

MicroServices tutorial for C#

Dr. Balázs Simon (sbalazs@iit.bme.hu), BME IIT, 2023

1 Introduction

This document describes how to create ASP.NET core microservices communicating through gRPC and how to host them in Docker. To be able to follow this tutorial need to install Docker Desktop:

<https://www.docker.com/products/docker-desktop/>

2 Back-end service

Open a command prompt from your project directory and issue the following commands to create a new web application (the `Ý` symbol means that the command is continued in the next line, and the whole command should be typed as a single line):

```
c:\Users\[user]\source\projects>mkdir HelloWorldBackend
```

```
c:\Users\[user]\source\projects>dotnet new web -n HelloWorldBackend&#221;  
-o HelloWorldBackend
```

(Make sure to use the simple command prompt and not PowerShell, since PowerShell handles the dash switches differently.)

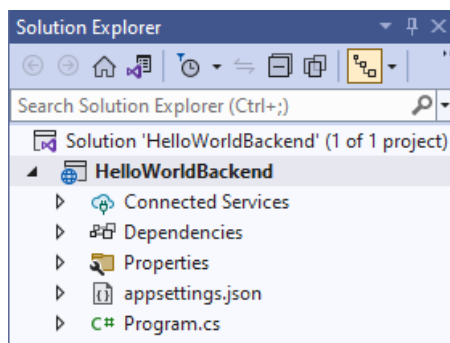
Enter the project folder:

```
c:\Users\[user]\source\projects>cd HelloWorldBackend
```

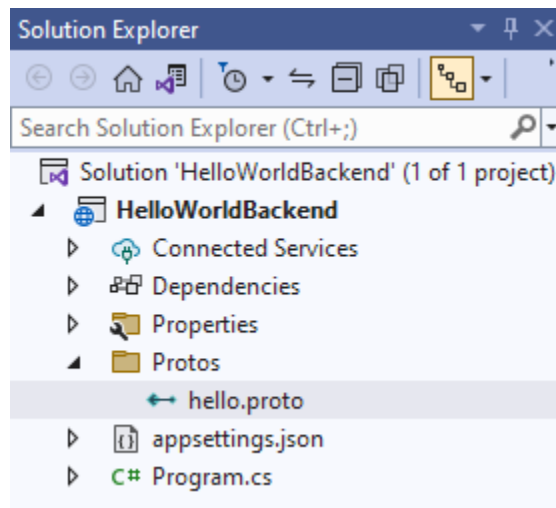
Inside the project folder add the following NuGet packages with the following commands:

```
dotnet add package Grpc.AspNetCore
```

Open the project in Visual Studio:



Right click on the project and select **Add > New Folder** to add a new folder called **Protos**. Right click on the **Protos** folder and select **Add > New Item** to add a new file called **hello.proto**. The project should look like this:



Type the following code into **hello.proto**:

```
syntax = "proto3";

option csharp_namespace = "HelloWorldMicroService";
option java_package = "hellomicro";

package HelloWorldMicroService;

service Hello {
  rpc SayHello (SayHelloRequest) returns (SayHelloReply);
}

message SayHelloRequest {
  string name = 1;
}

message SayHelloReply {
  string message = 1;
}
```

Click on the **HelloWorldBackend** project to see the **HelloWorldBackend.csproj** file and add the following lines displayed with a yellow background:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Grpc.AspNetCore" Version="2.52.0" />
  </ItemGroup>
  <ItemGroup>
```

```

    <Protobuf Include="Protos\hello.proto" GrpcServices="Server" />
  </ItemGroup>
</Project>

```

After saving the files, the Protobuf compiler will generate the necessary files for the server side to provide the base of the implementation of the service.

Add a file called **HelloBackendService.cs** to the project with the following content:

```

using Grpc.Core;
using HelloWorldMicroService;

namespace HelloWorldBackend
{
    public class HelloBackendService : Hello.HelloBase
    {
        public override async Task<SayHelloReply> SayHello(SayHelloRequest request,
                                                            ServerCallContext context)
        {
            return
                new SayHelloReply()
                {
                    Message = $"Hello from backend: {request.Name}"
                };
        }
    }
}

```

This class implements the **Hello** interface defined in the proto file.

Modify the **Program.cs** file to have the following content:

```

using HelloWorldBackend;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddGrpc();

var app = builder.Build();
app.MapGet("/", () => "Hello World Back-End!");
app.MapGrpcService<HelloBackendService>();

app.Run();

```

The **AddGrpc** method adds gRPC support to ASP.NET core. The **MapGrpcService** method publishes our back-end service through gRPC.

Modify the **Properties\launchSettings.json** file, and update the HTTP ports to 5000, the HTTPS ports to 5001:

