DevCruise 2019

ASP.NET Core 3.0 for Dummies

Code and slides

Clone the following repo from GitHub:

https://github.com/nickverschueren/devcruise-2019

Step 1

.NET Core “Hello World!”

Step 1: Start with a simple console application

- Create an empty folder on disk called “DevCruise”
- Start VisualStudio Code in the folder
- (Optional) Initialize a GIT repository
  - Don’t forget to add an appropriate `.gitignore` file
- Open a new terminal window
- Confirm the .NET Core version

```
› dotnet --version          ⇨ 3.0.100
```

- Generate the .NET Core Console application

```
› dotnet new console
```

Step 1: Start with a simple console application

```
using System;

namespace DevCruise
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Step 1: A very clean and simple “csproj” build script

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Step 2

Running the .NET Core “Hello World!” app

Step 2: Run it!

- Start debugging
  - Debug – Start Debugging (F5)
  - Select .NET Core
  - launch.json is generated automatically
  - Hit F5 again

  - In Debug Console you should see:

```
› Hello World!
```

Step 2: What else happened?

- VS Code generated (.vscode folder):
  - launch.json
  - tasks.json
- MSBuild generated:
  - “assets”, Debug and Nuget stuff file in “obj” folder
  - “bin” folder with out build output

Step 3

.NET Core Webserver “Hello World!”

Step 3: Make the console app into a webserver!

- Change DevCruise.csproj file

```
Sdk="Microsoft.NET.Sdk" ⇨ Sdk="Microsoft.NET.Sdk.Web"
```

- Run `dotnet restore` in the terminal window

- Add usings to Program.cs:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Hosting;
```

Step 3: Make the console app into a webserver!

Change Main()

```
static void Main(string[] args)
{
    var hostBuilder = Host.CreateDefaultBuilder(args);
    hostBuilder.ConfigureWebHostDefaults(webBuilder => {
            webBuilder.Configure(app => {
                app.Run(context =>
                    context.Response.WriteAsync("Hello World!"));
            });
        });
    var host = hostBuilder.Build();
    host.Run();
}
```

Step 3: Make the console app into a webserver!

- Start Debug
  - Delete `launch.json` and `tasks.json` from `.vscode` folder
  - Hit F5, select .NET Core
  - A new `launch.json` and `tasks.json` are generated
- Look for the hosting location in the Terminal window
- A browser should open at given URL e.g. http://localhost:5000
- You should see:

```
Hello World!
```

Step 4

Cleanup Program.cs

Step 4: Create Startup class

- Add a new file named `Startup.cs`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace DevCruise
{
    class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
        }

        public void Configure(IApplicationBuilder app)
        {
            app.Run(context => context.Response.WriteAsync("Hello World!"));
        }
    }
}
```

Step 4: Refactor Program.cs

- Change Main()

