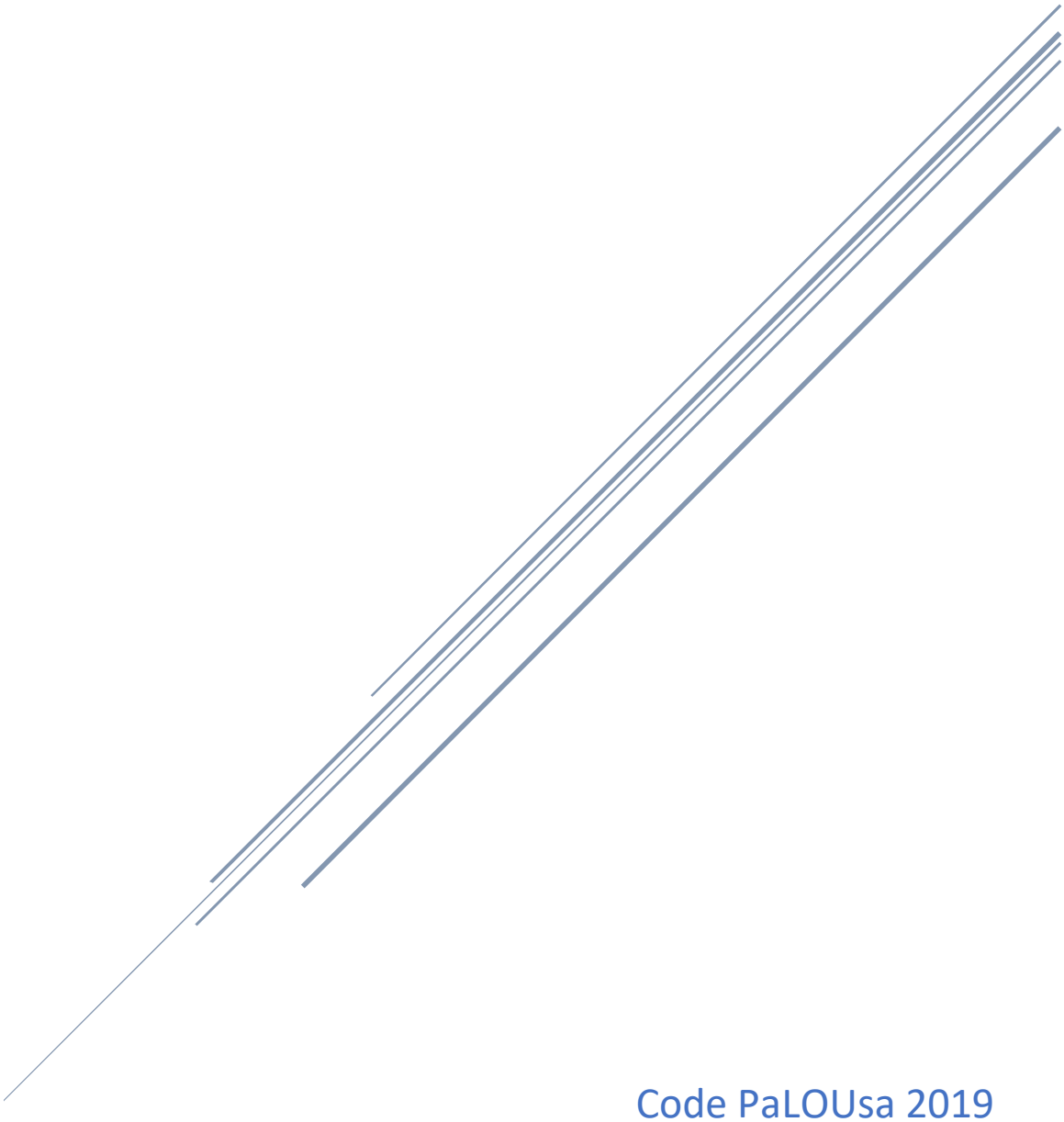
MICROSERVICES IN A DAY

Using .NET Core and AWS



Code PaLOUsa 2019
https://github.com/dvdmrk

{

“**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/"

},

"**foreword**":"The Strangler Pattern is a concept from Richardson's Microservice Pattern where a large monolithic application can be rewritten over time. It exists in three parts that we will demonstrate today: separating the presentation layer from the logic layer, creating new features as a service, recreating existing functionality in a new service."

}

Contents

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`.

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=RoutineCatalogue;Trusted_Connection=True;Multipl
eActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

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`

```
using System;
namespace RoutineCatalogue.Models.Entities
{
    public class BaseEntity
    {
        public Guid Id { get; set; }
    }
    public class AuditableEntity : BaseEntity
    {
        public DateTime CreateDate { get; set; }
        public DateTime? UpdateDate { get; set; }
        public User CreateBy { get; set; }
        public User UpdateBy { get; set; }
    }
    public class NamedAuditableEntity : AuditableEntity
    {
        public string Name { get; set; }
        public string Description { get; set; }
    }
}
```

Set.cs

```
namespace RoutineCatalogue.Models.Entities
{
    public class Set : AuditableEntity
    {
        public Exercise Exercise { get; set; }
        public Routine Routine { get; set; }
        public int? Repetitions { get; set; }
        public double? Weight { get; set; }
        public int Order { get; set; }
    }
}
```

Routine.cs

```
using System.Collections.Generic;
namespace RoutineCatalogue.Models.Entities
{
    public class Routine : NamedAuditableEntity
    {
        public ICollection<Set> Sets { get; set; }
    }
}
```

Exercise.cs

```
namespace RoutineCatalogue.Models.Entities
{
    public class Exercise : NamedAuditableEntity {}
}
```

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

```
namespace RoutineCatalogue.Models.Types
{
    public enum RoleType
    {
        Admin,
        Trainer,
        User
    }
}
```

Step 2 – Extend Identity

Create Identity Roles and User

Role.cs

```
using Microsoft.AspNetCore.Identity;
using System;
namespace RoutineCatalogue.Models.Entities
{
    public class Role : IdentityRole<Guid>
    {
        public Role() : base() { }
        public Role(string roleName) : base(roleName) { }
        public Role(string roleName, string description, DateTime creationDate) :
base(roleName)
        {
            this.Description = description;
            this.CreationDate = creationDate;
        }
        public string Description { get; set; }
        public DateTime CreationDate { get; set; }
    }
}
```

User.cs

```
using Microsoft.AspNetCore.Identity;
using System;
namespace RoutineCatalogue.Models.Entities
{
    public class User : IdentityUser<Guid>
    {
        public User() : base() { }
        public Role Role { get; set; }
    }
}
```

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

```
namespace RoutineCatalogue.Models.Settings
{
    public class ApplicationSettings
    {
        public string AdminUsername { get; set; }
        public string AdminPassword { get; set; }
    }
}
```

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

```
  "ApplicationSettings": {
    "AdminUsername": "     ",
    "AdminPassword": "    "
  },
```

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

```
using Microsoft.AspNetCore.Identity;
using RoutineCatalogue.Models.Entities;
using RoutineCatalogue.Models.Settings;
using RoutineCatalogue.Models.Types;
using RoutineCatalogue.MVC.Data;
using System;
using System.Threading.Tasks;
namespace RoutineCatalogue.MVC.Factories
{
    public class UserSeedFactory
    {
        public static async Task Initialize(ApplicationDbContext context,
UserManager<User> userManager, RoleManager<Role> roleManager, ApplicationSettings
appSettings)
        {
            context.Database.EnsureCreated();
            var password = appSettings.AdminPassword;
            var userName = appSettings.AdminUsername;
            foreach (string role in Enum.GetNames(typeof(RoleType)))
                if (await roleManager.FindByNameAsync(role) == null)
                    await roleManager.CreateAsync(new Role(role, $"This is the {role}
role.", DateTime.Now));
            if (await userManager.FindByNameAsync(userName) == null)
            {
                var user = new User
                {
                    UserName = userName,
                    Email = userName,
                    Role = await roleManager.FindByNameAsync(RoleType.Admin.ToString())
                };
                var result = await userManager.CreateAsync(user);
                if (result.Succeeded)
                {
                    await userManager.AddPasswordAsync(user, password);
                    await userManager.AddToRoleAsync(user, RoleType.Admin.ToString());
                    context.SaveChanges();
                }
            }
        }
    }
}
```