```

{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:7085",
      "sslPort": 44393
    }
  }
}

```

```

    }
  },
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}

```

Now you can run the application. However, we need a client to call the service. That client will be the front-end of our application described in the next chapter.

3 Front-end service

Open a command prompt from your project directory and issue the following commands to create a new web application:

```
c:\Users\[user]\source\projects>mkdir HelloWorldFrontend
```

```
c:\Users\[user]\source\projects>dotnet new web -n HelloWorldFrontend
-o HelloWorldFrontend
```

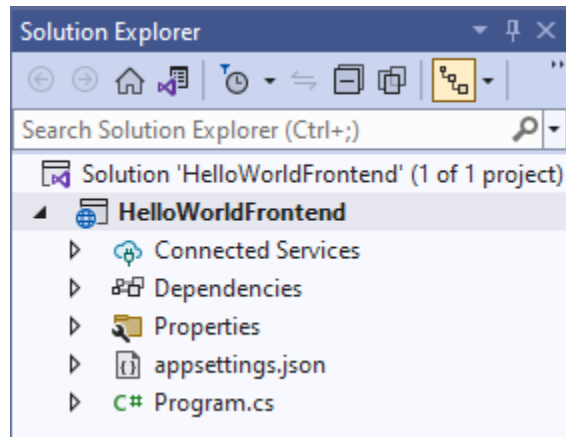
Enter the project folder:

```
c:\Users\[user]\source\projects>cd HelloWorldFrontend
```

Inside the project folder add the following NuGet packages with the following commands:

```
dotnet add package Grpc.AspNetCore
```

Open the project in Visual Studio:



Add a new folder to the project called **Protos**. Add a new file called **hello.proto** under the **Protos** folder with the same content as in the back-end project.

Click on the **HelloWorldFrontend** project to see the **HelloWorldFrontend.csproj** file and add the following lines displayed with a yellow background:

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

  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Grpc.AspNetCore" Version="2.52.0" />
  </ItemGroup>

  <ItemGroup>
    <Protobuf Include="Protos\hello.proto" GrpcServices="Both" />
  </ItemGroup>

</Project>
```

After saving the files, the Protobuf compiler will generate the necessary files for both the server side and the client side. We need the server side to publish our front-end service and we need the client side to call the back-end service.

Add a file called **HelloFrontendService.cs** to the project with the following content:

```
using Grpc.Core;
using HelloWorldMicroService;

namespace HelloWorldFrontend
{
    public class HelloFrontendService : Hello.HelloBase
    {
    }
```

```

private readonly Hello.HelloClient _backend;

public HelloFrontendService(Hello.HelloClient client)
{
    _backend = client;
}

public override async Task<SayHelloReply> SayHello(SayHelloRequest request,
                                                    ServerCallContext context)
{
    var replyFromBackend = await _backend.SayHelloAsync(request);
    return
        new SayHelloReply()
        {
            Message = $"Hello from frontend: {replyFromBackend.Message}"
        };
}
}
}

```

This class implements the **Hello** interface defined in the proto file and forwards the call to the back-end **Hello** service. The client for the back-end is provided by dependency injection through the constructor.

Modify the **appsettings.json** file to have the following content:

```

{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "HelloBackend": {
    "Url": "https://localhost:5001"
  }
}

```

This configures the location of the back-end service.

Modify the **Program.cs** file to have the following content:

```

using HelloWorldFrontend;
using HelloWorldMicroService;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddGrpc().AddJsonTranscoding();

var backendUrl = builder.Configuration.GetSection("HelloBackend")
    .GetValue<string>("Url");
builder.Services.AddGrpcClient<Hello.HelloClient>(o =>
{
    o.Address = new Uri(backendUrl);
});

var app = builder.Build();

```

```
app.UseStaticFiles();
app.MapGet("/", () => "Hello World Front-End!");
app.MapGrpcService<HelloFrontendService>();

app.Run();
```

The **AddGrpc** method adds gRPC support to ASP.NET core. The **AddGrpcClient** method adds a client proxy that can call the back-end. The **MapGrpcService** method publishes our front-end service through gRPC.

Modify the **Properties\launchSettings.json** file, and update the HTTP ports to 5100, the HTTPS ports to 5101:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:7085",
      "sslPort": 44393
    }
  },
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5100",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5101;http://localhost:5100",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Now we could run the application. However, we have the same problem as in the back-end: we need a client to call the front-end service.

