Code PaLOUsa 2019

https://github.com/dvdmrk

Microservices in a day

Using .NET Core and AWS

**Preface**

{

“**author**”: “David Merk”

“**shortBiography**”:” David Merk is a Software Engineer (QSR Automations), Mentor (Code Louisville/ JCTC), Organizer (StartUp Weekend), Volunteer (Civic Data Alliance), Speaker (CodePaLOUsa 2019), Workshop Creator (Microservices in a Day with .NET Core and AWS), and unabashed Career Changer, specializing in C# .NET, N-Tier and Microservice Architecture. Since starting his professional career as an Engineer he’s approached every obstacle as an opportunity and overcome it with passion; inventing creative and logical solutions that exceed expectations.\r\n  
His experience outside of technology has taught him to be a more effective communicator and senior. After changing career paths from College Educated English Teacher to Self-Taught Software Developer, he understands the plight of a neophyte and aims to balance content that is complex, and discriminated in an easily digestible format. He attributes his success inside the office, to his trial and error outside of the office. It's his personal belief that if you aren't failing- you aren't trying hard enough; and that message resonates with every professional solution he creates and presentation he instructs. ”,

“**links**”:

{

“linkedIn”:” <https://www.linkedin.com/in/davewritescode>”,

“github”:”<https://github.com/dvdmrk>”,

“blog”:”<https://davewritescode.com>”

},

“**purpose**”:”This Workshop is the result of all the things I wish I would have known when I started taking on microservices paired with the exceptional teaching of Chris Richardson’s ‘Microservice Patterns’ specifically Chapter 13 which discusses Refactoring a monolith.”,

“**audience**”:”This workshop is for anyone experiencing Monolith problems and wants to get an introduction to the considerations they should have in the future.”,

“**goal**”:”The goal of this workshop is part .NET Core, part Architecture, and part AWS. By the end you should be more confident implementing N-Tier Patterns alongside Microservice Patterns, specifically when it comes to communication. You should also have a better understanding of event driven development and both relational and non-relational databases.”,

“**packagesUsed**”:

{

”autoMapper”:” <https://automapper.org/>”,

“swagger”:”<https://swagger.io>”,

“restEase”:” <https://github.com/canton7/RestEase>”

},

“**specialThanks**”:”Before beginning the workshop I want to say thank you to my co-workers who have helped me with the ideation of this workshop; I want to say thank you to the people who wrote the packages that made it easier to give this presentation; I want to thank QSR for giving me time and encouragement in the creation of this workshop; and I want to thank my fiancé Shannon for encouraging me to persevere when I wanted to quit.”,

“**workWithMe**”:

{

“why”:”QSR Automations is a highly successful and always growing technology company in the Louisville area. We have an ever-progressing technology stack. We value a work/ life balance that allocates time for research, rapid prototyping, experimentation, and professional development. We have several great teams some of whom support technology used worldwide, others- like mine, who support APIs consumed by the likes of Google and OpenTable. We create have a great suite product that service the Restaurant industry, and have a great company culture.”,

“how”:” <https://www.qsrautomations.com/company/careers/>”

}

}

Contents

[Chapter 1) Creating the MVC Application 1](#_Toc17196771)

[Step 1 – Create Entities 1](#_Toc17196772)

[Step 2 – Extend Identity 2](#_Toc17196773)

[Create Identity Roles and User 2](#_Toc17196774)

[ASP.NET Dependency Injection 5](#_Toc17196775)

[Entity Framework 6](#_Toc17196776)

[Step 3 – Create ViewModels 7](#_Toc17196777)

[Step 4 – AutoMapper 9](#_Toc17196778)

[Step 5 – Repository Layer 11](#_Toc17196779)

[Step 6 – Create the ViewComponents 13](#_Toc17196780)

[Step 7 – Creating the Controllers 17](#_Toc17196781)

[Step 8 – Updating the Layout 22](#_Toc17196782)

[Step 9 – Enable Auditable 25](#_Toc17196783)

[Chapter 2) Decoupling Presentation and Logic 27](#_Toc17196784)

[Step 1 – Setup 27](#_Toc17196785)

[Side Quest! 27](#_Toc17196786)

[Step 2 – Creating the Controllers 28](#_Toc17196787)

[Step 3 – Configure Swagger 33](#_Toc17196788)

[Chapter 3) API Authentication 34](#_Toc17196789)

[What is JWT 34](#_Toc17196790)

[Step 1 – Building the Models 34](#_Toc17196791)

[Step 2 – Application Secret 34](#_Toc17196792)

[Step 2 – Building the Controllers 34](#_Toc17196793)

[Step 3 – Configuring the Authentication 36](#_Toc17196794)

[Step 4 – Adding Authorization Annotations 37](#_Toc17196795)

[Step 5 – Testing with Postman 37](#_Toc17196796)