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.

```
services.AddDefaultIdentity<IdentityUser>()
    .AddDefaultUI(UIFramework.Bootstrap4)
    .AddEntityFrameworkStores<ApplicationDbContext>();
services.AddIdentity<User, Role>(options =>
{
    options.User.RequireUniqueEmail = true;
})
.AddEntityFrameworkStores<ApplicationDbContext>()
.AddUserStore<UserStore<User, Role, ApplicationDbContext, Guid>>()
.AddRoleStore<RoleStore<Role, ApplicationDbContext, Guid>>()
.AddDefaultUI()
.AddDefaultTokenProviders();
```

Add the ApplicationSettings to the Services Collection.

```
services.Configure<ApplicationSettings>(Configuration.GetSection("ApplicationSettings"));
```

Inject the AppSettings into the Configure method.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
ApplicationDbContext context, RoleManager<Role> roleManager, UserManager<User>
userManager, IOptions<ApplicationSettings> appSettings)
```

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

```
UserSeedFactory.Initialize(context, userManager, roleManager, appSettings.Value).Wait();
```

Entity Framework

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

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using RoutineCatalogue.Models.Entities;
using System;
namespace RoutineCatalogue.MVC.Data
{
    public class ApplicationDbContext : IdentityDbContext<User, Role, Guid>
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) :
base(options) { }
        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
            //Generate Ids on Add
            builder.Entity<Routine>(b => { b.Property(u => u.Id).ValueGeneratedOnAdd();
});
            builder.Entity<Exercise>(b => { b.Property(u => u.Id).ValueGeneratedOnAdd();
});
            builder.Entity<Set>(b => { b.Property(u => u.Id).ValueGeneratedOnAdd(); });
            //One to Many Relationships
            builder.Entity<Set>().HasOne(c => c.Routine).WithMany(e => e.Sets);
        }
        public DbSet<Routine> Routine { get; set; }
        public DbSet<Exercise> Exercise { get; set; }
        public DbSet<Set> Set { get; set; }
        public DbSet<User> User { get; set; }
    }
}
```

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.

```
Add-Migration -Project RoutineCatalogue.MVC "InitialMigration"
```

```
Update-Database -Project RoutineCatalogue.MVC
```

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.

```
@using Microsoft.AspNetCore.Identity
@using RoutineCatalogue.Models.Entities

@inject SignInManager<User> SignInManager
@inject UserManager<User> UserManager
```

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.

```
var user = new User { UserName = Input.Email, Role = await
_roleManager.FindByNameAsync(RoleType.Trainer.ToString()), Email = Input.Email };
```

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.

```
var result = await _userManager.CreateAsync(user, Input.Password);
if (result.Succeeded)
{
    await _userManager.AddToRoleAsync(user, RoleType.Trainer.ToString());
    _logger.LogInformation("User created a new account with password.");
```

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

```
using System;
namespace RoutineCatalogue.Models.ViewModels
{
    public class BaseViewModel
    {
        public Guid Id { get; set; }
    }
    public class NamedViewModel : BaseViewModel
    {
        public string Name { get; set; }
        public string Description { get; set; }
    }
}
```

Create RoutineViewModel.cs

```
namespace RoutineCatalogue.Models.ViewModels
{
    public class RoutineViewModel : NamedViewModel { }
}
```

Create RoutineIndexViewModel.cs

```
using System;
using System.ComponentModel.DataAnnotations;
namespace RoutineCatalogue.Models.ViewModels
{
    public class RoutineIndexViewModel : NamedViewModel
    {
        [Display(Name = "Updated By")]
        public string UpdateBy { get; set; }
        [Display(Name = "Updated On")]
        public DateTime UpdateDate { get; set; }
    }
}
```

Create ExerciseViewModel.cs

```
namespace RoutineCatalogue.Models.ViewModels
{
    public class ExerciseViewModel : NamedViewModel { }
}
```

Create ExerciseIndexViewModel.cs

```
using System;
using System.ComponentModel.DataAnnotations;
namespace RoutineCatalogue.Models.ViewModels
{
    public class ExerciseIndexViewModel : NamedViewModel
    {
        [Display(Name = "Updated By")]
        public string UpdateBy { get; set; }
        [Display(Name = "Updated On")]
        public DateTime UpdateDate { get; set; }
    }
}
```

Create SetViewModel.cs

```
using System;
using System.ComponentModel.DataAnnotations;
namespace RoutineCatalogue.Models.ViewModels
{
    public class SetViewModel : BaseViewModel
    {
        [Display(Name = "Exercise Name")]
        public string Exercise { get; set; }
        [Display(Name = "Recommended # of Repetitions")]
        public int? Repetitions { get; set; }
        [Display(Name = "Recommended % of Max Weight")]
        public double? Weight { get; set; }
        public Guid RoutineId { get; set; }
        public Guid ExerciseId { get; set; }
    }
}
```

Create SetIndexViewModel.cs

```
using System.ComponentModel.DataAnnotations;
namespace RoutineCatalogue.Models.ViewModels
{
    public class SetIndexViewModel : BaseViewModel
    {
        [Display(Name="Exercise")]
        public string Exercise { get; set; }
        [Display(Name = "Recommended # of Repetitions")]
        public int? Repetitions { get; set; }
        [Display(Name = "Recommended % of Max Weight")]
        public double? Weight { get; set; }
    }
}
```

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.

```
services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());
services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

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

SetProfile.cs

```
using AutoMapper;
using RoutineCatalogue.Models.Entities;
using RoutineCatalogue.Models.ViewModels;
namespace RoutineCatalogue.Models.AutoMapperProfiles
{
    public class SetProfile : Profile
    {
        public SetProfile()
        {
            CreateMap<Set, SetViewModel>()
                .ForMember(dest => dest.Exercise, map => map.MapFrom(source =>
source.Exercise.Name))
                .ForMember(dest => dest.RoutineId, map => map.MapFrom(source =>
source.Routine.Id))
                .ForMember(dest => dest.ExerciseId, map => map.MapFrom(source =>
source.Exercise.Id));
            CreateMap<SetViewModel, Set>()
                .ForMember(dest => dest.Exercise, map => map.MapFrom(source =>
source.ExerciseId))
                .ForMember(dest => dest.Routine, map => map.MapFrom(source =>
source.RoutineId))
                .ForMember(dest => dest.UpdateDate, map => map.Ignore())
                .ForMember(dest => dest.UpdateBy, map => map.Ignore())
                .ForMember(dest => dest.CreateBy, map => map.Ignore());
            CreateMap<Set, SetIndexViewModel>()
                .ForMember(dest => dest.Exercise, map => map.MapFrom(source =>
source.Exercise.Name));
        }
    }
}
```

ExerciseProfile.cs

```
using AutoMapper;
using RoutineCatalogue.Models.Entities;
using RoutineCatalogue.Models.ViewModels;
namespace RoutineCatalogue.MVC.AutoMapperProfiles
{
    public class ExerciseProfile : Profile
    {
        public ExerciseProfile()
        {
            CreateMap<Exercise, ExerciseViewModel>();
            CreateMap<ExerciseViewModel, Exercise>()
                .ForMember(dest => dest.UpdateDate, map => map.Ignore())
                .ForMember(dest => dest.UpdateBy, map => map.Ignore())
                .ForMember(dest => dest.CreateBy, map => map.Ignore());
            CreateMap<Exercise, ExerciseIndexViewModel>();
        }
    }
}
```

RoutineProfile.cs

```
using AutoMapper;
using RoutineCatalogue.Models.Entities;
using RoutineCatalogue.Models.ViewModels;
namespace RoutineCatalogue.MVC.AutoMapperProfiles
{
    public class RoutineProfile : Profile
    {
        public RoutineProfile()
        {
            CreateMap<Routine, RoutineViewModel>();
            CreateMap<RoutineViewModel, Routine>()
                .ForMember(dest => dest.CreateDate, map => map.Ignore())
                .ForMember(dest => dest.UpdateDate, map => map.Ignore())
                .ForMember(dest => dest.UpdateBy, map => map.Ignore())
                .ForMember(dest => dest.CreateBy, map => map.Ignore());
            CreateMap<Routine, RoutineIndexViewModel>();
        }
    }
}
```

Step 5 – Repository Layer

Creating a generic Repository.

IRepository.cs