```
static void Main(string[] args)
{
    var hostBuilder = Host.CreateDefaultBuilder(args);
    hostBuilder.ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
    var host = hostBuilder.Build();
    host.Run();
}
```

- Start Debug (F5)

Step 5

Serving Static Files

Step 5: Serve static files

- Change Program.cs
  - Add the following where they should go:
    - `using System.IO;`
    - `var webRoot = Path.Combine(Environment.CurrentDirectory, "wwwroot");`
    - `webBuilder.UseWebRoot(webRoot);`
- Change Startup.cs
  - Comment out `app.Run(…)`
  - Add `app.UseStaticFiles();`
- Create a new sub-folder called `wwwroot`
  - Add an `index.html` file of you choice
- Hit F5!

Step 5: Serve static files

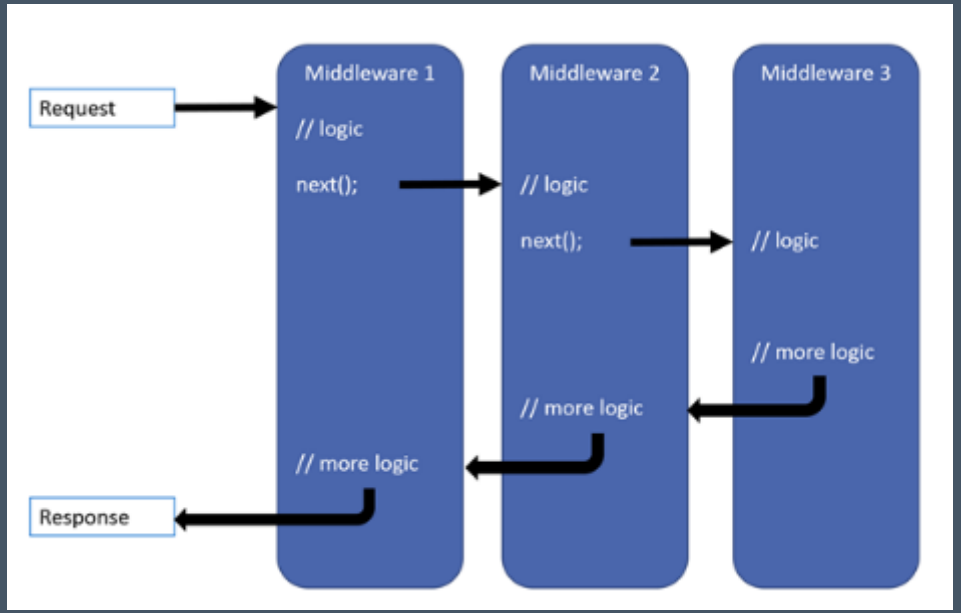
- Got a 404? What went wrong?
  - ASP.NET Core will honor default files out-of-the-box!
- Try explicitly requesting `/index.html`
- To resolve the problem add `app.UseDefaultFiles();`
  - Observe: the order in the Configure method is important!
  - Try reordering UseStaticFiles and UseDefaultFiles

The ASP.NET Core Pipeline

A little bit of theory…

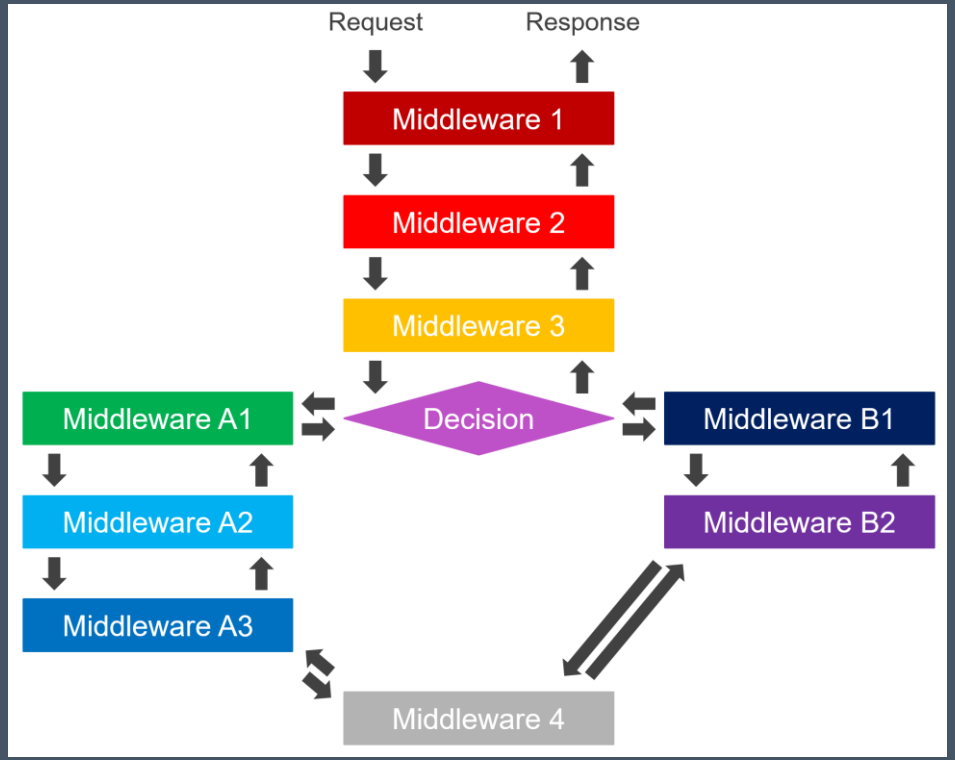
ASP.NET Core Pipeline

- The pipeline consists of chained components called “Middlewares”
- Each middleware gets a delegate to the next middleware in the chain
- The request context gets passed down the chain until handled
- The chain then rolls back up to the first middleware



ASP.NET Core Pipeline

- Pipelines do not have to be straight all the time, they can
  - Branch
  - Merge
  - Terminate
- Example: the “`Map`” and “`MapWhen`” middlewares for handling some requests differently
  - “/admin” requires authentication
  - “/” does not require authentication



ASP.NET Core Pipeline

Typical usage of middleware:

- Add to dependency injection
  - `services.Add`*`Middleware`*`(`*`configurationDelegate`*`);`

