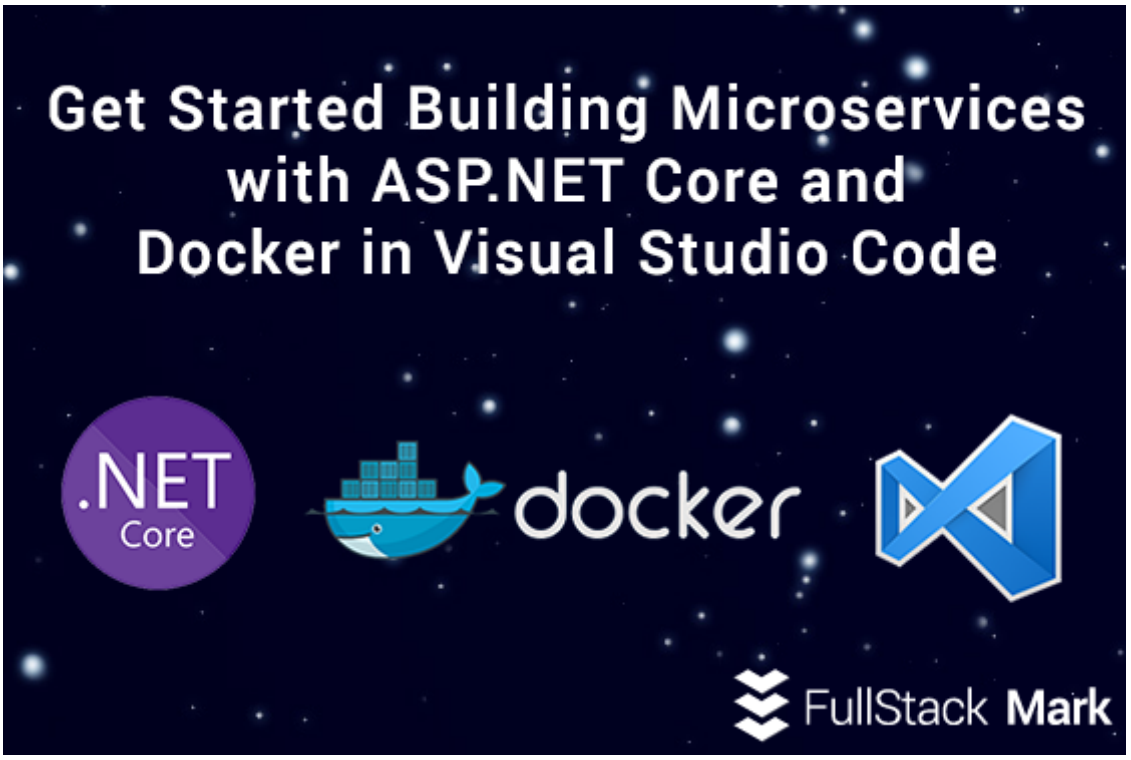
Get Started Building Microservices with ASP.NET Core and Docker in Visual Studio Code

December 19, 2017



Containers and microservices are two *huge*, emerging trends in software development today.