```
using RoutineCatalogue.Models.Entities;
using RoutineCatalogue.Models.ViewModels;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
namespace RoutineCatalogue.MVC.Repositories
{
    public interface IRepository<Model, ViewModel, IndexVieModel> where Model :
BaseEntity where ViewModel : BaseViewModel
    {
        ViewModel Get(Guid id);
        Task<List<IndexVieModel>> GetAll();
        Guid Add(Model entity);
        void Update(ViewModel viewModel);
        void Delete(Guid id);
        bool EntityExists(Guid id);
    }
}
```

Repository.cs

```
using AutoMapper;
using Microsoft.EntityFrameworkCore;
using RoutineCatalogue.Models.Entities;
using RoutineCatalogue.Models.ViewModels;
using RoutineCatalogue.MVC.Data;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
namespace RoutineCatalogue.MVC.Repositories
{
    public class Repository<Model, ViewModel, IndexVieModel> : IRepository<Model,
ViewModel, IndexVieModel> where Model : BaseEntity where ViewModel : BaseViewModel
    {
        DbSet<Model> _context;
        IQueryable<Model> _reader;
        IMapper _mapper;
        Func<Task<int>> _saveChanges;
        public Repository(ApplicationDbContext context, IMapper mapper)
        {
            _context = context.Set<Model>();
            switch (typeof(Model).ToString())
            {
                case "Set":
                    _reader = (context.Set<Set>().Include(x => x.Exercise).Include(x =>
x.Routine)) as IQueryable<Model>;
                    break;
                case "Exercise":
                    _reader = (context.Set<Exercise>().Include(x => x.CreateBy).Include(x
=> x.UpdateBy)) as IQueryable<Model>;
                    break;
                case "Routine":
                    _reader = (context.Set<Routine>().Include(x => x.CreateBy).Include(x
=> x.UpdateBy).Include(x => x.Sets).ThenInclude(x => x.Exercise)) as IQueryable<Model>;
                    break;
            }
            _saveChanges = new Func<Task<int>>(() => context.SaveChangesAsync());
        }
```

```
        public Guid Add(Model entity)
        {
            _context.Add(entity);
            _saveChanges();
            return entity.Id;
        }
        public void Delete(Guid id)
        {
            _context.Remove(_context.Find(id));
            _saveChanges();
        }
        public bool EntityExists(Guid id)
        {
            return _context.Any(e => e.Id == id);
        }
        public ViewModel Get(Guid id)
        {
            var entity = _reader.FirstOrDefault(x => x.Id == id);
            return _mapper.Map<ViewModel>(entity);
        }
        public Task<List<IndexVieModel>> GetAll()
        {
            return _reader.Select(e => _mapper.Map<IndexVieModel>(e)).ToListAsync();
        }
        public void Update(ViewModel viewModel)
        {
            var entity = _reader.FirstOrDefault(x => x.Id == viewModel.Id);
            _mapper.Map(viewModel, entity);
        }
    }
}
```

Dependency Injection in your MVC Projects Startup Class

```
services.AddScoped(typeof(IRepository<,,>), typeof(Repository<,,>));
services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());
services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

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

```
using AutoMapper;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using RoutineCatalogue.Models.Entities;
using RoutineCatalogue.MVC.Data;
using System.Linq;
using System.Threading.Tasks;
namespace RoutineCatalogue.MVC.ViewComponents
{
    public class ExerciseListViewComponent : ViewComponent
    {
        ApplicationDbContext _context;
        IMapper _mapper;
        public ExerciseListViewComponent(ApplicationDbContext context, IMapper mapper)
        {
            _context = context;
            _mapper = mapper;
        }
        public async Task<IViewComponentResult> InvokeAsync()
        {
            return View(await _context.Set<Exercise>().Select(e =>
_mapper.Map<SelectListItem>(e)).ToListAsync());
        }
    }
}
```

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

```
CreateMap<Exercise, SelectListItem>()
    .ForMember(dest => dest.Value, map => map.MapFrom(source => source.Id))
    .ForMember(dest => dest.Text, map => map.MapFrom(source => source.Name));
```

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.

```
@model IEnumerable<SelectListItem>
<div class="col-md-12">
    <label class="control-label">Exercises</label>
</div>
<div class="col-md-8">
    <select name="ExerciseId" asp-items="@Model" class="form-control"></select>
</div>
<div class="col-md-4">
    <button type="button" class="btn btn-primary btn-block" onclick="addToSetList();">Add
Set <i class="fas fa-plus-circle"></i></button>
</div>
<script>
    function addToSetList() {
        $.ajax({
            type: "POST",
            url: "/Set/Create",
            data: { routineId: $("#Id").val(), exerciseId: $("[name=ExerciseId]").val()
},
            success: function (data) {
                $("#SetList").replaceWith(data)
            }
        });
    }
</script>
```

Create Class SetListViewComponent.cs inside the ViewComponents Folder

```
using AutoMapper;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using RoutineCatalogue.Models.Entities;
using RoutineCatalogue.Models.ViewModels;
using RoutineCatalogue.MVC.Data;
using System;
using System.Linq;
using System.Threading.Tasks;
namespace RoutineCatalogue.MVC.ViewComponents
{
    public class SetListViewComponent : ViewComponent
    {
        ApplicationDbContext _context;
        IMapper _mapper;
        public SetListViewComponent(ApplicationDbContext context, IMapper mapper)
        {
            _mapper = mapper;
            _context = context;
        }
        public async Task<IViewComponentResult> InvokeAsync(Guid id)
        {
            return View(await _context.Set<Set>().Include(e => e.Exercise).Where(e =>
e.Routine.Id == id).Select(e => _mapper.Map<SetIndexViewModel>(e)).ToListAsync());
        }
    }
}
```

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

```
@using RoutineCatalogue.Models.ViewModels
@model IEnumerable<SetIndexViewModel>

@{
    var isDetails = ViewContext.RouteData.Values["Action"].ToString() == "Details";
}
@if (Model.Count() > 0)
{
    <div id="SetList">

        <table class="table">
            <thead>
                <tr>
                    <th>Exercise</th>
                    <th>Repetitions</th>
                    <th>Weight</th>
                    @if (isDetails)
                    {
                        <th>Actions</th>
                    }
                </tr>
            </thead>
            <tbody>
                @foreach (var item in Model)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Exercise)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Repetitions)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Weight)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Exercise)
                        </td>
                        @if (isDetails)
                        {
```

```
                            <td>
                                <span class="actions">
                                    <a onclick="populateWindow(`Set`, `Edit`,
`@item.Id`);" data-toggle="modal" data-target="#window"><i class="fas fa-edit"></i></a>
                                    <a onclick="populateWindow(`Set`, `Details`,
`@item.Id`);" data-toggle="modal" data-target="#window"><i class="fas fa-info-
circle"></i></a>
                                    <a onclick="populateWindow(`Set`, `Delete`,
`@item.Id`);" data-toggle="modal" data-target="#window"><i class="far fa-trash-
alt"></i></a>
                                </span>
                            </td>
                        }

                    </tr>
                }
            </tbody>
        </table>
    </div>
}
```

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

```
IRepository<Set, SetViewModel, SetIndexViewModel> _repo;
public SetController(IRepository<Set, SetViewModel, SetIndexViewModel> repo)
{
    _repo = repo;
}
```

Get

```
public IActionResult Details(Guid? id)
{
    if (id == null) return NotFound();
    var set = _repo.Get((Guid)id);
    if (set == null) return NotFound();
    return PartialView(set);
}
```

GetAll

```
public IActionResult Index(Guid id)
{
    return ViewComponent("SetList", id);
}
```

Post

```
[HttpPost]
public IActionResult Create(Guid routineId, Guid exerciseId)
{
    _repo.Add(new Set { Routine = new Routine { Id = routineId }, Exercise = new Exercise
{ Id = exerciseId } });
    return ViewComponent("Set", routineId);
}
```

Update

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Edit(SetViewModel set)
{
    if (ModelState.IsValid)
    {
        try
        {
            _repo.Update(set);
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!_repo.EntityExists(set.Id)) return NotFound();
            else throw;
        }
        return RedirectToAction(nameof(Edit), "Routine", new { Id = set.RoutineId });
    }
    return View(set);
}
```

Delete