```
services.AddResponseCompression(o =>
    o.Providers.Add<GzipCompressionProvider>());
```

- Add to the pipeline in the right order
  - `app.Use`*`Middleware`*`();`

```
app.UseResponseCompression();
```

- Question: Does this middleware come first or last in the pipeline?

ASP.NET Core Dependency Injection

- Dependencies are configured in the `ConfigureServices` method
  - Services can be registered:
    - using interfaces, concrete classes and factory methods
    - with lifetimes: Transient (default), Scoped (per request, kind of) and Singleton
  - Services can be resolved:
    - Constructor injection
    - Parameter injection (in certain cases!)
    - Using `IServiceProvider` injected (not recommended)
    - No property injection
  - The default implementation can be easily replaced by Autofac, Ninject, …

Step 6

Building an API

Adding API functionality to the pipeline

- Add the required dependency: this will automatically resolve all controllers later on

```
services.AddControllers();
```

- Add routing to enable resolving actions based on routes

```
app.UseRouting();
```

- Add endpoints for our actions

```
app.UseEndpoints(builder => builder.MapControllers());
```

- Remember to add output compression

Anatomy of an API Controller

- `ApiController` attribute
- `Route` attribute
- `ControllerBase` inheritance (Optional)
- `Http`*`Verb`* attributes
- `public` methods that return `IActionResult`
- Result helper methods: `Ok`, `CreatedAt`, `BadRequest`, `Conflict`, …

```
using Microsoft.AspNetCore.Mvc;

namespace DevCruise
{
    [ApiController]
    [Route("/api/room")]
    public class RoomController : ControllerBase
    {
        [HttpGet]
        public IActionResult GetRooms()
        {
            return Ok(new [] {"Fes", "Rabat",
                "Nador"} );
        }
    }
}
```

Adding the API Controller

- Add new folder named “Controllers”
- Add new file named “RoomController.cs”
- Copy code from previous slide to RoomController file.
- Start debugging and browse to /api/room

Step 7

Adding a data store

Describing a data store

- Add a persistence provider
  - Open a terminal window
  - Type: `dotnet add package Microsoft.EntityFrameworkCore.SqLite`
- In the `Model` folder add a new file called `DevCruiseDbContext.cs`

```
using Microsoft.EntityFrameworkCore;

namespace DevCruise.Model
{
    public class DevCruiseDbContext : DbContext
    {
        public DevCruiseDbContext(DbContextOptions<DevCruiseDbContext> options) : base(options)
        {
            if (Database.EnsureCreated()) Initialize();
        }
    }
}
```

Describing a data store

- Add the DevCruiseDbContext to dependency injection

```
services.AddDbContext<DevCruiseDbContext>(o =>
    o.UseSqlite("Data Source=App_Data/DevCruiseDb.sqlite;"));
```

  - The string is the relative location for the database file.

- The DbContext is now ready to hold data and to be injected where we need it

Creating a model

Refactor the RoomController

- Add a new folder called `Model`
- Add a new file called `Room.cs`

```
namespace DevCruise.Model
{
    public enum Room
    {
        Fes = 1,
        Rabat = 2,
        Nador = 3
    }
}
```

- Change the GetAll method to return `Enum.GetNames(typeof(Room))`
- Remember to add `using`

Creating a model

- Add 2 more files to the Model folder: Speaker.cs and Session.cs

```
using System.ComponentModel.DataAnnotations;

namespace DevCruise.Model
{
    public class Speaker
    {
        [Key]
        public int Id { get; set; }
        [Required, MaxLength(250)]
        public string Email { get; set; }
        [Required, MaxLength(50)]
        public string FirstName { get; set; }
        [Required, MaxLength(100)]
        public string LastName { get; set; }
        [MaxLength(2000)]
        public string Bio { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace DevCruise.Model
{
    public class Session
    {
        [Key]
        public int Id { get; set; }
        [Required, MaxLength(10)]
        public string Code { get; set; }
        [Required, MaxLength(100)]
        public string Title { get; set; }
        [Required, MaxLength(2000)]
        public string Description { get; set; }
    }
}
```

Adding the entities to the DbContext

- Add 2 properties to the DbContext