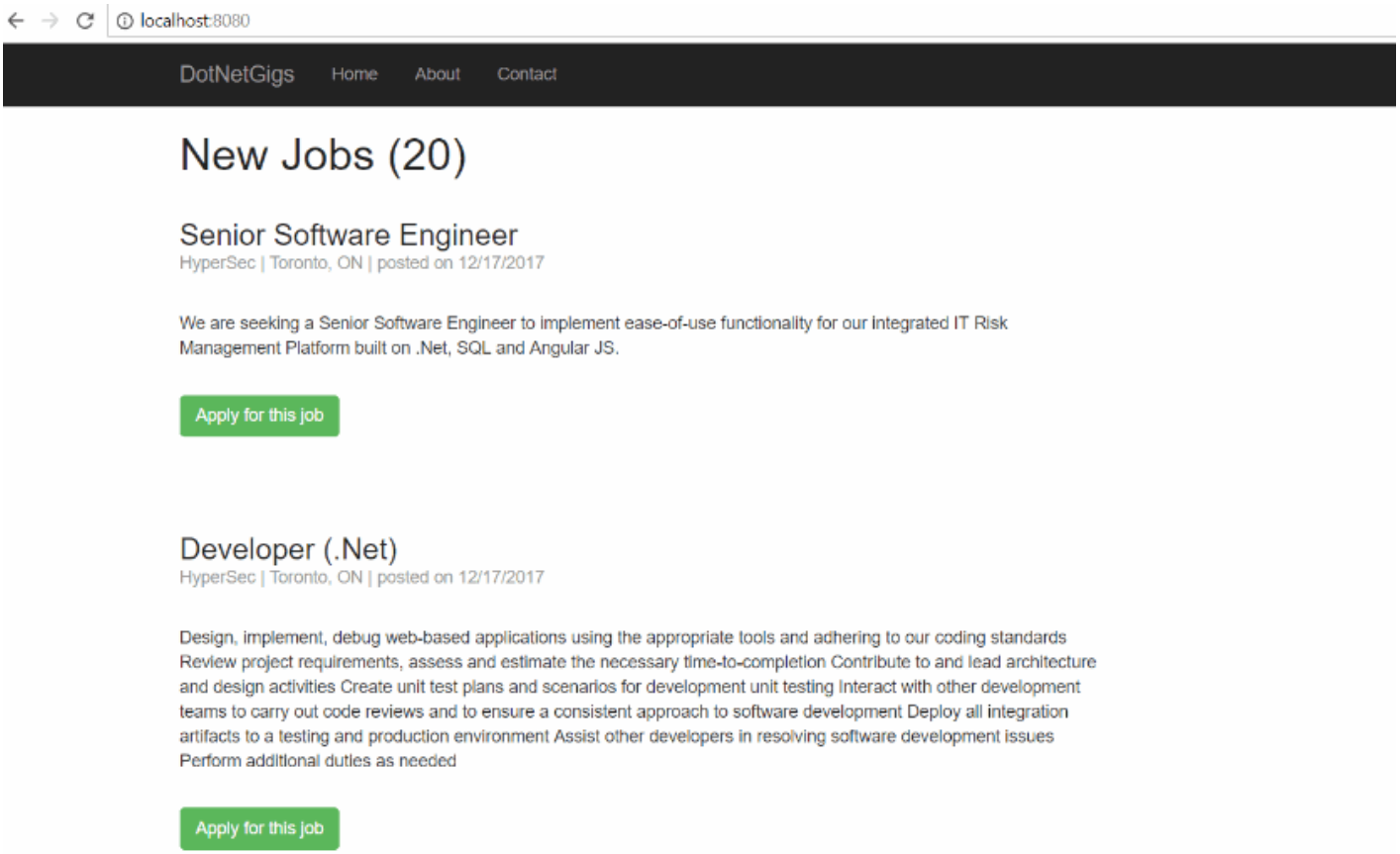
For the uninitiated, containers are a super cool way to package up your application, its dependencies, and configuration in a portable, easily distributable image file. This image can then be downloaded and run in an execution environment called a container on any number of other computers acting as a container host. Microservices represent an architectural style in which the system can be broken up into individual services, each one with a single, narrowly focused capability that is exposed with an API to the rest of the system as well as external consumers like web and mobile apps.

Looking at the characteristics of both concepts, we can start to see why they might work well together to help us develop systems that are easier to deploy, scale, maintain and provide an increased level of stability compared to a traditional monolithic approach.

Two key elements of .NET Core's design are its modularity and lightweight nature. These properties make it ideal for building containerized microservice applications. In this post, we'll see how to combine ASP.NET Core and Docker using a cross-platform approach to build, debug and deploy a microservices-based proof-of-concept using Visual Studio Code, .NET Core CLI and Docker CLI.

Please Note - Both of these topics, particularly microservices - are vast and deep so there are many very important aspects I'll be skimming over or simply not mention here. The goal of this post is to get from zero to off-the-ground with ASP.NET Core-based microservices and Docker. Some of the most critical and challenging exercises in microservice architecture can be properly identifying and defining domain and data models, bounded contexts and their relationships. This post does

not dive deeply into design and modeling theory. Likewise, for containers, there are many other important areas we will not be exploring in this guide like the principles of container design and orchestration.



The demo web app we'll build in this post is powered by 3 ASP.NET Core microservices, RabbitMQ, Redis and Sql Server on Linux all running in docker containers

---

Dev Environment

- Windows 10 and PowerShell
- Visual Studio Code - v1.19.0
  - C# for Visual Studio Code extension
  - Docker extension
- SQL Server Management Studio 17.4
- .NET Core SDK v2.0.0
- Docker Community Edition 17.09.1-ce-win42 using Linux containers

Solution setup

Starting with an empty directory, you can create a new solution using the .NET Core CLI.

```
> dotnet new sln --name dotnetgigs
```

In the same directory, I created a new directory called **services** to house the microservices we'll be using: Applicants.Api, Identity.Api and Jobs.Api.