4 C# client

Open a command prompt from your project directory and issue the following commands to create a new web application:

```
c:\Users\[user]\source\projects>mkdir HelloWorldClient
```

```
c:\Users\[user]\source\projects>dotnet new console -n HelloWorldClient  
-o HelloWorldClient
```

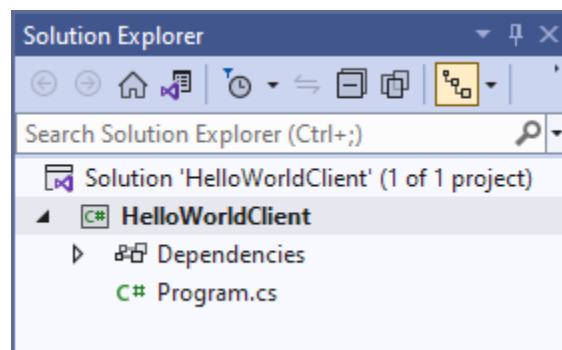
Enter the project folder:

```
c:\Users\[user]\source\projects>cd HelloWorldClient
```

Inside the project folder add the following NuGet packages with the following commands:

```
dotnet add package Google.Protobuf  
dotnet add package Grpc.Net.Client  
dotnet add package Grpc.Tools
```

Open the project in Visual Studio:



Add a new folder to the project called **Protos**. Add a new file called **hello.proto** under the **Protos** folder with the same content as in the back-end project.

Click on the **HelloWorldClient** project to see the **HelloWorldClient.csproj** file and add the following lines displayed with a yellow background:

```
<Project Sdk="Microsoft.NET.Sdk">  
  <PropertyGroup>  
    <OutputType>Exe</OutputType>  
    <TargetFramework>net7.0</TargetFramework>  
    <ImplicitUsings>enable</ImplicitUsings>  
    <Nullable>enable</Nullable>  
  </PropertyGroup>  
  
  <ItemGroup>  
    <PackageReference Include="Google.Protobuf" Version="3.22.3" />  
    <PackageReference Include="Grpc.Net.Client" Version="2.52.0" />  
    <PackageReference Include="Grpc.Tools" Version="2.54.0">
```



```

        <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
        <PrivateAssets>all</PrivateAssets>
    </PackageReference>
</ItemGroup>
<ItemGroup>
    <Protobuf Include="Protos\hello.proto" GrpcServices="Client" />
</ItemGroup>
</Project>

```

After saving the files, the Protobuf compiler will generate the necessary files for both the the client side.

Modify the **Program.cs** to include the following code:

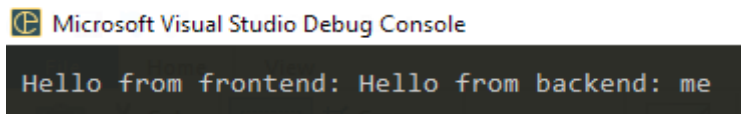
```

using Grpc.Net.Client;
using HelloWorldMicroService;

var channel = GrpcChannel.ForAddress("https://localhost:5101");
var client = new Hello.HelloClient(channel);
var reply = await client.SayHelloAsync(new SayHelloRequest() { Name = "me " });
Console.WriteLine(reply.Message);

```

Make sure that both the back-end and front-end application is running. If you run the client application, the result is the following:



5 JavaScript client

Another way of calling the front-end service is to add a JavaScript client. One limitation with gRPC is that not every platform can use it. Browsers don't fully support HTTP/2. Instead, we need to provide an additional REST API interface with JSON messages for our front-end service so that it can also be called from JavaScript. Fortunately, ASP.NET core has a built-in solution for this through JSON transcoding.

Go back to the **HelloWorldFrontend** application. In the command line add the following NuGet packages:

```

c:\Users\[user]\source\projects\HelloWorldFrontend>dotnet add package Microsoft.AspNetCore.Grpc.JsonTranscoding

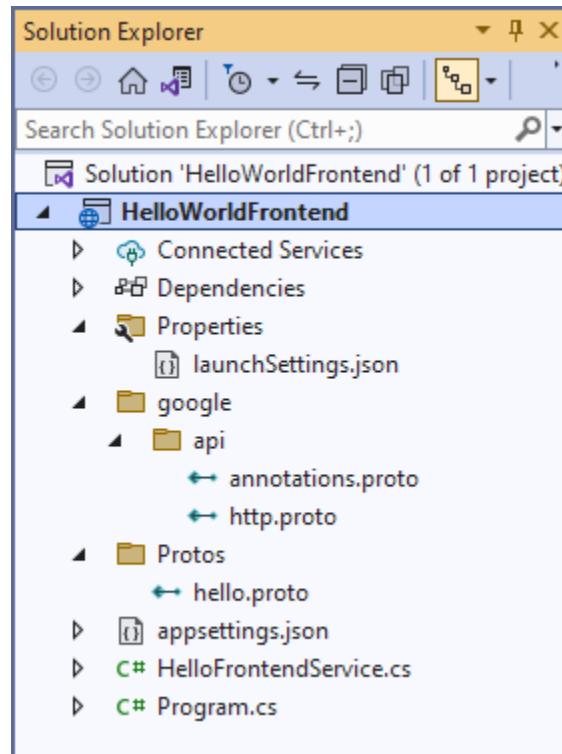
```