```
public IActionResult Delete(Guid? id)
{
    if (id == null) return NotFound();
    var set = _repo.Get((Guid)id);
    if (set == null) return NotFound();
    return PartialView(set);
}
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public IActionResult DeleteConfirmed(Guid id)
{
    var entity = _repo.Get(id);
    _repo.Delete(id);
    return RedirectToAction(nameof(Edit), "Routine", entity.RoutineId);
}
```

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

Dependency Injection

```
IRepository<Exercise, ExerciseViewModel, ExerciseIndexViewModel> _repo;
public ExerciseController(IRepository<Exercise, ExerciseViewModel,
ExerciseIndexViewModel> repo)
{
    _repo = repo;
}
```

Get

```
public IActionResult Details(Guid? id)
{
    if (id == null) return NotFound();
    var exercise = _repo.Get((Guid)id);
    if (exercise == null) return NotFound();
    return PartialView(exercise);
}
```

GetAll

```
public async Task<IActionResult> Index()
{
    return View(await _repo.GetAll());
}
```

Post

```
public IActionResult Create()
{
    return PartialView();
}
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Create(Exercise exercise)
{
    if (ModelState.IsValid)
    {
        _repo.Add(exercise);
        return RedirectToAction(nameof(Index));
    }
    return View(exercise);
}
```

Update

```
public IActionResult Edit(Guid? id)
{
    if (id == null) return NotFound();
    var exercise = _repo.Get((Guid)id);
    if (exercise == null) return NotFound();
    return PartialView(exercise);
}
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Edit(ExerciseViewModel exercise)
{
    if (ModelState.IsValid)
    {
        try
        {
            _repo.Update(exercise);
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!_repo.EntityExists(exercise.Id)) return NotFound();
            else throw;
        }
        return RedirectToAction(nameof(Index));
    }
    return View(exercise);
}
```

Delete

```
public IActionResult Delete(Guid? id)
{
    if (id == null) return NotFound();
    var exercise = _repo.Get((Guid)id);
    if (exercise == null) return NotFound();
    return PartialView(exercise);
}
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public IActionResult DeleteConfirmed(Guid id)
{
    _repo.Delete(id);
    return RedirectToAction(nameof(Index));
}
```

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

Dependency Injection

```
IRepository<Routine, RoutineViewModel, RoutineIndexViewModel> _repo;
public RoutineController(IRepository<Routine, RoutineViewModel, RoutineIndexViewModel>
repo)
{
    _repo = repo;
}
```

Get

```
public IActionResult Details(Guid? id)
{
    if (id == null) return NotFound();
    var exercise = _repo.Get((Guid)id);
    if (exercise == null) return NotFound();
    return View(exercise);
}
```

GetAll

```
public async Task<IActionResult> Index()
{
    return View(await _repo.GetAll());
}
```

Post

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Create(Routine routine)
{
    if (ModelState.IsValid)
        return RedirectToAction("Edit", new { id = _repo.Add(routine) });
    return View(routine);
}
public IActionResult Edit(Guid? id)
{
    if (id == null) return NotFound();
    var exercise = _repo.Get((Guid)id);
    if (exercise == null) return NotFound();
    return View(exercise);
}
```

Update

```
public IActionResult Edit(Guid? id)
{
    if (id == null) return NotFound();
    var exercise = _repo.Get((Guid)id);
    if (exercise == null) return NotFound();
    return View(exercise);
}
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Edit(RoutineViewModel routine)
{
    if (ModelState.IsValid)
    {
        try
        {
            _repo.Update(routine);
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!_repo.EntityExists(routine.Id)) return NotFound();
            else throw;
        }
        return RedirectToAction(nameof(Index));
    }
    return View(routine);
}
```

Delete

```
public IActionResult Delete(Guid? id)
{
    if (id == null) return NotFound();
    var routine = _repo.Get((Guid)id);
    if (routine == null) return NotFound();
    return PartialView(routine);
}
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public IActionResult DeleteConfirmed(Guid id)
{
    _repo.Delete(id);
    return RedirectToAction(nameof(Index));
}
```

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.

```
    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.1.3/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position"
asp-fallback-test-value="absolute"
              crossorigin="anonymous"
              integrity="sha256-eSi1q2PG6J7g7ib17yAaWMcrr5GrtohYChqibrV7PBE=" />
        <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery"
                crossorigin="anonymous"
                integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=">
        </script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.1.3/js/bootstrap.bundle.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn &&
window.jQuery.fn.modal"
                crossorigin="anonymous"
                integrity="sha256-E/V4cWE4qvAeO5MOhjtGtqDzPndRO1LBk8lJ/PR7CA4=">
        </script>
    </environment>
    <script src="~/js/site.js" asp-append-version="true"></script>
```

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

```
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Index">Home</a>
</li>
@if (SignInManager.IsSignedIn(User))
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Routine" asp-
action="Index">Routines</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Exercise" asp-
action="Index">Exercises</a>
    </li>
}
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
</li>
```

Add Font Awesome after the title HTML Tag.

```
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css"
integrity="sha384-fnmOCqbTlWIlj8LyTjo7mOUStjsKC4pOpQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr"
crossorigin="anonymous">
```

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

```
@using RoutineCatalogue.Models.Entities
@using Microsoft.AspNetCore.Identity
@inject SignInManager<User> SignInManager
@inject UserManager<User> UserManager
```

Add the Bootstrap Modal after the footer.

```
<div class="modal fade" id="window" tabindex="-1" role="dialog" aria-
labelledby="windowLabel" aria-hidden="true">
    <div class="modal-dialog" role="document">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title" id="windowLabel">Modal title</h5>
                <button type="button" class="close" data-dismiss="modal" aria-
label="Close">
                    <span aria-hidden="true"></span>
                </button>
            </div>
            <div class="modal-body">
            </div>
        </div>
    </div>
</div>
```

site.css

```
/*RoutineCatalogue Specific Styles*/
#window .form-group {
    width: 100% !important;
}

.actions, .actions button {
    font-size: 20px;
    font-weight: bold;
}

label.control-label {
    font-weight: bold;
}

div#SetList {
    width: 100%;
}

#SetList th:nth-child(2) {
    width: 100px;
}

form {
    width: 98% !important;
}
```

site.js

```
function populateWindow(controller, action, id) {
    var url = "/" + controller;
    url += "/" + action;
    url += id != undefined ? "/" + id : "";
    $("#window .modal-title").text(controller + " " + action);
    $.get(url, function (data) {
        $("#window .modal-body").html(data);
    });
}
```

Step 9 – Enable Auditable

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

```
private readonly IHttpContextAccessor _httpContextAccessor;
public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options,
IHttpContextAccessor httpContextAccessor) : base(options)
{
      _httpContextAccessor = httpContextAccessor;
}
```

Now override the base SaveChangeaAsync Method.

```
public override Task<int> SaveChangesAsync(CancellationToken cancellationToken = default)
{
    foreach (var change in base.ChangeTracker.Entries().ToList())
        if (change.Entity is BaseAuditableEntity) {
            var identity = (_httpContextAccessor.HttpContext.User.Identity as
ClaimsIdentity).FindFirst(ClaimTypes.NameIdentifier);
            if (change.State == EntityState.Added) {
                (change.Entity as BaseAuditableEntity).CreateDate = DateTime.Now;
                (change.Entity as BaseAuditableEntity).CreateBy =
Set<User>().Find(Guid.Parse(identity.Value));
            }
            else if (change.State == EntityState.Modified)
            {
                (change.Entity as BaseAuditableEntity).UpdateDate = DateTime.Now;
                (change.Entity as BaseAuditableEntity).UpdateBy =
Set<User>().Find(Guid.Parse(identity.Value));
            }
        }
    return base.SaveChangesAsync(cancellationToken);
}
```

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.

```
"ConnectionStrings": {
  "Development": "Server=(localdb)\\mssqllocaldb; Database=FandomFitnessDB;
Trusted_Connection=True;MultipleActiveResultSets=True;"
},
```

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.

```
services.AddDbContext<ApplicationDbContext>(opt =>
opt.UseSqlServer(Configuration.GetConnectionString("Development")));
```