Within each microservice directory, I created a new Web API project. Note that you can omit the name parameter and the new project will inherit the name of the parent directory.

```
> dotnet new webapi
```

Next, I added each project to the previously created solution file:

```
> dotnet sln add services/applicants.api/applicants.api.csproj
```

```
> dotnet sln add services/jobs.api/jobs.api.csproj
```

```
> dotnet sln add services/identity.api/identity.api.csproj
```

Welcome Docker

At this point, we'll step away from the code for a bit to introduce Docker into our solution and workflow.

One thing we should understand: since we are using Visual Studio Code and a CLI development approach we need to know many of the steps involved in working with Docker in much greater detail than if we were using Visual Studio. Visual Studio 2017 has excellent support for Docker built-in so it offers much greater productivity and saves you from mucking with dockerfiles and the CLI directly. Visual Studio Code, on the other hand, is not nearly as refined at the moment and requires a much more hands-on approach. For our purposes, there is still a lot of value in the CLI approach we'll be using as it forces a greater understanding of the tooling and process involved in Docker development with .NET Core. These steps are also largely cross-platform as they should work in a mac or linux environment with very little adjustment.

Creating Debuggable Containers

One area in particular where the current developer experience with Docker is a bit lacking in Visual Studio Code compared to Visual Studio is debugging. We want to be able to debug our services while they run in Docker. I wasn't quite sure at first how to go about this but a little googling surfaced this thread (thanks galvesribeiro) and the following Dockerfile:

```
FROM microsoft/aspnetcore:latest
RUN mkdir app

#Install debugger
RUN apt-get update
RUN apt-get install curl -y unzip
RUN curl -sSL https://aka.ms/getvsdbgsh | bash /dev/stdin -v latest -l /vsdbg

EXPOSE 80/tcp

#Keep the debugger container on
ENTRYPOINT ["tail", "-f", "/dev/null"]
```

Here's what's going on:

```
FROM microsoft/aspnetcore:latest
```

This command simply tells Docker to use Microsoft's official aspnetcore runtime image as its base. This means our microservice images are automatically provisioned with the .NET Core runtime and ASP.NET Core libs required to *run* an ASP.NET Core application. If we wanted to *build* our app in the container we would need to base it off the `aspnetcore-build` image as it is equipped with the full .NET Core SDK required to build and publish your application. So, when choosing a base image it's important to be aware that they are optimized for different use cases and as such we should look for one that suits our intended usage to avoid unnecessary bloat in our custom image. More info on the official ASP.NET Core images can be found on Microsoft's ASP.NET Core Docker hub repository page. Note also that we are using linux-based images as per our docker setup.

```
#Install debugger
RUN apt-get update
RUN apt-get install curl -y unzip
RUN curl -sSL https://aka.ms/getvsdbgsh | bash /dev/stdin -v latest -l /vsdbg
```

This section installs the VSCode debugger in the container so we can remotely debug the application running inside the container from Visual Studio Code - more on this shortly.

```
#Keep the debugger container on
ENTRYPOINT ["tail", "-f", "/dev/null"]
```

The `ENTRYPOINT` command gives you a way to identify which executable should be run when a container is started. Normally, if we were simply running an ASP.NET Core application directly it would look something like `ENTRYPOINT ["dotnet", "myaspnetapp.dll"]` using the CLI to launch our app. Because this container is used for debugging we want the ability to start/stop the debugger and our application without having to stop and re-start the entire container each time we launch the debugger. To accomplish this we use `tail -f /dev/null`as the ENTRYPOINT which allows the debugger to start and stop in the background but doesn't stop the container because tail keeps running infinitely in the foreground.

I added the same Dockerfile to the project root of each microservice.

Using Docker-Compose to Organize Multi-Container Solutions

It is relatively easy to work with a single Dockerfile using the CLI commands build and run to create an image and spin up new containers. However, as your solution grows to include multiple containers, working with a collection of dockerfiles in this fashion will become painful and error-prone. To make life easier, we can leverage Docker-Compose to encapsulate these commands along with the configuration data for each container to define a set of related services which can be deployed together as a multi-container Docker application.