Create the directory structure **/google/api** in the project and add the **http.proto** and **annotations.proto** files attached to this tutorial into the **/google/api** folder. The original locations of these files are:

<https://github.com/dotnet/aspnetcore/blob/main/src/Grpc/JsonTranscoding/test/testassets/Sandbox/google/api/http.proto>

<https://github.com/dotnet/aspnetcore/blob/main/src/Grpc/JsonTranscoding/test/testassets/Sandbox/google/api/annotations.proto>

The project should look like this:



Modify the **Protos\hello.proto** file to have the following content:

```
syntax = "proto3";

option csharp_namespace = "HelloWorldMicroService";

import "google/api/annotations.proto";

package HelloWorldMicroService;

service Hello {
  rpc SayHello (SayHelloRequest) returns (SayHelloReply) {
    option (google.api.http) = {
      post: "/api/hello",
      body: "*"
    };
  }
}

message SayHelloRequest {
  string name = 1;
}

message SayHelloReply {
  string message = 1;
}
```

This extra annotation will map the **SayHello** method to a **POST** request, where the body of the request is the **SayHelloRequest** message in JSON format and the HTTP response body contains the **SayHelloReply** message in JSON format.

We also have to modify the **Program.cs** file:

```
using HelloWorldFrontend;
using HelloWorldMicroService;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddGrpc().AddJsonTranscoding();
builder.Services.AddGrpcClient<Hello.HelloClient>(o =>
{
    o.Address = new Uri("https://localhost:5001");
});

var app = builder.Build();
app.UseStaticFiles();
app.MapGrpcService<HelloFrontendService>();

app.Run();
```

The **AddJsonTranscoding** method activates the JSON transcoding for gRPC. The **UseStaticFiles** method is needed so that we can publish our HTML page with the JavaScript code.

Create a new folder called **wwwroot** and add a new HTML file called **hello.html** under this folder with the following content. This is our client:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Hello World MicroService client</title>
    <script>
        function init() {
            output = document.getElementById("output");
        }

        function send_message() {
            let text = textID.value;
            let data = { name: text }
            fetch('/api/hello', { method: "POST", body: JSON.stringify(data),
headers: { "Content-type": "application/json; charset=UTF-8" } })
                .then((response) => response.json())
                .then((result) => {
                    writeToScreen(result.message);
                })
                .catch(err => console.log(err));
        }

        function writeToScreen(message) {
            var pre = document.createElement("p");
            pre.style.wordWrap = "break-word";
            pre.innerHTML = message;
```

```

        output.appendChild(pre);
    }

    window.addEventListener("load", init, false);

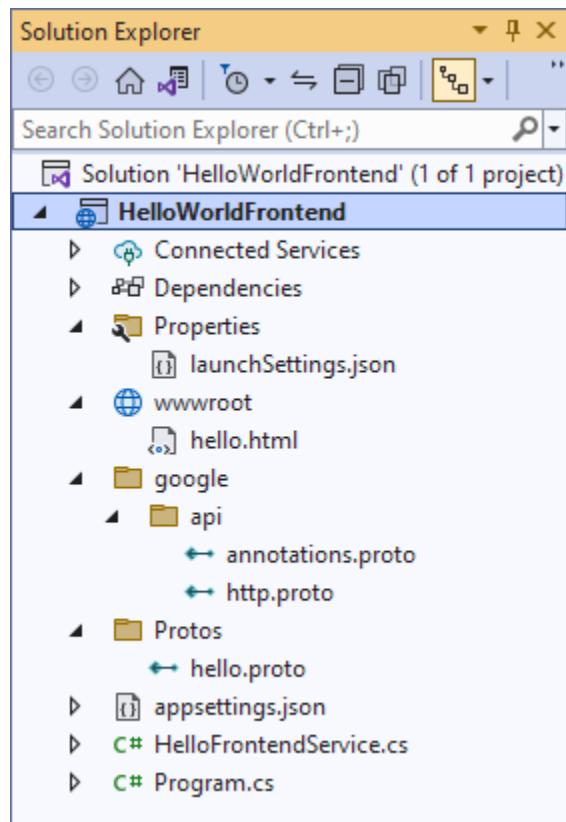
</script>
</head>
<body>
    <h1 style="text-align: center;">Hello World MicroService client</h1>