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.

```
using RoutineCatalogue.Models.Entities;
using RoutineCatalogue.Models.ViewModels;
using RoutineCatalogue.MVC.Data;
namespace RoutineCatalogue.MVC
{
    public class SetService
    {
        ApplicationDbContext _context;
        public SetService(ApplicationDbContext context)
        {
            _context = context;
        }
        public Set CreateSet(SetViewModel vm)
        {
            return new Set
            {
                Exercise = _context.Set<Exercise>().Find(vm.ExerciseId),
                Routine = _context.Set<Routine>().Find(vm.RoutineId)
            };
        }
    }
}
```

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

```
services.AddScoped<SetService>();
```

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

```
IRepository<Set, SetViewModel, SetIndexViewModel> _repo;
SetService _service;
public SetController(IRepository<Set, SetViewModel, SetIndexViewModel> repo, SetService
service)
{
    _repo = repo;
    _service = service;
}
```

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

```
_repo.Add(_service.CreateSet(vm));
```

Step 2 – Creating the Controllers

Add the generic repository DI to the API Projects Startup Class

```
services.AddScoped(typeof(IRepository<,,>), typeof(Repository<,,>));
```

Add the IHttpContextAccssor Required by ApplicationDbContext to the Startup.

```
services.TryAddSingleton<IHttpContextAccessor, HttpContextAccessor>();
```

Add AutoMapper Required by the Generic Repository to the Startup.

```
services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());
```

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

```
using Microsoft.AspNetCore.Mvc;
using RoutineCatalogue.Models.Entities;
using RoutineCatalogue.Models.ViewModels;
using RoutineCatalogue.MVC.Repositories;
using System;
namespace RoutineCatalogue.API.Controllers
{
    public class ExerciseController : ControllerBase
    {
        IRepository<Exercise, ExerciseViewModel, ExerciseIndexViewModel> _repo;
        public ExerciseController(IRepository<Exercise, ExerciseViewModel,
ExerciseIndexViewModel> repo)
        {
            _repo = repo;
        }
        [HttpGet("{id}")]
        public IActionResult Get(Guid id)
        {
            try
            {
                return Ok(_repo.Get(id));
            }
            catch (Exception e)
            {
                return BadRequest(new { e.Message });
            }
        }
        [HttpPost]
        public IActionResult Post([FromBody] Exercise body)
        {
            try
            {
                return Ok(_repo.Add(body));
            }
            catch (Exception e)
            {
                return BadRequest(new { e.Message });
            }
        }        [HttpPut]
        public IActionResult Put([FromBody] ExerciseViewModel body)
        {
            try
            {
                _repo.Update(body);
                return Ok();
            }
            catch (Exception e)
            {
                return BadRequest(new { e.Message });
            }
        }
```

```
        [HttpDelete("{id}")]
        public IActionResult Delete(Guid id)
        {
            try
            {
                _repo.Delete(id);
                return Ok();
            }
            catch (Exception e)
            {
                return BadRequest(new { e.Message });
            }
        }
    }
}
```

RoutineController.cs

```
using Microsoft.AspNetCore.Mvc;
using RoutineCatalogue.Models.Entities;
using RoutineCatalogue.Models.ViewModels;
using RoutineCatalogue.MVC.Repositories;
using System;
namespace RoutineCatalogue.API.Controllers
{
    public class RoutineController : ControllerBase
    {
        IRepository<Routine, RoutineViewModel, RoutineIndexViewModel> _repo;
        public RoutineController(IRepository<Routine, RoutineViewModel,
RoutineIndexViewModel> repo)
        {
            _repo = repo;
        }
        [HttpGet("{id}")]
        public IActionResult Get(Guid id)
        {
            try
            {
                return Ok(_repo.Get(id));
            }
            catch (Exception e)
            {
                return BadRequest(new { e.Message });
            }
        }
        [HttpPost]
        public IActionResult Post([FromBody] Routine body)
        {
            try
            {
                return Ok(_repo.Add(body));
            }
            catch (Exception e)
            {
                return BadRequest(new { e.Message });
            }
        }
```

```
        [HttpPut]
        public IActionResult Put([FromBody] RoutineViewModel body)
        {
            try
            {
                _repo.Update(body);
                return Ok();
            }
            catch (Exception e)
            {
                return BadRequest(new { e.Message });
            }
        }
        [HttpDelete("{id}")]
        public IActionResult Delete(Guid id)
        {
            try
            {
                _repo.Delete(id);
                return Ok();
            }
            catch (Exception e)
            {
                return BadRequest(new { e.Message });
            }
        }
        [HttpGet]
        public async Task<IActionResult> Get()
        {
            return Ok(await _repo.GetAll());
        }
    }
}
```

ExerciseController.cs

```
using Microsoft.AspNetCore.Mvc;
using RoutineCatalogue.Models.Entities;
using RoutineCatalogue.Models.ViewModels;
using RoutineCatalogue.MVC.Repositories;
using System;
namespace RoutineCatalogue.API.Controllers
{
    public class ExerciseController : ControllerBase
    {
        IRepository<Exercise, ExerciseViewModel, ExerciseIndexViewModel> _repo;
        public ExerciseController(IRepository<Exercise, ExerciseViewModel,
ExerciseIndexViewModel> repo)
        {
            _repo = repo;
        }
        [HttpGet]
        public async Task<IActionResult> Get()
        {
            return Ok(await _repo.GetAll());
        }
        [HttpGet("{id}")]
        public IActionResult Get(Guid id)
        {
            try
            {
                return Ok(_repo.Get(id));
            }
            catch (Exception e)
            {
                return BadRequest(new { e.Message });
            }
        }
        [HttpPost]
        public IActionResult Post([FromBody] Exercise body)
        {
            try
            {
                return Ok(_repo.Add(body));
            }
            catch (Exception e)
            {
                return BadRequest(new { e.Message });
            }
        }
        [HttpPut]
        public IActionResult Put([FromBody] ExerciseViewModel body)
        {
            try
            {
                _repo.Update(body);
                return Ok();
            }
```

```
            catch (Exception e)
            {
                return BadRequest(new { e.Message });
            }
        }
        [HttpDelete("{id}")]
        public IActionResult Delete(Guid id)
        {
            try
            {
                _repo.Delete(id);
                return Ok();
            }
            catch (Exception e)
            {
                return BadRequest(new { e.Message });
            }
        }
    }
}
```

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.

```
app.UseSwagger();
app.UseSwaggerUI(c => {
   c.SwaggerEndpoint("/swagger/v1/swagger.json", "Routine API V1");
});
```

Now add Swagger to the Services Collection.

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo
    {
        Version = "v1",
        Title = "Routine API",
        Description = "Create a routine, add an Exercise and Set. This is the back end."
    });
});
```

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

```
namespace RoutineCatalogue.Models.ViewModels
{
    public class ApiSigninModel
    {
        public string Email { get; set; }
        public string Password { get; set; }
    }
}
```

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.

```
namespace RoutineCatalogue.Models.Settings
{
    public class ApplicationSettings
    {
        public string AdminUsername { get; set; }
        public string AdminPassword { get; set; }
        public string ApplicationSecret { get; set; }
    }
}
```

appsettings.json

```
  "ApplicationSettings": {
    "ApplicationSecret": "CodePaLOUsa2019!"
  },
```

Step 2 – Building the Controllers

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

Constructor Injection

```
    public class UserController : ControllerBase
    {
        private readonly RoleManager<Role> _roleManager;
        private readonly UserManager<User> _userManager;
        private readonly ApplicationSettings _appSettings;

        public UserController(RoleManager<Role> roleManager, UserManager<User>
userManager, IOptions<ApplicationSettings> appSettings)
        {
            _roleManager = roleManager;
            _userManager = userManager;
            _appSettings = appSettings.Value;
        }
```

Signin

```
        [AllowAnonymous]
        [HttpPost("Signin")]
        public async Task<IActionResult> Signin([FromBody] ApiSigninModel signInModel)
        {
            var user = await _userManager.FindByNameAsync(signInModel.Email);
            if (user != null && await _userManager.CheckPasswordAsync(user,
signInModel.Password))
            {
                var tokenDescriptor = new SecurityTokenDescriptor
                {
                    Subject = new ClaimsIdentity(new Claim[]
                    {
                        new Claim("UserId", user.Id.ToString()),
                        new Claim("Role",
_userManager.GetRolesAsync(user).Result.FirstOrDefault())
                    }),
                    Expires = DateTime.UtcNow.AddHours(6),
                    SigningCredentials = new SigningCredentials(new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_appSettings.ApplicationSecret)),
SecurityAlgorithms.HmacSha256Signature)
                };
                var tokenHandler = new JwtSecurityTokenHandler();
                var securityToken = tokenHandler.CreateToken(tokenDescriptor);
                var token = tokenHandler.WriteToken(securityToken);

                return Ok(new { BearerToken = token });
            }
            return Unauthorized();
        }
```

Signup

