Challenge 1

By: Mosh Hamedani

Building APIs with ASP.NET Core

Build the following API endpoints:

- /api/makes
- /api/features

The first endpoint should return the list of makes and their models in the following format:

```
[
   {
      id: 1,
      name: 'Make1',
      models: [
         { id: 1, name: 'Model1' },
         …
      ]
]
```

The second endpoint should return the list of standard vehicle features in the following format:

```
[
   { id: 1, name: 'Feature1' },
   …
]
```

Challenge 1

By: Mosh Hamedani

Installing Entity Framework Core

You need to install a couple of NuGet packages.

If you're using Visual Studio, use the following command in Package Manager Console:

Install-Package <PackageName>

If you're using VSCode and terminal, you should use .NET CLI (Command-line Interface):

dotnet add package <PackageName>
dotnet restore

The packages you need to install are:

- **Microsoft.EntityFrameworkCore.Design**
- **Microsoft.EntityFrameworkCore.SqlServer**

Installing Entity Framework Commands

To create migrations and update the database, you need to install Entity Framework commands. We use these commands to create migrations and update the database. These commands are not installed by default and you need to install them manually by editing the **.csproj** file.

If you're using Visual Studio, add the following code snippet below the **<Project>** element on the top:

```
  <ItemGroup>
    <DotNetCliToolReference
         Include="Microsoft.EntityFrameworkCore.Tools"
Version="1.1.0" />
  </ItemGroup>
```

If you're using the terminal, use this code snippet instead:

```
  <ItemGroup>
    <DotNetCliToolReference
         Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
Version="1.0.0" />
  </ItemGroup>
```

Creating a DbContext

The constructor of DbContexts are different in Entity Framework Core. Instead of receiving a string as the connection string or connection string name, they receive **DbContextOptions**:

```
public VegaDbContext(DbContextOptions<VegaDbContext> options)
    : base(options)
```

You need to register your DbContext as a service for dependency injection. In the **Startup** class, add the following line in **ConfigureServices**() method:

```
services.AddDbContext<VegaDbContext>(options =>
                                        options.UseSqlServer("…"));
```

UseSqlServer() is an extension method, so you need to import the namespace for it:

```
using Microsoft.EntityFrameworkCore;
```

As the argument to **UseSqlServer()** method, you pass the connection string. Store the connection string in **appsettings.json:**

```
“ConnectionStrings”: {
    “Default”: “server=localhost; database=vega; user id=…; password=…”
}
```

If you’re running SQL Server on Windows, instead of **user id** and **password**, you specify “**integration security=SSPI**”. Otherwise, if you’re running SQL Server as a Docker image on Mac/Linux, **user id** should be **sa** and **password** should be what we set up earlier: **MyComplexPassword!234**

Now, in **ConfigureServices**, when calling the **UseSqlServer()** method, you can read the connection string from appsettings.json using:

```
Configuration.GetConnectionString(“Default”)
```

Creating Migrations

If you’re using Visual Studio, the commands are exactly like the ones in Entity Framework 6: **Add-Migration** and **Update-Database.**

If you’re using the terminal, try the following:

dotnet ef migrations add <MigrationName>
dotnet ef database update

Creating Controllers

In ASP.NET Core, you derive your controllers from the **Controller** class. Unlike ASP.NET MVC 5, we only have one **Controller** class. With this class, you can return JSON objects (for APIs) or Razor views.

Challenge 1

By: Mosh Hamedani

You need to decorate your actions with the **HttpGet** attribute so they'll be accessible as endpoints:

```
[HttpGet("/api/makes")]
public IEnumerable<Make> GetMakes() { … }
```

Note: After creating an action, the route will not be accessible until the server is restarted because routes are loaded during the application startup. So, stop the terminal process and run **dotnet watch run** again.

Using DbContext

In the **Startup** class you registered **DbContext** as a service for dependency injection. This means you can inject it into the constructor of your controllers.

When reading data, you may notice there is no ToList() method on your DbSets. You need to import **Microsoft.EntityFrameworkCore.** Then, you can call **ToListAsync** on your DbSets:

```
context.Makes.ToListAsync();
```

If you're using the traditional SQL Server as your backend, it's better to use the synchronous ToList() method. To use this, you should import System.Linq. But if you're using a scalable database like SQL Azure, use async methods.

Mapping

Separate your domain models from the models that your APIs work with. Use AutoMapper to map your domain classes to these models, which I refer to as "resources".

To use AutoMapper, install the following packages:

- **AutoMapper**
- **AutoMapper.Extensions.Microsoft.DependencyInjection**

Challenge 1

By: Mosh Hamedani

Register AutoMapper for dependency injection in Startup.ConfigureServices:

```
services.AddAutoMapper();
```

This is an extension method and you need to manually import the namespace.

Next, create a profile class to define your mappings.

```
public class MappingProfile : Profile { }
```

In the constructor of this class, you can define maps using **CreateMap().**

Finally, to use AutoMapper, inject the **IMapper** interface to your controllers.