[Chapter 4) Returning HyperMedia 38](#_Toc17196797)

[Step 1 – Creating the Models 38](#_Toc17196798)

[Step 2 – Creating the Methods 39](#_Toc17196799)

[Step 3 – Returning HyperMedia 39](#_Toc17196800)

[Recap 41](#_Toc17196801)

[Chapter 5) Communication, Cache, and Authentication 42](#_Toc17196802)

[Step 1 – Setting up RestEase 42](#_Toc17196803)

[Step 2 – Build the WorkoutService Service 43](#_Toc17196804)

[Step 3 – Build the RoutineService Service 44](#_Toc17196805)

[Step 4 – Cache Factory 45](#_Toc17196806)

[Step 5 – Add Authentication 45](#_Toc17196807)

[Chapter 6) Creating Restful APIs 47](#_Toc17196808)

[Create the Models 47](#_Toc17196809)

[Create the Hypermedia 48](#_Toc17196810)

[Create the Controllers 49](#_Toc17196811)

[Chapter 7) Dynamo DB 50](#_Toc17196812)

[Step 1 - Add the Models 50](#_Toc17196813)

[Step 2 – Add the Services 51](#_Toc17196814)

[Step 3 – Configure the DI 53](#_Toc17196815)

[Step 4 – Update the SetController to Reference the IWorkoutService instance 53](#_Toc17196816)

[Step 5 – Update the ApplicationSettings.cs 54](#_Toc17196817)

[Step 6 – Update the appsettings.json 54](#_Toc17196818)

[Chapter 8) Eventing on Update 55](#_Toc17196819)

[Step 1 - Expose API Endpoint 55](#_Toc17196820)

[Step 2 - Create Factory method to rehydrate cache 55](#_Toc17196821)

[Step 3 – Implement RestEase in the RoutineCatalogue.MVC Project 56](#_Toc17196822)

[Step 4 – Extending the Repository 57](#_Toc17196823)

[Step 5 – Implement the DI 58](#_Toc17196824)

[Step 6 – Lock down with CORS policies 58](#_Toc17196825)

# 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) Decoupling Presentation and Logic

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) Returning HyperMedia

This is a concept for Mature Restful Development. The idea behind this is that your API Endpoints should return a list of Actions that can be preformed on the object. We’re going to use composition to add HyperMedia to our API responses.

## Step 1 – Creating the Models

Start off by creating a new class in the types folder of the Models Project. Name this class HyperMediaType.cs



Next create a new Folder in your Models Project named ApiModels, then create a new class called HyperMedia.cs.



## Step 2 – Creating the Methods

Now create a new class named HyperMediaResponse.cs

Step 3 – Returning HyperMedia  
Finally, for the Controllers. Instead of simply returning ok or bad and the requested object, we’re going to return an anonymous object with our object and hypermedia.



Repeat this process for ExerciseController with a modification.

Now, lets repeat this process for the RoutineController.

# Recap

We’ve created several new projects throughout the course of this workshop. Now it’s time to create a new solution. This is going to be a second Microservice. Chapter 1 taught us about building an N-Tier application. In Chapter 2 we started on the first tenant of “the Strangler Pattern” by decoupling our frontend technology from our backend technology. In chapter 3 we implemented API Authentication. Finally, Chapter 4, we implemented the 3 Phases of a Mature Restful API on our relational data models. The remainder of this workshop will guide us towards the final 2 tenants of the Strangler Pattern: New Feature development as a Microservice, and Porting functionality as a Microservice.

Go to File, New Project, ASP.NET Core Web Application:   
Project Name: WorkoutService  
Solution Name: WorkoutService  
No Authentication is needed, as we’ll implement that ourselves in the next Chapter.  
Click Create.

# Chapter 5) Communication, Cache, and Authentication

Let’s start off by deleting the ValuesController. Then navigate to the appsettings.json file. We need to add communication with the RoutineService so we can get a list of routines. Retrieve the Port Number from the existing API Project, this can be found in the launchSettings.json file within the sslPort property. Once you’ve found that, add it to the new solutions appsettings.json file.

Now we need to bring in the RestEase nuget package. This is an Abstraction of the IHttpClient class. It saves you from writing a lot of code to make an API call by allowing you to execute it as a method after defining the in an interface, and the server in the Startup.

|  |  |
| --- | --- |
| Package Manager | Install-Package RestEase -Version 1.4.10 |
| .NET CLI | dotnet add package RestEase --version 1.4.10 |
| PackageReference | <PackageReference Include="RestEase" Version="1.4.10" /> |

## Step 1 – Setting up RestEase

Build the Models

Build the Service Interface

Dependency Injection

## Step 2 – Build the WorkoutService Service

Create a new Folder in your WorkoutService Microservice Project. Name it Models. Create a new Class in that folder named Routine.cs

Create a new Folder in your WorkoutService Microservice Project. Name it Services. Create a new Class in that folder named IRoutineService.cs

Dependency Injection

Configure RestEase in the WorkoutService Startup File.