```
        [AllowAnonymous]
        [HttpPost("Signup")]
        public async Task<IActionResult> Signup([FromBody] ApiSigninModel signupModel)
        {
            var user = new User
            {
                UserName = signupModel.Email,
                Email = signupModel.Email,
                Role = await _roleManager.FindByNameAsync(RoleType.User.ToString())
            };
            var result = await _userManager.CreateAsync(user, signupModel.Password);

            if (result.Succeeded)
            {
                await _userManager.AddToRoleAsync(user, RoleType.User.ToString());
            }

            if (!result.Errors.Any())
                return await Signin(signupModel);
            return Unauthorized();
        }
```

Step 3 – Configuring the Authentication

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

```
//Add ApplicationSettings values to Services Collection
services.Configure<ApplicationSettings>(Configuration.GetSection("ApplicationSettings"));
//Encode your key specified in appsettings.json
var key =
Encoding.UTF8.GetBytes(Configuration["ApplicationSettings:ApplicationSecret"].ToString())
;

//Add Identity to the Services Collection
services.AddIdentity<User, Role>(options =>
{
    options.User.RequireUniqueEmail = true;
})
.AddEntityFrameworkStores<ApplicationDbContext>()
.AddUserStore<UserStore<User, Role, ApplicationDbContext, Guid>>()
.AddRoleStore<RoleStore<Role, ApplicationDbContext, Guid>>()
.AddDefaultTokenProviders();
```

```
//Add Authentication to the Services Collection
services.AddAuthentication(x =>
{
    x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
    x.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(x => {
    x.RequireHttpsMetadata = false;
    x.SaveToken = false;
    x.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(key),
        ValidateIssuer = false,
        ValidateAudience = false,
        ClockSkew = TimeSpan.Zero
    };
});
```

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.

```
[Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
```

Add the following above the class definition of the RoutineController.

```
[Route("api/[controller]")]
[Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
```

Add the following above the class definition of the ExerciseController.

```
[Route("api/[controller]")]
[Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
```

Finally, add the following, above the class definition of the UserController.

```
[Route("api/[controller]")]
[Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
```

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

```
namespace RoutineCatalogue.Models.Types
{
    public enum HyperMediaType
    {
        GET,
        POST,
        PUT,
        DELETE
    }
}
```

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

```
namespace RoutineCatalogue.Models.ApiModels
{
    public class HyperMedia<T>
    {
        public string Rel { get; set; }
        public string Href { get; set; }
        public string Action { get; set; }
    }
}
```

Step 2 – Creating the Methods

Now create a new class named HyperMediaResponse.cs

```
using RoutineCatalogue.Models.Types;
using System;
using System.Collections.Generic;
namespace RoutineCatalogue.Models.ApiModels
{
    public class HyperMediaResponse<T>
    {
        public List<HyperMedia<T>> HyperMedia;
        public HyperMediaResponse(Guid? id = null)
        {
            HyperMedia = new List<HyperMedia<T>>();
            if (typeof(T).Name != "Set") HyperMedia.Add(createHyperMedia("GET ALL",
"GET"));
            foreach (var action in Enum.GetNames(typeof(HyperMediaType)))
            {
                if (id == null && action != "Post") continue;
                HyperMedia.Add(createHyperMedia(action, action, id));
            }
        }
        private HyperMedia<T> createHyperMedia(string rel, string action, Guid? id =
null)
        {
            var media = new HyperMedia<T>
            {
                Rel = rel + " " + typeof(T).Name,
                Href = "/api/" + typeof(T).Name + "/" + (id == Guid.Empty ? "" :
id.ToString()),
                Action = action.ToString(),
            };
            return media;
        }
    }
}
```

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.

```
return Ok(new { message = _repo.Get(id), hypermedia = new HyperMediaResponse<Set>(id) });
return Ok(new { message = _repo.Add(body), hypermedia = new
HyperMediaResponse<Set>(body.Id) });
return Ok(new { message = "Set Updated!", hypermedia = new
HyperMediaResponse<Set>(body.Id) });
return Ok(new { message = "Set Deleted!", hypermedia = new HyperMediaResponse<Set>() });
```

Repeat this process for ExerciseController with a modification.

```
return Ok(new { message = await _repo.GetAll(), hypermedia = new
HyperMediaResponse<Exercise>(Guid.Empty) });
return Ok(new { message = _repo.Get(id), hypermedia = new
HyperMediaResponse<Exercise>(id) });
return Ok(new { message = _repo.Add(body), hypermedia = new
HyperMediaResponse<Exercise>(body.Id) });
return Ok(new { message = "Exercise Updated!", hypermedia = new
HyperMediaResponse<Exercise>(body.Id) });
return Ok(new { message = "Exercise Deleted!", hypermedia = new
HyperMediaResponse<Exercise>() });
```

Now, lets repeat this process for the RoutineController.

```
return Ok(new { message = await _repo.GetAll(), hypermedia = new
HyperMediaResponse<Routine>(Guid.Empty) });
return Ok(new { message = _repo.Get(id), hypermedia = new HyperMediaResponse<Routine>(id)
});
return Ok(new { message = _repo.Add(body), hypermedia = new
HyperMediaResponse<Routine>(body.Id) });
return Ok(new { message = "Routine Updated!", hypermedia = new
HyperMediaResponse<Routine>(body.Id) });
return Ok(new { message = "Routine Deleted!", hypermedia = new
HyperMediaResponse<Routine>() });
```

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.

```
"ApplicationSettings": {
  "RoutineServiceIP": "https://localhost:     "
},
```

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

```
services.AddSingleton(RestClient.For<IRoutineService>(Configuration["ApplicationSettings:
ApplicationSecret"].ToString()));
```

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

```
using System;
using System.Collections.Generic;
namespace WorkoutService.Models
{
    public class Routine : BaseResponse
    {
        public string Name { get; set; }
        public string CreatedBy { get; set; }
        public string Description { get; set; }
        public IEnumerable<Set> Sets { get; set; }
    }
    public class Set : BaseResponse
    {
        public string Exercise { get; set; }
        public string Directions { get; set; }
        public Guid ExerciseId { get; set; }
        public int? Repetitions { get; set; }
        public double? Weight { get; set; }
        public int Order { get; set; }
    }
    public class BaseResponse
    {
        public Guid Id { get; set; }
    }
}
```

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

```
using RestEase;
using System.Collections.Generic;
using System.Threading.Tasks;
using WorkoutService.Models;
namespace WorkoutService.Services
{
    public interface IRoutineService
    {
        [Get("api/Routine/GetRoutinesWithSets")]
        Task<IEnumerable<Routine>> GetRoutines();
    }
}
```

Dependency Injection

Configure RestEase in the WorkoutService Startup File.

```
services.AddSingleton(RestClient.For<IRoutineService>(Configuration["ApplicationSettings:
RoutineServiceIP"].ToString()));
```

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.cs

```
using Microsoft.EntityFrameworkCore;
using RoutineCatalogue.Models.ApiModels;
using RoutineCatalogue.Models.Entities;
using RoutineCatalogue.MVC.Data;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
namespace RoutineCatalogue.API.Services
{
    public class RoutineService
    {
        ApplicationDbContext _context;
        public RoutineService(ApplicationDbContext context)
        {
            _context = context;
        }
        public async Task<IEnumerable<object>> GetAll()
        {
            return await _context.Set<Routine>().Include(x => x.Sets).ThenInclude(x =>
x.Exercise).Select(r => new
            {
                Id = r.Id,
                Name = r.Name,
                Description = r.Description,
                CreatedBy = r.CreateBy.Email,
                Sets = r.Sets.Select(s => new SetBase
                {
                    Id = s.Id,
                    ExerciseId = s.Exercise.Id,
                    Exercise = s.Exercise.Name,
                    Directions = s.Exercise.Description,
                    Repetitions = s.Repetitions,
                    Weight = s.Weight,
                    Order = s.Order
                })
            }).ToListAsync();
        }
    }
}
```

Now add this service to the Startup Services Collection.