    <br />

    <div style="text-align: center;">
        <form action="">
            <input onclick="send_message()" value="Send" type="button" />
            <input id="textID" name="message" value="me" type="text" /><br />
        </form>
    </div>
    <div id="output"></div>
</body>
</html>

```

The project should look like this:

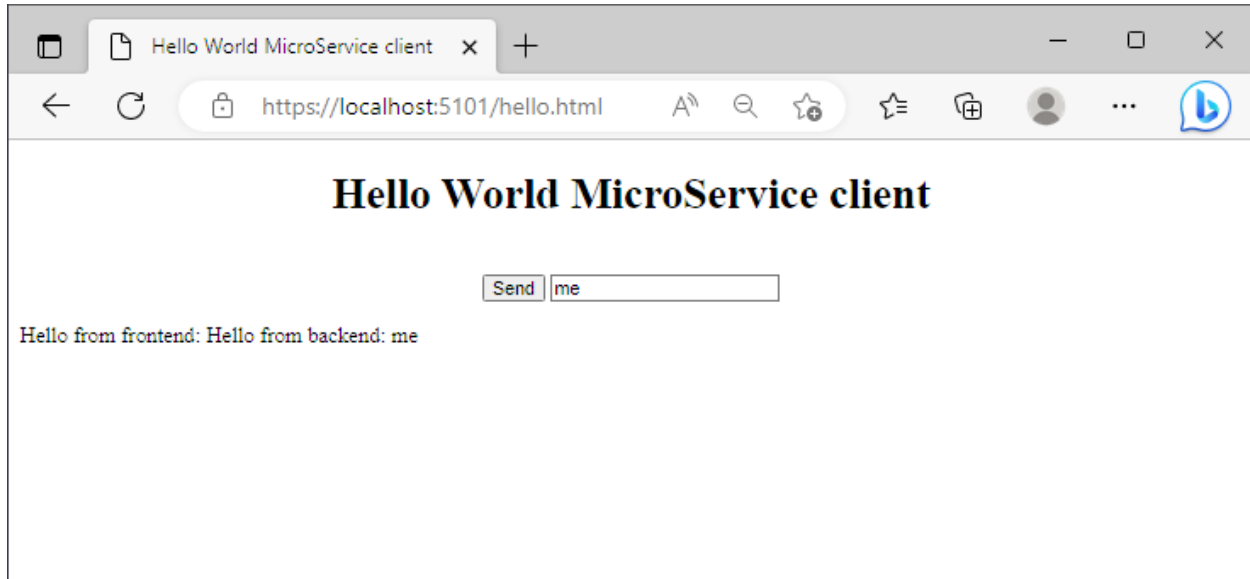


Make sure that the back-end application is running. Run the front-end application with **Ctrl+F5**.

To test the client, type the following URL into a browser (FireFox, Chrome, IE, etc.):

https://localhost:5101/hello.html

Type something in the input field and click on the **Send** button:



6 Hosting in Docker

To host our applications in Docker we have to dockerize them. This involves creating a Dockerfile and compiling the Docker image. The Dockerfile describes how the application is installed into the base image and how it is run inside the container.

Create a file called **Dockerfile** inside the **HelloWorldBackend** project with the following content:

```
FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build
WORKDIR /src
COPY ./HelloWorldBackend.csproj .
RUN dotnet restore HelloWorldBackend.csproj
COPY . .
RUN dotnet build HelloWorldBackend.csproj -c Release -o /app/build

FROM build AS publish
RUN dotnet publish HelloWorldBackend.csproj -c Release -o /app/publish

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

This means that we are using the ASP.NET Core 7.0 runtime as the base image which will expose its HTTP port 80 and HTTPS port 443. We are using ASP.NET Core 7.0 SDK as a build image. Into this build image we copy our back-end application, build it and publish it. Then we copy the published application to the final image which is based on the ASP.NET Core 7.0 runtime base image and when the container starts we start the application with the **dotnet** command.

Create a file called **Dockerfile** inside the **HelloWorldFrontend** project with the following content:

```
FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build
WORKDIR /src
COPY ./HelloWorldFrontend.csproj .
RUN dotnet restore HelloWorldFrontend.csproj
COPY . .
RUN dotnet build HelloWorldFrontend.csproj -c Release -o /app/build

FROM build AS publish
RUN dotnet publish HelloWorldFrontend.csproj -c Release -o /app/publish

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

This is the same thing as the previous Dockerfile but for the front-end application.

Open a command line window, go into the back-end project's folder and issue the following commands to compile the back-end application, the corresponding Docker image and run a container based on this image:

```
c:\Users\[user]\source\projects\HelloWorldBackend>docker build
--tag=hellomicro.backend:latest .
```