With Dockerfiles added to each microservice project I created a new Docker-Compose.yml file in the solution root. We'll flesh it out in a bit but starting out you can see it's just a simple YAML-based file with a section defining each of our application's services and some instructions to tell docker how we'd like it to build and configure our containers.

```
version: '3'

services:

  applicants.api:
    image: applicants.api
    build:
      context: ./services/applicants.api
      dockerfile: Dockerfile.debug
    ports:
    - "8081:80"
    volumes:
      - ./services/applicants.api/bin/pub/:/app
    container_name: applicants.api

  identity.api:
    image: identity.api
    build:
      context: ./services/identity.api
      dockerfile: Dockerfile.debug
    ports:
    - "8084:80"
    volumes:
      - ./services/identity.api/bin/pub/:/app
    container_name: identity.api

  jobs.api:
    image: jobs.api
    build:
      context: ./services/jobs.api
      dockerfile: Dockerfile.debug
    ports:
    - "8083:80"
    volumes:
      - ./services/jobs.api/bin/pub/:/app
    container_name: jobs.api
```

Here's what these options do:

- `image` - The image name to start the container from. When specified with the build directive, Docker-Compose will use this as the name when creating the image.
- `build` - Contains the path to the dockerfile to use as well as the context docker should use to build the container from.
- `ports` - Exposes ports to the host machine and shares them among different services started by the docker-compose. Essentially, it provides the network plumbing so we can talk to services running in containers by mapping to ports on the host.

- `volumes` - Allows us to mount paths on our host inside the container. In our case, this is especially useful to support debugging. You can see we point the published output from our app's bin directory **/services/jobs.api/bin/pub** to a volume labeled **/app** that the container can access. This allows us to rebuild our application and launch the debugger from Visual Studio Code without touching the container - rad!
- `container_name` - Allows us to specify a custom name for the container, rather than a generated default.

Adding a Database

With the core microservices defined for our solution, we can start thinking about data for our application. Microsoft recently launched Sql Server on Linux and associated docker images so this is a great opportunity to try it out. Sql Server on Linux - what a time to be alive. I created a Database folder in the solution root along with a new Dockerfile to pull from Microsoft's official image. You'll also notice extra bits in the Dockerfile to run the *SqlCmdStartup.sh* script. This script provisions the databases required by our microservices. Finally, I extended the *docker-compose.yml* file to include the new service. Notice that I am mapping local port **5433** to Sql Server's TCP port so I can use Sql Server Management Studio on my desktop to talk to the database running inside the container - rad! If you have a local instance of sql server running you'll want to do the same.

```
...
sql.data:
   image: mssql-linux
   build:
     context: ./Database
     dockerfile: Dockerfile
   ports:
      - "5433:1433"
   container_name: mssql-linux
```

For production use, it's typically not advisable to put your database in a container, however, there are exceptions to every rule so if you're thinking about production you'll need to carefully test and evaluate any containerized database.

Adding a Data Access Layer to the Microservice

We talk to the containerized Sql Server from our application the exact same way we would if it were installed normally. However, Docker does provide a method of identifying it as a dependency by adding a `depends_on` key to each service in the *docker-compose.yml* file that uses it. This is a handy way to manage dependencies between services when using compose.

```
...
jobs.api:
    ...
    depends_on:
      - sql.data
```

This tells docker to create and start the `sql.data` container before `jobs.api`.

As this is a very small and simple demo, the data access code is pretty straight forward and makes use of Dapper ORM to interact with the database. Check out the ApplicantRepository.cs or JobRepository.cs classes to see how it is implemented.

Caching with Redis

Caching is an essential part of any distributed system. We'll add a redis instance to our architecture that the Identity.Api microservice can use as a backing store for user/session information. This requires only 2 new lines in *docker-compose.yml* and boom - we have a redis instance. For this service, we don't require any additional dockerfile or configuration.

```
...
user.data:
    image: redis
```

The redis cache is mainly used by IdentityRepository. You can see where the client connection is established and wired up in the container in Startup.cs which in turn is injected into the repository. Finally, in startup you will notice the redis host address is resolved for the connection using the configuration provider in the line:

```
configuration.EndPoints.Add(Configuration["RedisHost"]);
```

What makes this special is that the value is set in the *docker-compose.yml* by adding an `environment` key to the Identity.Api service definition and passed in when the container is run. This is also used for passing in the connection string to services using the database.

```
...
identity.api:
    image: identity.api
    environment:
      - RedisHost=user.data:6379
...
```

Event-Based Communication Between Microservices using RabbitMQ and MassTransit

An important rule for microservices architecture is that each microservice must own its data. In a traditional, monolithic application we often have one centralized database where we can retrieve and modify entities across the whole application often in the same process. In microservices, we don't have this kind of freedom. Microservices are independent and run in their own process. So, if a change to an entity or some other notable event occurs in one microservice and must be communicated to other interested services we can use a message bus to publish and consume messages between microservices. This keeps our microservices completely decoupled from one another and any other external systems they may integrate with.

