Microservices in a day

Using .NET Core and AWS

Code PaLOUsa 2019

https://github.com/dvdmrk

Contents

[Chapter 1 1](#_Toc17012921)

[Creating the MVC Application 1](#_Toc17012922)

[Step 1 – Create Entities 1](#_Toc17012923)

[Step 2 – Extend Identity 3](#_Toc17012924)

[Create Identity Roles and User 3](#_Toc17012925)

[ASP.NET Dependency Injection 6](#_Toc17012926)

[Entity Framework 7](#_Toc17012927)

[Step 3 – Create ViewModels 8](#_Toc17012928)

[Step 4 – AutoMapper 11](#_Toc17012929)

[Step 5 – Repository Layer 13](#_Toc17012930)

[Step 6 – Create the ViewComponents 15](#_Toc17012931)

[Step 7 – Creating the Controllers 19](#_Toc17012932)

[Step 8 – Updating the Layout 24](#_Toc17012933)

[Step 9 – Enable Auditable 27](#_Toc17012934)

[Chapter 2 29](#_Toc17012935)

[Step 1 – Setup 29](#_Toc17012936)

[Side Quest! 29](#_Toc17012937)

[Step 2 – Creating the Controllers 30](#_Toc17012938)

[Step 3 – Configure Swagger 35](#_Toc17012939)

[Chapter 3 36](#_Toc17012940)

[What is JWT 36](#_Toc17012941)

[Step 1 – Building the Models 36](#_Toc17012942)

[Step 2 – Application Secret 36](#_Toc17012943)

[Step 2 – Building the Controllers 37](#_Toc17012944)

[Step 3 – Configuring the Authentication 38](#_Toc17012945)

[Step 4 – Adding Authorization Annotations 39](#_Toc17012946)

[Step 5 – Testing with Postman 39](#_Toc17012947)

[Chapter 4 40](#_Toc17012948)

[Chapter 5 41](#_Toc17012949)

[Chapter 6 42](#_Toc17012950)

[Chapter 7 43](#_Toc17012951)

[Chapter 8 44](#_Toc17012952)

# 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 – Create the ViewComponents

ViewComponents replace PartialViews in ASP.NET Core. Right click on your MVC Project and Add, New Folder, name it ` ViewComponents`.

Create Class ExerciseListViewComponent.cs



Modify ExerciseProfile.cs to include an additional mapper from Exercise to SelectListItem.



Create a new folder named Components in RoutineCatalogue.MVC/Views/Shared then Create a new folder inside of Components named ExerciseList, then create a new View in that folder named Default.cshtml.



Create Class SetListViewComponent.cs inside the ViewComponents Folder



Create a new Folder inside of the Components Folder named SetList, then create a new View in that folder named Default.cshtml.





## Step 7 – Creating the Controllers

All of our controllers will inherit from Controller.

Add a new Class, SetController.cs to your Controllers folder.

Dependency Injection



Get



GetAll



Post



Update



Delete



Add a new Class, ExerciseController.cs to your Controllers folder.

Dependency Injection



Get



GetAll



Post



Update



Delete



Add a new Class, RoutineController.cs to your Controllers folder.

Dependency Injection



Get



GetAll



Post



Update



Delete



## Step 8 – Updating the Layout

Start off by copying the Folders in the Chapter 1 Resources Folder of this project into the Views Folder of your MVC Project.

\_Layout.cs

Move JQuery, Bootstrap, and site.js to the head of the layout page.



Add navigation to Routine and Exercise from your primary nav menu.



Add Font Awesome after the title HTML Tag.



Inject the UserManager and SignInManager into your layout page before any HTML is evaluated.



Add the Bootstrap Modal after the footer.



site.css

site.js



## Step 9 – Enable Auditable

In your Data Folder, ApplicationDbContext Class, update the constructor.



Now override the base SaveChangeaAsync Method.