```
c:\Users\[user]\source\projects\HelloWorldBackend>docker run --rm
-p5000:80 hellomicro.backend:latest
```

The last command starts the back-end container and exposes its internal port **80** as an external port **5000**. Because of the **--rm** flag, the container will be automatically deleted when it is stopped. The compiled image will not be deleted and a new container instance can be started from it any time.

Open another command line window, do the same for the front-end:

```
c:\Users\[user]\source\projects\HelloWorldFrontend>docker build
--tag=hellomicro.frontend:latest .
```

```
c:\Users\[user]\source\projects\HelloWorldFrontend>docker run --rm
-p5100:80 hellomicro.frontend:latest
```

The last command starts the back-end container and exposes its internal port **80** as an external port **5100**.

You can now try to call the back-end or the front-end with the C# client we have created in the previous chapter, however, they will not work. One problem is that HTTPS is not started inside the container. The other problem is that even if it were started, it would require a server certificate. During development we were using an automatically generated certificate which is not available in the published version of the application. So we have to acquire a certificate somehow. One option is to get a certificate signed by a certificate authority. This is recommended in a real production environment, however, this usually costs some money. Another option is to generate our own self-signed certificates. These certificates are not trusted automatically like the ones signed by an official certificate authority, but we can make our operating system to trust them, and they are free. This second option is not recommended for real production environments, but for this tutorial it is sufficient.

First, we have to stop the current containers. Open a third command line and issue the following command:

docker container list

You should see the two running containers:

```
λ docker container list
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS                               NAMES
724ae3d1e45f   hellomicro.frontend:latest          "dotnet HelloWorldFr..." 16 minutes ago Up 16 minutes 0.0.0.0:5101->5101/tcp             inspiring_banzai
98e4909904fc   hellomicro.backend:latest           "dotnet HelloWorldBa..." 19 minutes ago Up 19 minutes 0.0.0.0:5001->5001/tcp             boring_jemison
```

You can stop these containers by referring to their names or container ids. For example, we can stop the front-end based on its name with the following command (the name of your own container is probably something else):

docker stop inspiring_banzai

We can stop the back-end based on its identifier, and is it enough to specify only a part of the identifier (the identifier of your own container is probably something else):

docker stop 98e

Now, let's generate our self-signed certificate. Open a command prompt from the back-end application's folder and issue the following command to generate the certificate:

```
c:\Users\[user]\source\projects\HelloWorldBackend>dotnet dev-certs https -ep certs\hm-backend.pfx -p mypassword
```

Do the same thing in the front-end project:

```
c:\Users\[user]\source\projects\HelloWorldFrontend>dotnet dev-certs https -ep certs\hm-frontend.pfx -p mypassword
```

We also have to trust these certificates. The front-end must trust the back-ends certificate and the client must trust the front-end's. Configuring this is a bit complex, so we will use a workaround in this tutorial. However, in a real production environment, trusting the certificates must be configured properly.

Modify the front-end's **Program.cs** as follows:

```
using Google.Api;
using HelloWorldFrontend;
using HelloWorldMicroService;

var httpHandler = new HttpClientHandler();
httpHandler.ServerCertificateCustomValidationCallback =
    HttpClientHandler.DangerousAcceptAnyServerCertificateValidator;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddGrpc().AddJsonTranscoding();

var backendUrl =
builder.Configuration.GetSection("HelloBackend").GetValue<string>("Url");
builder.Services.AddGrpcClient<Hello.HelloClient>(o =>
{
    o.Address = new Uri(backendUrl);
}).ConfigurePrimaryHttpMessageHandler(() =>
{
    var handler = new HttpClientHandler();
    handler.ServerCertificateCustomValidationCallback =
        HttpClientHandler.DangerousAcceptAnyServerCertificateValidator;
    return handler;
});

var app = builder.Build();
app.UseStaticFiles();
app.MapGet("/", () => "Hello World Front-End!");
app.MapGrpcService<HelloFrontendService>();

app.Run();
```

This option allows the front-end to trust any certificate. Of course, this is a workaround and it is not recommended in a production environment.

Now rebuild the Docker image of the front-end:

```
c:\Users\[user]\source\projects\HelloWorldFrontend>docker build
--tag=hellomicro.frontend:latest .
```

And we can run it like this:

```
c:\Users\[user]\source\projects\HelloWorldBackend>docker run --rm
-p5100:80 -p5101:443
-e ASPNETCORE_URLS="https://+;http://+"
-e ASPNETCORE_HTTPS_PORT=5101
-e ASPNETCORE_Kestrel__Certificates__Default__Password="mypassword"
-e ASPNETCORE_Kestrel__Certificates__Default__Path=/https/hm-frontend.pfx
```



```
-v %cd%\certs:/https/
hellomicro.frontend:latest
```

We can run the back-end container with the following command:

