/>

<https://valentina-db.com/dokuwiki/doku.php?id=valentina:products:vstudio:vstudio>

<https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/docker/visual-studio-tools-for-docker?view=aspnetcore-2.1>

<https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/multi-container-microservice-net-applications/multi-container-applications-docker-compose>