```
public DbSet<Session> Sessions { get; set; }
public DbSet<Speaker> Speakers { get; set; }
```

  - These properties will be generated into tables by the framework

- Add an override to the DbContext

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
   var session = modelBuilder.Entity<Session>();
   session.HasIndex(s => s.Code).IsUnique();
   var speaker = modelBuilder.Entity<Speaker>();
   speaker.HasIndex(s => s.Email).IsUnique();
}
```

  - This will add unique indexes on the fields Code and Email, our business keys

Initializing the data

- Change the constructor of DevCruiseDbContext and make it a partial class

```
public partial class DevCruiseDbContext : DbContext
{
    public DevCruiseDbContext(
         DbContextOptions<DevCruiseDbContext> options) : base(options)
    {
        EnsureAppDataDirectory();
        if (Database.EnsureCreated()) Initialize();
    }
...
```

- Copy over the DevCruiseDbContext.Initializer.cs file from the Model in the “After” project

Show me the Data!

- Add a SpeakerController
- Debug the result!
- Now create the SessionController on your own

```
using DevCruise.Model;
using Microsoft.AspNetCore.Mvc;
using System.Linq;

namespace DevCruise.Controllers
{
    [ApiController]
    [Route("/api/speaker")]
    public class SpeakerController : ControllerBase
    {
        private DevCruiseDbContext _dbContext;
        public SpeakerController(DevCruiseDbContext dbContext)
        {
            _dbContext = dbContext;
        }

        [HttpGet]
        public IActionResult GetSpeakers()
        {
            return Ok(_dbContext.Speakers.ToList());
        }
    }
}
```

Adding methods with parameters

- Add a new GET method to the SessionController for a single session

```
[HttpGet("{code}")]
public IActionResult GetSession(string code)
{
    var session = _dbContext.Sessions.SingleOrDefault(
        s => s.Code == code);
    if(session == null)
        return NotFound();
    return Ok(session);
}
```

Step 8

Documenting the API with Swagger

Adding the GET methods for single entities

- Add SwashBuckle using Nuget
  - In the terminal window run

```
dotnet add package Swashbuckle.AspNetCore --version 5.0.0-rc2
```

- Add the generation of the Swagger document and Swagger UI

```
services.AddSwaggerGen(c => c.SwaggerDoc("v1.0", new OpenApiInfo
    { Title = "DevCruise API Documentation", Version = "1.0" }));
```

```
app.UseSwagger();
app.UseSwaggerUI(c => c.SwaggerEndpoint(
    "/swagger/v1.0/swagger.json", "DevCruise API v1.0"));
```

› These middlewares need to execute BEFORE the Routing middleware

› The “v1.0” string in the generation translates to “/v1.0/” in the UI middleware

Try out SwaggerUI

- Debug the app
- Browse to /swagger
- Also try /swagger/v1.0/swagger.json

- If you want you can change launch.json to always show SwaggerUI for now on

```
"serverReadyAction": {
    "action": "openExternally",
    "pattern": "^\\s*Now listening on:\\s+https?://\\S+",
    "uriFormat": "http://localhost:5000/swagger"
},
```

Our work is not finished!

- We need to annotate the possible responses our methods can produce, e.g.:

```
[ProducesResponseType(typeof(string[]),
    StatusCodes.Status200OK)]
```

  - The typeof() function allows us to pass a class type as a Type parameter
  - StatusCodes is a class in the namespace Microsoft.AspNetCore.Http
  - If our method can return more than one possible response, we add one attribute per type, e.g. Status200OK and Status404NotFound
- All methods should be annotated correctly to produce an accurate Swagger document
- Note that all models used are now also shown in the Schemas section of Swagger UI

Other annotations

- A lot of other annotations can be done to enhance the experience of the end-user of the documentation
  - Tags can be added to sort calls (by controller is the default)
  - Security requirements, e.g. scopes
  - Descriptions on parameters
  - Descriptions on the model
  - Include existing XML documentation in the Swagger document
  - …
- The necessary attributes live in the Swashbuckle.AspNetCore.Annotations Nuget-package

Step 9

Async all the way!