To add messaging to the solution I first added a RabbitMQ message broker container by extending *docker-compose.yml*:

```
...
rabbitmq:
  image: rabbitmq:3-management
  ports:
    - "15672:15672"
  container_name: rabbitmq
...
```

Next, to publish and consume messages within the microservices I opted to use MassTransit which is a lightweight, message bus framework that works with RabbitMQ and Azure Service Bus. I could've very well used a raw rabbit client but MassTransit provides a nice, friendly abstraction over rabbit. One shortcut I took is not creating a single component to represent the event bus so in each microservice's Startup.cs you can see an instance of the bus is created and registered with the container.

```
builder.Register(c =>
{
   return Bus.Factory.CreateUsingRabbitMq(sbc =>
     {
       sbc.Host("rabbitmq", "/", h =>
       {
          h.Username("guest");
          h.Password("guest");
       });

       sbc.ExchangeType = ExchangeType.Fanout;
     });
})
.As<IBusControl>()
.As<IBus>()
.As<IPublishEndpoint>()
.SingleInstance();
```

After that, publishing messages to Rabbit is a breeze. Simply inject the instance of the bus, in our case into a controller and you're ready to publish events. An example of this is in the JobsController in Jobs.Api.

```
[HttpPost("/api/jobs/applicants")]
public async Task<IActionResult> Post([FromBody]JobApplicant model)
{
  // fetch the job data
  var job = await _jobRepository.Get(model.JobId);
  var id = await _jobRepository.AddApplicant(model);
  // dispatch 'ApplicantApplied' event message
  await _bus.Publish<ApplicantAppliedEvent>(new { model.JobId, model.ApplicantId,
job.Title });
  return Ok(id);
}
```

Consuming events is fairly straightforward too. MassTransit provides a nice mechanism for defining message consumers via its `IConsumer` interface which is where our message handling code goes.

An example of this can be seen in the Identity.Api where a message consumer for the `ApplicantApplied` event is defined in ApplicantAppliedEventConsumer. These consumer classes can then be registered with the Autofac container and invoked automatically by MassTransit with just a little extra configuration on the bus instance we register in the container.

```
// register a specific consumer
builder.RegisterType<ApplicantAppliedEventConsumer>();

builder.Register(context =>
{
  var busControl = Bus.Factory.CreateUsingRabbitMq(cfg =>
  {
     var host = cfg.Host(new Uri("rabbitmq://rabbitmq/"), h =>
     {
       h.Username("guest");
       h.Password("guest");
     });

   // https://stackoverflow.com/questions/39573721/disable-round-robin-pattern-and-use-
fanout-on-masstransit
   cfg.ReceiveEndpoint(host, "dotnetgigs" + Guid.NewGuid().ToString(), e =>
   {
      e.LoadFrom(context);
   });
});

return busControl;

})
.SingleInstance()
.As<IBusControl>()
.As<IBus>();
```

If you look in the *ApplicantAppliedEventConsumer* class you'll see it's not doing much. It just increments a value in the redis cache but it clearly illustrates how asynchronous, event-driven communication between microservices can work.

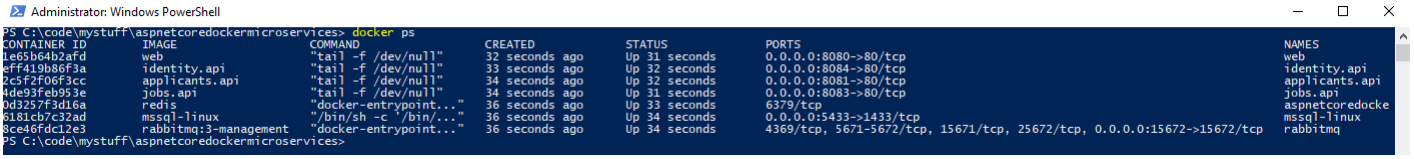
```
public async Task Consume(ConsumeContext<ApplicantAppliedEvent> context)
{
   // increment the user's application count in the cache
   await
_identityRepository.UpdateUserApplicationCountAsync(context.Message.ApplicantId.ToString()

}
```

Consuming Microservices with an ASP.Net Core MVC Web App

To consume our microservices and complete the DotNetGigs demo app I added a new ASP.NET Core MVC application to the solution. The most common method for client web and mobile applications to talk to microservices is over http - commonly via an API gateway. We're not using a gateway in this demo but I did create a simple http client so the mvc app can talk directly to the different microservices. The extent of the app's functionality is captured in the animated gif above - it can retrieve a list of jobs from Jobs.Api, the user can apply, and messages are dispatched and handled by the other microservices - that's it!