```
c:\Users\[user]\source\projects\HelloWorldBackend>docker run --rm
-p5000:80 -p5001:443
-e ASPNETCORE_URLS="https://+;http://+"
-e ASPNETCORE_HTTPS_PORT=5001
-e ASPNETCORE_Kestrel__Certificates__Default__Password="mypassword"
-e ASPNETCORE_Kestrel__Certificates__Default__Path=/https/hm-backend.pfx
-v %cd%\certs:/https/
hellomicro.backend:latest
```

We also have to modify the clients's **Program.cs** as follows to trust any certificate:

```
using Grpc.Net.Client;
using HelloWorldMicroService;

var httpHandler = new HttpClientHandler();
httpHandler.ServerCertificateCustomValidationCallback =
    HttpClientHandler.DangerousAcceptAnyServerCertificateValidator;

var channel = GrpcChannel.ForAddress("https://localhost:5101",
    new GrpcChannelOptions { HttpHandler = httpHandler });
var client = new Hello.HelloClient(channel);
var reply = await client.SayHelloAsync(new SayHelloRequest() { Name = "me " });
Console.WriteLine(reply.Message);
```

You can now try to call the back-end or the front-end with the C# client we have created in the previous chapter, however, only the back-end will work. The reason for this is that the two containers cannot communicate with each other. To make this work, we need to define a virtual network between them. But now, it will be easier to do it with Docker Compose, which can also start both containers at the same time and we don't even have to type these long commands by hand.

Stop the currently running back-end and front-end containers.

For the networking to work, we have to change the URL of the back-end in the **appsettings.json** inside the **HelloWorldFrontend** project:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "HelloBackend": {
    "Url": "https://hm-backend"
  }
}
```

Inside Docker, the name of the back-end container will be **hm-backend**, so it will be accessible through this domain name. The HTTPS port will be the default **443** port, so we don't have to specify it.

Now go to the main folder that contains both the front-end and back-end projects (e.g. **c:\Users\[user]\source\projects**), and create a file called **docker-compose.yml** with the following content:

```
version: '3'
services:
  hm-backend:
    container_name: hm-backend
    build:
      context: HelloWorldBackend
    image: hellomicro.backend:latest
    ports:
      - 5000:80
      - 5001:443
    environment:
      - ASPNETCORE_URLS=https://+;http://+
      - ASPNETCORE_HTTPS_PORT=5001
      - ASPNETCORE_Kestrel__Certificates__Default__Password=mypassword
      - ASPNETCORE_Kestrel__Certificates__Default__Path=/https/hm-backend.pfx
    networks:
      - hello-micro-network
    volumes:
      - ./HelloWorldBackend/certs:/https/
  hm-frontend:
    container_name: hm-frontend
    build:
      context: HelloWorldFrontend
    image: hellomicro.frontend:latest
    ports:
      - 5100:80
      - 5101:443
    environment:
      - ASPNETCORE_URLS=https://+;http://+
      - ASPNETCORE_HTTPS_PORT=5101
      - ASPNETCORE_Kestrel__Certificates__Default__Password=mypassword
      - ASPNETCORE_Kestrel__Certificates__Default__Path=/https/hm-frontend.pfx
    networks:
      - hello-micro-network
    volumes:
      - ./HelloWorldFrontend/certs:/https/:ro
networks:
  hello-micro-network:
    driver: bridge
```

The elements have the following meaning:

- **version:** Specifies the format version of the compose file.
- **services:** Each object in this key defines a service, i.e., a container.
 - **build:** If given, docker-compose is able to build an image from a Dockerfile
 - **context:** If given, it specifies the build-directory, where the Dockerfile is looked-up.
 - **image:** The name of the image built.

- **ports:** Specifies which ports are exposed from the container to the host machine.
- **environments:** Specifies the environment variables inside the container.
- **networks:** This is the identifier of the named networks to use. A given name-value must be listed in the networks section.
- **volumes:** Specifies the files or directories mounted from the host into the container.
- **networks:** In this section, we're specifying the networks available to our services. In this example, we let docker-compose create a named network of type 'bridge' for us. This will allow communication between the two containers.