```
services.AddScoped<RoutineService>();
```

Finally, add RoutineService to the RoutineController Constructor Injection.

```
IRepository<Routine, RoutineViewModel, RoutineIndexViewModel> _repo;
RoutineService _routineService;
public RoutineController(IRepository<Routine, RoutineViewModel, RoutineIndexViewModel>
repo, RoutineService routineService)
{
    _repo = repo;
    _routineService = routineService;
}
```

And add a special method for returning this type of object to the RoutineController.

```
[HttpGet("GetRoutinesWithSets")]
[AllowAnonymous]
public async Task<IEnumerable<object>> GetRoutinesWithSets()
{
    return await _routineService.GetAll();
}
```

Step 4 – Cache Factory

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

```
using Microsoft.Extensions.Caching.Memory;
namespace WorkoutService.Services
{
    public class RoutineFactory
    {
        IRoutineService _routineService;
        IMemoryCache _cache;
        public RoutineFactory(IRoutineService routineService, IMemoryCache cache)
        {
            _routineService = routineService;
            _cache = cache;
            var routines = _routineService.GetRoutines().Result;
            _cache.Set("Routines", routines);
            foreach (var routine in routines)
                _cache.Set($"{routine.Id}", routine);
        }
    }
}
```

Dependency Injection

Configure the .NET Core Cache and the RoutineFactory in the WorkoutService Startup Service Collection.

```
services.AddMemoryCache();
services.AddSingleton<RoutineFactory>();
```

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.

```
namespace WorkoutService.Models
{
    public class ApplicationSettings
    {
        public string RoutineServiceIP { get; set; }
        public string ApplicationSecret { get; set; }
    }
}
```

Add a new Application Setting to your ApplicationSettings JSON in appsettings.json.

```
  "ApplicationSettings": {
    "RoutineServiceIP": "https://localhost:44394",
    "ApplicationSecret": "CodePaLOUsa2019!"
  },
```

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.

```
using System;
namespace WorkoutService.Models
{
    public class Workout
    {
        public int Order { get; set; }
        public double Weight { get; set; }
        public Guid RoutineId { get; set; }
        public Guid SetId { get; set; }
        public int Repetitions { get; set; }
    }
}
```

Next, let's create the Hypermedia Classes. Start with the Hypermedia.cs in the Models Folder.C

```
namespace WorkoutService.Models
{
    public class Hypermedia
    {
        public string Rel { get; set; }
        public string Href { get; set; }
        public string Action { get; set; }
    }
}
```

Create the Service

Now create HypermediaService.cs in the Services Folder.

```
using System;
using WorkoutService.Models;
namespace WorkoutService.Services
{
    public class HypermediaService
    {
        public Hypermedia GetHypermediaFromRoutineId(Guid id)
        {
            return new Hypermedia
            {
                Rel = "FirstSet",
                Href = $"/api/Set?RoutineId={id}",
                Action = "GET"
            };
        }
        public Hypermedia GetHypermediaForNextSet()
        {
            return new Hypermedia
            {
                Rel = "NextSet",
                Href = "/api/Set",
                Action = "POST",
            };
        }
    }
}
```

Create the Hypermedia

Add HypermediaService to the Services Collection.

```
services.AddSingleton<HypermediaService>();
```

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

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Caching.Memory;
using System;
using System.Collections.Generic;
using WorkoutService.Models;
using WorkoutService.Services;
namespace WorkoutService.Controllers
{
    [Route("api/[controller]")]
    [Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
    public class RoutineController : ControllerBase
    {
        IMemoryCache _cache;
        HypermediaService _hypermediaService;
        public RoutineController(IMemoryCache cache, HypermediaService hypermediaService)
        {
            _cache = cache;
            _hypermediaService = hypermediaService;
        }
        [HttpGet]
        public IActionResult GetRoutines()
        {
            var routines = new List<Routine>();

            return Ok(_cache.TryGetValue("Routines", out routines));
        }
        [HttpGet("{id}")]
        public IActionResult GetRoutine(Guid id)
        {
            var routine = new Routine();
            _cache.TryGetValue(id, out routine);
            return Ok(new { routine, hypermedia =
_hypermediaService.GetHypermediaFromRoutineId(routine.Id) });
        }
    }
}
```

Create the Controllers

Finally, lets create our SetController.cs

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Caching.Memory;
using System;
using System.Linq;
using WorkoutService.Models;
using WorkoutService.Services;
namespace WorkoutService.Controllers
{
    [Route("api/[controller]")]
    [Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
    public class SetController : ControllerBase
    {
        IMemoryCache _cache;
        HypermediaService _hypermediaService;
        public SetController(IMemoryCache cache)
        {
            _cache = cache;
        }
        // GET api/values
        [HttpGet("{routineId}")]
        public IActionResult StartRoutine(Guid routineId)
        {
            if (routineId == null || routineId == Guid.Empty) return BadRequest();
            var routine = new Routine();
            _cache.TryGetValue(routineId, out routine);
            var set = routine.Sets.First();
            return Ok(new { set, hypermedia =
_hypermediaService.GetHypermediaForNextSet() });
        }

        [HttpPost]
        public IActionResult PostWorkout(Workout workout)
        {
            if (workout.Order < 0) return BadRequest();
            //Update DynamoDB  with values
            var routine = new Routine();
            _cache.TryGetValue(workout.RoutineId, out routine);
            var set = routine.Sets.Skip(workout.Order + 1).FirstOrDefault();
            return set == null ? Ok(new { message = "Congrats! You completed a workout."
}) : Ok(new { set, hypermedia = _hypermediaService.GetHypermediaForNextSet() });
        }
    }
}
```

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 WorkoutRoutine

```
using System;
using System.Collections.Generic;
namespace WorkoutService.Models
{
    public class WorkoutRoutine
    {
        public WorkoutRoutine()
        {
            DateStarted = DateTime.Now;
        }
        public ICollection<Workout> Workouts { get; set; }
        public DateTime DateStarted { get; set; }
        public DateTime? DateCompleted { get; set; }
    }
}
```

Add another new class named WorkoutHistory

```
using System.Collections.Generic;
namespace WorkoutService.Models
{
    public class WorkoutHistory
    {
        public WorkoutHistory() {}
        public WorkoutHistory(string key)
        {
            Key = key;
        }
        public string Key { get; set; }
        public ICollection<WorkoutRoutine> WorkoutRoutines { get; set; }
    }
}
```

Step 2 – Add the Services

Create a new class in the Services Folder named IWorkoutRepository

```
using System;
using System.Threading.Tasks;
using WorkoutService.Models;
namespace WorkoutService.Services
{
    public interface IWorkoutRepository
    {
        Task<WorkoutHistory> Read(Guid routineId);
        Task Write(Workout workout);
        void CompleteWorkout(Guid id);
    }
}
```

Create another new class in the Services Folder named WorkoutRepository

```
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Microsoft.AspNetCore.Http;
using System;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;
using WorkoutService.Models;
namespace WorkoutService.Services
{
    public class WorkoutRepository : DynamoDBContext, IWorkoutRepository
    {
        private DynamoDBOperationConfig _config;
        private IHttpContextAccessor _httpContextAccessor;
        public WorkoutRepository(IAmazonDynamoDB db, string tableName,
IHttpContextAccessor httpContextAccessor) : base(db)
        {
            _config = new DynamoDBOperationConfig()
            {
                OverrideTableName = tableName
            };
            _httpContextAccessor = httpContextAccessor;
        }
        public async Task<WorkoutHistory> Read(Guid routineId)
        {
            return await LoadAsync<WorkoutHistory>(getKeyFromRoutine(routineId),
_config);
        }
        public async Task Write(Workout workout)
        {
            var history = Read(workout.RoutineId).Result;
            if (history == null) history = new
WorkoutHistory(getKeyFromRoutine(workout.RoutineId));

            var routine = getMostRecentNotCompletedWorkout(history);
            if (routine == null) history.WorkoutRoutines.Add(new WorkoutRoutine());

            routine.Workouts.Add(workout);
            await SaveAsync(history, _config);
        }
        public void CompleteWorkout(Guid id)
        {
            var history = Read(id).Result;
            var routine = getMostRecentNotCompletedWorkout(history);
            routine.DateCompleted = DateTime.Now;
            SaveAsync(history, _config);
        }
        private WorkoutRoutine getMostRecentNotCompletedWorkout(WorkoutHistory wh)
        {
            return wh.WorkoutRoutines.OrderByDescending(x =>
x.DateStarted).FirstOrDefault(x => x.DateCompleted == null);
        }
```