Building and Debugging the Solution

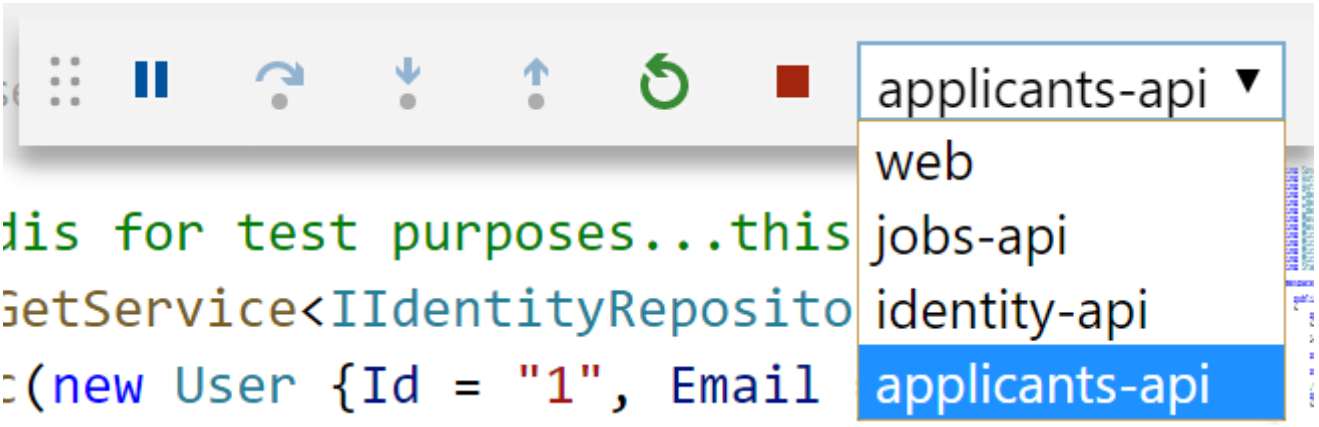
From the project's root folder (where *docker-compose.yml* resides) use the Docker CLI to build and start the containers for the solution: `PS> docker-compose up -d`. This step will take a few minutes or more as all the base images must be downloaded. When it completes you can check that all **7** containers for the solution have been built and started successfully by running `PS> docker ps`.



Additionally, you can connect to the Sql Server on Linux instance in the container using SQL Server Management Studio to ensure the databases **dotnetgigs.applicants** and **dotnetgigs.jobs** were created. The server name is: **localhost,5433** with username **sa** and password **Pass@word**.

At this point, you can run and debug the solution from Visual Studio Code. Simply open the root folder in VSCode and start up each of the projects in the debugger. Unfortunately, they need to be started individually (if you know a way around this please let me know :) The order they're started does not matter.

Update: Thanks to Burhan below for pointing out that it is very easy to launch all the projects simultaneously using [compound launch configurations](). The sample code has been updated with his suggestion so you can launch all the projects in the solution in one shot by selecting the *All Projects* config in the VSCode debugger. Thanks Burhan!



With all services running in the debugger you can hit the web app in your browser at **http://localhost:8080** and set breakpoints in any of the projects to debug directly.

Wrapping Up

If you're still here, you rock! I hope this guide is true to its title and can help you get off the ground and running with ASP.NET Core microservices and Docker. I'd like to finish by recapping a few key benefits these technologies and architectural style can provide developers and organizations.

- Docker containers ensure consistency across multiple development and release cycles. This helps teams realize cost and time savings by preventing annoying deployment issues, improving DevOps and production stability.
- .NET Core's modularity and lightweight nature make it ideal for microservice development.
- Teams can move faster with microservices because they can be developed, deployed and tested independently of each other, unlike traditional monolithic applications.
- Docker provides a better return on investment than traditional deployment models by dramatically reducing infrastructure resources. By nature, Docker can run the same application with fewer resources because of the reduced infrastructure requirements.

- Microservices can produce more scalable and resilient systems. Because they typically only have a single, independent responsibility they are easier to scale and replicate than monolithic services.

Thanks for reading and if you have any questions or feedback I'd love to hear it in the comments below!

source code here