Open a command prompt from the directory of the **docker-compose.yml** file, and issue the following command to check the file for syntax errors:

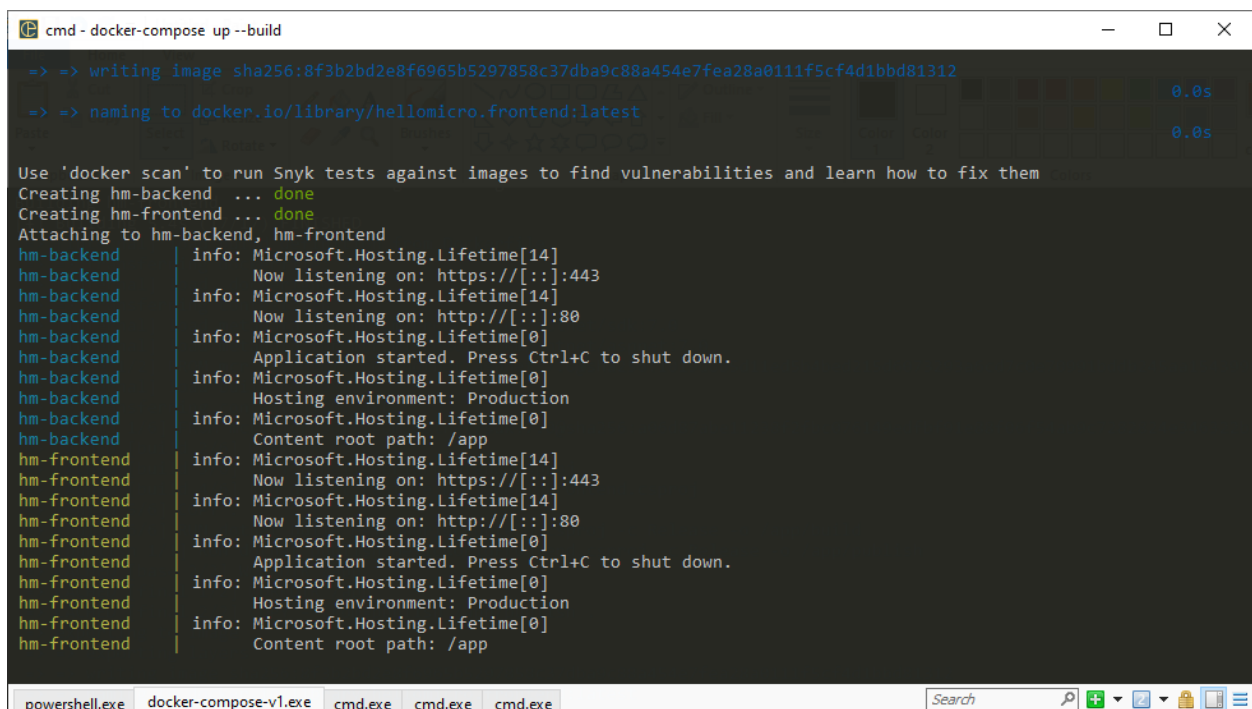
```
c:\Users\[user]\source\projects>docker-compose config
```

If there are no errors you can build the images, create the defined containers, and start them in one command:

```
c:\Users\[user]\source\projects>docker-compose up --build
```

(Once the images are built, you don't need the --build flag any more.)

The two containers should start in parallel and their logs should be merged:



```
cmd - docker-compose up --build

=> => writing image sha256:8f3b2bd2e8f6965b5297858c37dba9c88a454e7fea28a0111f5cf4d1bbd81312
=> => naming to docker.io/library/hellomicro.frontend:latest

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
Creating hm-backend ... done
Creating hm-frontend ... done
Attaching to hm-backend, hm-frontend
hm-backend | info: Microsoft.Hosting.Lifetime[14]
hm-backend | Now listening on: https://[::]:443
hm-backend | info: Microsoft.Hosting.Lifetime[14]
hm-backend | Now listening on: http://[::]:80
hm-backend | info: Microsoft.Hosting.Lifetime[0]
hm-backend | Application started. Press Ctrl+C to shut down.
hm-backend | info: Microsoft.Hosting.Lifetime[0]
hm-backend | Hosting environment: Production
hm-backend | info: Microsoft.Hosting.Lifetime[0]
hm-backend | Content root path: /app
hm-frontend | info: Microsoft.Hosting.Lifetime[14]
hm-frontend | Now listening on: https://[::]:443
hm-frontend | info: Microsoft.Hosting.Lifetime[14]
hm-frontend | Now listening on: http://[::]:80
hm-frontend | info: Microsoft.Hosting.Lifetime[0]
hm-frontend | Application started. Press Ctrl+C to shut down.
hm-frontend | info: Microsoft.Hosting.Lifetime[0]
hm-frontend | Hosting environment: Production
hm-frontend | info: Microsoft.Hosting.Lifetime[0]
hm-frontend | Content root path: /app
```

Now you can call the front-end from the client and it should work properly:

```
c:\Users\[user]\source\projects\HelloWorldClient>dotnet run
```

```
λ dotnet run  
Hello from frontend: Hello from backend: me
```

To shut down and remove the containers, open another command prompt from the directory of the **docker-compose.yml** file and issue the following command:

```
c:\Users\[user]\source\projects>docker-compose down
```