Microservices in a day

Using .NET Core and AWS

Code PaLOUsa 2019

Microservice Patterns Workshop

Contents

[Chapter 1 1](#_Toc16921396)

[Creating the MVC Application 1](#_Toc16921397)

[Step 1 – Create Entities 1](#_Toc16921398)

[Step 2 – Extend Identity 3](#_Toc16921399)

[Create Identity Roles and User 3](#_Toc16921400)

[ASP.NET Dependency Injection 6](#_Toc16921401)

[Entity Framework 7](#_Toc16921402)

[Step 3 – Create ViewModels 8](#_Toc16921403)

[Step 4 – AutoMapper 10](#_Toc16921404)

[Step 5 – Repository Layer 12](#_Toc16921405)

[Chapter 2 13](#_Toc16921406)

[Chapter 3 14](#_Toc16921407)

[Chapter 4 15](#_Toc16921408)

[Chapter 5 16](#_Toc16921409)

[Chapter 6 17](#_Toc16921410)

[Chapter 7 18](#_Toc16921411)

[Chapter 8 19](#_Toc16921412)

# Chapter 1

## Creating the MVC Application

Open Visual Studio and create a new ASP.NET Core Web Application. Name the Solution `RoutineCatalogue` and name the Project `RoutineCatalogue.MVC`. Select Web Application (Model-View-Controller) and ensure your framework is set to .NET Core/ ASP.NET Core 2.2. Check Configure for HTTPS. Change Authentication to use Individual User Accounts and select `Store user accounts in-app`. Click Create.

Navigate to the `appsettings.json` file and replace the database name with `RoutineCatalogue`.



## Step 1 – Create Entities

Right click on the RoutineCatalogue Solution, select Add > New Project, select Class Library (.NET Core), name this project `RoutineCatalogue.Models`. Add a new folder named `Entities`. Delete the autogenerated `Class1.cs`. Create a new class in the Entities folder named `BaseEntities.cs`



Set.cs

Routine.cs



Exercise.cs



Create a new folder in the Models Project named Types. Add the class RoleType.cs.



## Step 2 – Extend Identity

### Create Identity Roles and User

Role.cs



User.cs



Create a new folder in the Models Project named Settings. Create a new class in that folder named ApplicationSettings.cs.



Add the ApplicationSettings to the appsettings.json file in your MVC Project.



Create a new folder in the MVC Project named Factories. Create a new class in that folder named `UserSeedFactory.cs`.



### ASP.NET Dependency Injection

Replace Default Identity Dependency Injection with new DI. In your Startup.cs Class replace the call to `AddDefaultIdentity` to the Service Collection.



Add the ApplicationSettings to the Services Collection.



Inject the AppSettings into the Configure method.



Call the Initialize method on the UserSeedFactory as the concluding action of the Configure method.



### Entity Framework

Finally lets update the ApplicationDbContext class located in the Data Folder of our MVC Project.



Now we can update the database. Delete the existing migration and database snapshot from the Data Folder. Run the following 2 command in the nuget package manager console.



As a result of changing our Identity, we need to now fix references to this in our dependency injection. Navigate to your `\_LoginPartial.cshtml` and update the dependency injection at the top of this file to reflect the user class.



Right click on the MVC Project, select Add, select New Scaffolded Item, select Identity, click Add, check Account\Register.Add the following to the OnPostAsync method’s user instantiation inside of the Register.cshtml.cs file that we just scaffolded.



Finally, add the user to the role after the user is added successfully. This will ensure you have an Administrative account and anyone else who signs up will have Trainer access.



## Step 3 – Create ViewModels

It’s best practice to not return the full model to the view, therefore we return a flattened and often concatenated POCO (Plain Old Common language runtime Object)/ DTO (Data Transfer Object). Since these Models will be returned to the Views, we call them ViewModels.

Create a new folder inside of your Models Project named ViewModels

Create BaseViewModels.cs



Create RoutineViewModel.cs



Create RoutineIndexViewModel.cs



Create ExerciseViewModel.cs



Create ExerciseIndexViewModel.cs



Create SetViewModel.cs



Create SetIndexViewModel.cs



## Step 4 – AutoMapper

AutoMapper is an Object to Object Mapper. There are 3 ways to install it. You want to specify that it’s in your MVC Project.

|  |  |
| --- | --- |
| Package Manager Console | Install-Package AutoMapper -Version 9.0.0 |
| .NET CLI | dotnet add package AutoMapper --version 9.0.0 |
| Package Reference | <PackageReference Include="AutoMapper" Version="9.0.0" /> |

The same goes for AutoMapper’s Dependency Injection Package

|  |  |
| --- | --- |
| Package Manager Console | Install-Package AutoMapper.Extensions.Microsoft.DependencyInjection -Version 7.0.0 |
| .NET CLI | dotnet add package AutoMapper.Extensions.Microsoft.DependencyInjection --version 7.0.0 |
| Package Reference | <PackageReference Include="AutoMapper.Extensions.Microsoft.DependencyInjection" Version="7.0.0" /> |

Adding AutoMapper right above the AddMvc in your Startup ConfigureServices Method will give you access to the IMapper interface from any constructor you inject it into.



Create a new Folder named AutoMapperProfiles. Add the following MappingProfile.

SetProfile.cs



ExerciseProfile.cs



RoutineProfile.cs



## Step 5 – Repository Layer

Creating a generic Repository.

IRepository.cs



Repository.cs





Dependency Injection in your MVC Projects Startup Class  


## Step 6 – Creating the Controllers

# Chapter 2

Creating the Routine recommended Service

# Chapter 3

API Authentication

# Chapter 4

Creating Workout Service

# Chapter 5

Microservice Communication

# Chapter 6

ASP.NET Core Cache

# Chapter 7

Dynamo DB

# Chapter 8

Duplicating Authentication