# Chapter 2

Creating the Routine API. Right click on the solution. Select Add, New Project. Choose ASP.NET Core Web Application. Choose API; ensure .NET Core and ASP.NET 2.2 are selected. Click Create.

## Step 1 – Setup

Go to the launchSettings.json file located in the Properties Folder. Under the Profiles field, change launchUrl to “launchUrl”:”swagger”, for both “IIS Express” and “API”. Navigate to the appsettings.json add your connection string from the other project here.  


Now lets add our database to the Dependency Injection of the new API Project. Navigate to Startup.cs and add the context to the Services Collection.



### Side Quest!

I just realized that Set’s aren’t saving. And there’s no reason why they should. So what we need to do is create a new SetService in a new Services Folder.



Then we’ve got to configure it in the Service Collection.



Then we’ve got to add it to the SetController Dependency Injection.



Then we have to call it prior to saving the Set.



## Step 2 – Creating the Controllers

Add the generic repository DI to the API Projects Startup Class



Add the IHttpContextAccssor Required by ApplicationDbContext to the Startup.



Add AutoMapper Required by the Generic Repository to the Startup.



Delete the ValuesController. Right click on the Controllers Folder, add a new Class named SetController.cs. All of our controllers will inherit from ControllerBase because it retains all of the Controller functionality without the view support.

SetController.cs





RoutineController.cs





ExerciseController.cs





## Step 3 – Configure Swagger

Configure Swagger in the Startup.

|  |  |
| --- | --- |
| Package Manager | Install-Package Swashbuckle.AspNetCore -Version 5.0.0-rc2 |
| .NET CLI | dotnet add package Swashbuckle.AspNetCore --version 5.0.0-rc2 |
| PackageReference | <PackageReference Include="Swashbuckle.AspNetCore" Version="5.0.0-rc2" /> |

Add Swagger to the Configure Method in the Startup Class of your API Project.

Now add Swagger to the Services Collection.





# Chapter 3

API Authentication

In relation to the “Strangler Pattern” we have effectively implemented the first stage. We’ve decoupled our backend from our front end. This is also possible with the implementation of IActionResult being able to return anything. If our needs were different, we could have engineered a service that would return either a view or a JSON object.

Now, we need to secure our API Endpoints and validate that users are signed in when preforming CRUD operations on the RoutineCatalogue. We’re going to implement JSON Web Tokens as a means of assigning identity and authenticating API Users.

What is JWT – Link to slides

## Step 1 – Building the Models

We only actually need 1 model for our authentication because we aren’t extending it to include any information beyond that of MVC, or what’s required to sign in. Create a new Class in the ViewModels Folder of the RoutineCatalogue.Models Project named ApiSigninModel.cs



## Step 2 – Application Secret

Let’s add a secret property named ApplicationSecret, for authentication to our ApplicationSettings Class, and a corresponding key in our API Projects appsettings.json file.appsettings.json

## Step 2 – Building the Controllers

Create a new class in the Controllers folder of the RoutineCatalogue.API Project named UserController.cs

Constructor Injection

Signin

Signup

## Step 3 – Configuring the Authentication

The rest of the code to include Authentication exists in the Startup Class of the API Project.

Finally, lets ensure our Authentication is configured on the startup of the application by adding `app.UseAuthentication();` to the Configure method.

## Step 4 – Adding Authorization Annotations

We need to let the controllers know how to authenticate. We’ll do this by attaching an authorize data annotation to the top of each class that passes the JWT Schema as a parameter.

Add the following to the top of your SetController Class right after the Route Annotation.

Add the following above the class definition of the RoutineController. Add the following above the class definition of the ExerciseController.Finally, add the following, above the class definition of the UserController.

## Step 5 – Testing with Postman

You can now test logging in with Swagger, but any other operation will require Postman so you can send the bearer token as part of the headers.

# Chapter 4

Creating Workout Service

# Chapter 5

Microservice Communication

# Chapter 6

ASP.NET Core Cache

# Chapter 7

Dynamo DB

# Chapter 8

Duplicating Authentication