```
        private string getKeyFromRoutine(Guid routineId)
        {
            var userId = (_httpContextAccessor.HttpContext.User.Identity as
ClaimsIdentity).FindFirst(ClaimTypes.NameIdentifier);
            return $"{userId}-{routineId}";
        }
    }
}
```

Step 3 – Configure the DI

Add the following Services to the Startup Service Collection

```
services.AddSingleton<IAmazonDynamoDB, AmazonDynamoDBClient>();
services.AddSingleton<IWorkoutRepository, WorkoutRepository>();
services.TryAddSingleton<IHttpContextAccessor, HttpContextAccessor>();
```

Step 4 – Update the SetController to Reference the IWorkoutService instance

Constructor Injection

```
        IMemoryCache _cache;
        HypermediaService _hypermediaService;
        IWorkoutRepository _workoutRepository;
        public SetController(IMemoryCache cache, HypermediaService hypermediaService,
IWorkoutRepository workoutRepository)
        {
            _cache = cache;
            _hypermediaService = hypermediaService;
            _workoutRepository = workoutRepository;
        }
```

HttpPost

```
        [HttpPost]
        public async Task<IActionResult> PostWorkout(Workout workout)
        {
            if (workout.Order < 0) return BadRequest();
            await _workoutRepository.Write(workout);
            var routine = new Routine();
            _cache.TryGetValue(workout.RoutineId, out routine);
            var set = routine.Sets.Skip(workout.Order + 1).FirstOrDefault();
            if (set == null)
            {
                _workoutRepository.CompleteWorkout(workout.RoutineId);
                return Ok(new { message = "Congrats! You completed a workout." });
            }
            return Ok(new { set, hypermedia =
_hypermediaService.GetHypermediaForNextSet() });
        }
```

Step 5 – Update the ApplicationSettings.cs

```
public class ApplicationSettings
{
    public string RoutineServiceIP { get; set; }
    public string ApplicationSecret { get; set; }
    public string DynamoDb { get; set; }
}
```

Step 6 – Update the appsettings.json

```
"ApplicationSettings": {
  "RoutineServiceIP": "https://localhost:44394",
  "ApplicationSecret": "CodePaLOUsa2019!",
  "DynamoDb": "WorkoutRepository"
},
```

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

```
IMemoryCache _cache;
HypermediaService _hypermediaService;
RoutineFactory _routineFactory;
public RoutineController(IMemoryCache cache, HypermediaService hypermediaService,
RoutineFactory routineFactory)
{
    _cache = cache;
    _hypermediaService = hypermediaService;
    _routineFactory = routineFactory;
}
```

HttpPost

```
[HttpPost]
public void HydrateCache()
{
    _routineFactory.HyrdrateCache();
}
```

Step 2 - Create Factory method to rehydrate cache

```
IRoutineService _routineService;
IMemoryCache _cache;
public RoutineFactory(IRoutineService routineService, IMemoryCache cache)
{
    _routineService = routineService;
    _cache = cache;
    HyrdrateCache();
}
public void HyrdrateCache()
{
    var routines = _routineService.GetRoutines().Result;
    _cache.Set("Routines", routines);
    foreach (var routine in routines)
        _cache.Set($"{routine.Id}", routine);
}
```

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

```
using System.Threading.Tasks;
namespace RoutineCatalogue.MVC.Services
{
    public interface IPublisherAdapter
    {
        Task<bool> Publish();
    }
}
```

Create another new class in this folder named PublisherAdapter.cs

```
using System.Threading.Tasks;
namespace RoutineCatalogue.MVC.Services
{
    public class Publisher : IPublisherAdapter
    {
        IPublisher _publisher;
        public Publisher(IPublisher publisher)
        {
            _publisher = publisher;
        }
        public Task<bool> Publish()
        {
            return _publisher.Publish();
        }
    }
}
```

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

```
using RestEase;
using System.Threading.Tasks;
namespace RoutineCatalogue.MVC.Services
{
    public interface IPublisher
    {
        [Post("api/Routine/")]
        Task<bool> Publish();
    }
}
```

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

```
using RoutineCatalogue.Models.Entities;
using RoutineCatalogue.Models.ViewModels;
using RoutineCatalogue.MVC.Services;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
namespace RoutineCatalogue.MVC.Repositories
{
    public class PublishRepository<Model, ViewModel, IndexVieModel> : Repository<Model,
ViewModel, IndexVieModel> where Model : BaseEntity where ViewModel : BaseViewModel
    {
        Repository<Model, ViewModel, IndexVieModel> _inner;
        IPublisher _publisher;
        public PublishRepository(Repository<Model, ViewModel, IndexVieModel> inner,
IPublisher publisher)
        {
            _inner = inner;
            _publisher = publisher;
        }
        public Guid Add(Model entity)
        {
            _publisher.Publish();
            return _inner.Add(entity);
        }
        public void Delete(Guid id)
        {
            _publisher.Publish();
            _inner.Delete(id);
        }
        public bool EntityExists(Guid id)
        {
            return _inner.EntityExists(id);
        }
        public ViewModel Get(Guid id)
        {
            return _inner.Get(id);
        }
        public Task<List<IndexVieModel>> GetAll()
        {
            return _inner.GetAll();
        }
        public void Update(ViewModel viewModel)
        {
            _publisher.Publish();
            _inner.Update(viewModel);
        }
    }
}
```

Step 5 – Implement the DI

Add the IP for the WorkoutService to your appsettings.json in the RoutineCatalogue.API Project

```
    "ApplicationSecret": "CodePaLOUsa2019!",
    "WorkoutServiceIP": "https://localhost:44362"
```

Add the WorkoutServiceIP Property to the ApplicationSettings Class in the RoutineCatalogue.Models Project

```
    public class ApplicationSettings
    {
        public string AdminUsername { get; set; }
        public string AdminPassword { get; set; }
        public string ApplicationSecret { get; set; }
        public string WorkoutServiceIP { get; set; }
    }
```

Make a few changes to the Services Collection in the RoutineCatalogue.API Project

```
var config = Configuration.GetSection("ApplicationSettings").Get<ApplicationSettings>();

//Add ApplicationSettings values to Services Collection

services.Configure<ApplicationSettings>(Configuration.GetSection("ApplicationSettings"));
//Encode your key specified in appsettings.json
var key = Encoding.UTF8.GetBytes(config.ApplicationSecret);
```

and

```
services.AddScoped(typeof(IRepository<,,>), typeof(PublishRepository<,,>));
services.AddScoped(typeof(Repository<,,>));
services.TryAddSingleton<IHttpContextAccessor, HttpContextAccessor>();
services.AddScoped<IPublisherAdapter, Publisher>();
services.AddSingleton(RestClient.For<IPublisher>(config.WorkoutServiceIP));
```

Step 6 – Lock down with CORS policies

Dependency Injection

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

```
services.AddCors(options =>
{
    options.AddPolicy("RoutineService",
        builder =>
        {
            builder.WithOrigins(config.RoutineServiceIP)
                                .AllowAnyHeader()
                                .AllowAnyMethod();
        });
});
```

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

```
services.AddCors(options =>
{
    options.AddPolicy("WorkoutService",
        builder =>
        {
            builder.WithOrigins(config.WorkoutServiceIP)
                                .AllowAnyHeader()
                                .AllowAnyMethod();
        });
});
```

Data Annotation

Add this Data Annotation to the RoutineCatalogue.API Projects RoutineControllers GetRoutinesWithSets Method

```
[HttpGet("GetRoutinesWithSets")]
[EnableCors("WorkoutService")]
[AllowAnonymous]
public async Task<IEnumerable<object>> GetRoutinesWithSets()
{
    return await _routineService.GetAll();
}
```

And add this DataAnnotation to your WorkoutServices RoutineController Post

```
[HttpPost]
[EnableCors("RoutineService")]
public void HydrateCache()
{
    _routineFactory.HyrdrateCache();
}
```