Step 3 – Build the RoutineService Service  
Create a new Folder in your API Project named Services. In this folder add a new Class named RoutineService.csNow add this service to the Startup Services Collection.Finally, add RoutineService to the RoutineController Constructor Injection.And add a special method for returning this type of object to the RoutineController.

## Step 4 – Cache Factory

Create a new class in the WorkoutService’s Services Folder Named RoutineFactory.

Dependency Injection  
Configure the .NET Core Cache and the RoutineFactory in the WorkoutService Startup Service Collection.Add the RoutineFactory to the Configure Constructor to ensure it’s executed when the application is built.

## Step 5 – Add Authentication

Return to your RoutineService Startup and Copy the AddAuthentication method. We’re going to paste this in the WorkoutService Startup.  
Start off by adding a new Class to the Models Folder named ApplicationSettings.cs.Add a new Application Setting to your ApplicationSettings JSON in appsettings.json.

# Chapter 6) Creating Restful APIs

## Create the Models

Let’s start off by creating the Workout.cs file in the Models Folder of the WorkoutService Project.Next, let’s create the Hypermedia Classes. Start with the Hypermedia.cs in the Models Folder.CCreate the Service

Now create HypermediaService.cs in the Services Folder.

## Create the Hypermedia

Add HypermediaService to the Services Collection.

Next, we need to create a new class in the Controllers Folder, named RoutineController.cs

Create the Controllers  
Finally, lets create our SetController.cs

# Chapter 7) Dynamo DB

DynamoDB is a NoSQL Db also known as a Document DB, and is powered by Key/ Value pairs. It has single millisecond response times, and being that we’re expecting a lot more of our users to be completing routines than creating them, we needed a creative way of storing them. We will be using the UserId-RoutineId as the Key which will be available with every request, and a list of Workouts as the value. This will empower us to be able to return a users entire history per workout, or per user if we look for the partial key, and it will also empower us to gather some really big data by searching just on the RoutineId.

## Step 1 - Add the Models

Create a new class in the Models folder named WorkoutRoutineAdd another new class named WorkoutHistory

## Step 2 – Add the Services

Create a new class in the Services Folder named IWorkoutRepositoryCreate another new class in the Services Folder named WorkoutRepository

## Step 3 – Configure the DI

Add the following Services to the Startup Service Collection

Step 4 – Update the SetController to Reference the IWorkoutService instance

Constructor InjectionHttpPost

Step 5 – Update the ApplicationSettings.cs  
  


Step 6 – Update the appsettings.json  


# Chapter 8) Eventing on Update

Initially I had plans to implement the Pub/Sub pattern using AWS SNS and SQS, however due to time constraints that will have to be left for a later commit. Then I planned on demonstrating the observer pattern, but it doesn’t seem applicable when working with multiple instances of the same service. So the most fitting and least prohibitive solution I’ve found was to create a wrapper for the IRepository that will make an API call after saving in our RoutineService and force our WorkoutService to rehydrate its cache.

## Step 1 - Expose API Endpoint

RoutineController Workout Microservice

Constructor Injection  
HttpPost

Step 2 - Create Factory method to rehydrate cache



## Step 3 – Implement RestEase in the RoutineCatalogue.MVC Project

Pull in the package reference as depicted in Chapter 5.

Create a new Folder named Service in the MVC Project. Create a new class in this folder named IPublisherAdapter.cs

Create another new class in this folder named PublisherAdapter.cs

We are implementing the publisher as an interface because it will serve as an adapter when in future iterations we have a more mature Pub/Sub pattern in place.  
Create IPublisher.cs in the RoutineCatalogue.MVC Services Folder

## Step 4 – Extending the Repository

We are going to use a cool pattern called the Chain of Responsibility Pattern to make an API Post prior to saving to the database. Create a new file in the Repositories Folder of the MVC Project named PublisherRepository.cs

## Step 5 – Implement the DI

Add the IP for the WorkoutService to your appsettings.json in the RoutineCatalogue.API ProjectAdd the WorkoutServiceIP Property to the ApplicationSettings Class in the RoutineCatalogue.Models ProjectMake a few changes to the Services Collection in the RoutineCatalogue.API Projectand

## Step 6 – Lock down with CORS policies

Dependency Injection

Add this to the bottom of your Services Collector in the WorkoutServices Startup



Add this to the bottom of your Services Collector in the RoutineCatalogue.API Projects Startup

Data Annotation

Add this Data Annotation to the RoutineCatalogue.API Projects RoutineControllers GetRoutinesWithSets MethodAnd add this DataAnnotation to your WorkoutServices RoutineController Post