Async – Await Pattern

- Async methods should ALWAYS return a `Task` or `Task<T>`

```
public async Task<IActionResult> GetSpeakers()
{
    var speakers = await _dbContext.Speakers.ToListAsync();
    return Ok(speakers);
}
```

- The `async` keyword tells the runtime to initialize an async state machine
- The `await` keyword tells the runtime to wait for a result and unwrap the Task
- More than one `Task` can be awaited in a single method, also in parallel using `Task.WhenAll()`
- The classes `Task` and `Task<T>` live in the `System.Threading.Tasks` namespace

Async – Await Pattern

- Common practice is to suffix `async` methods with “Async”
- The actual value returned inside an `async` method IS NOT a `Task`
  - The return value gets wrapped in a `Task` by the runtime
- The `await` keyword can be used in-line in a statement
  - Braces can be used to indicate what is actually awaited
- Lambda’s and anonymous methods can also be `async`/`await`
- Interfaces do not use the `async` keyword, but do use a `Task` as return value
- To implement a function that returns a Task without `async` and `await` (e.g. when mocking), use `Task.FromResult()` or `Task.CompletedTask`
  - This is expensive, don’t overuse

https://github.com/davidfowl/AspNetCoreDiagnosticScenarios/blob/master/AsyncGuidance.md

EntityFramework Core and Async

- EntityFramework Core provides Async methods for almost all operations
- The necessary extension methods live in the `Microsoft.EntityFrameworkCore` namespace
- Examples:
  - `ToListAsync()`
  - `ToArrayAsync()`
  - `SingleOrDefaultAsync()`
  - `ToDictionaryAsync()`
  - `CountAsync()`
  - `SumAsync()`

Exercise

- Let’s make this entire API async!

Step 10

Using ViewModels

Introducing ViewModels

- Entities are annotated for persistence
  - More of different validations may be required for the API
- Simpler models may be required for lists
- Complex structures can be reduced down in the API
- We may want to protect some fields from changing e.g. the database ID

Introducing ViewModels

- Create a new folder called ViewModels underneath the Controllers folder
- Create a class called Session in the ViewModels
  - Other attributed to try:

    EmailAddress, StringLength, Compare,

    CreditCard, MinLength, Range, Phone, …

```
namespace DevCruise.Controllers.ViewModels
{
    public class Session
    {
        public string Code { get; set; }
        public string Title { get; set; }
    }
}
```

- Create another class called SessionDetail

```
namespace DevCruise.Controllers.ViewModels
{
    public class SessionDetail
    {
        [Required, MaxLength(10)]   public string Code { get; set; }
        [Required, MaxLength(100)]  public string Title { get; set; }
        [Required, MaxLength(2000)] public string Description { get; set; }
    }
}
```

Mapping ViewModels to Models

- In the Terminal window run:
  - `› dotnet add package Automapper`
- Add a folder underneath Controllers called MappingProfiles
- Add a new class called SessionMapping

```
using AutoMapper;
namespace DevCruise.Controllers.MappingProfiles
{
    public class SessionMapping : Profile
    {
        public SessionMapping()
        {
            CreateMap<Model.Session, ViewModels.Session>();
            CreateMap<Model.Session, ViewModels.SessionDetail>();
        }
    }
}
```

Mapping ViewModels to Models

- Add Automapper to dependency injection

```
services.AddSingleton<IConfigurationProvider>(s =>
    new MapperConfiguration(c => c.AddMaps(typeof(Startup))));
services.AddScoped(s => s.GetService<IConfigurationProvider>().CreateMapper(s.GetService));
```

- Add an `IMapper` parameter to the controller and a read-only member field to keep it in
- Use the `Map<T>()` function of the `IMapper` to convert the entities to viewmodels

```
return Ok(_mapper.Map<ViewModels.SessionDetail[]>(sessions));

return Ok(_mapper.Map<ViewModels.SessionDetail>(session));
```

- Remember to also change the `ProducesResponseType` attributes to the viewmodels!
- Try it out!

Can we improve on the mapping?

- We are mapping the results of the database query, so we are querying fields we don't use

```
SELECT "s"."Id", "s"."Code", "s"."Description", "s"."Title"
FROM "Sessions" AS "s"
```

- We can prevent this by using the `ProjectTo<T>()` extension method

```
var sessions = await _dbContext.Sessions
    .ProjectTo<ViewModels.Session>(_mapper.ConfigurationProvider)
    .ToListAsync();
return Ok(sessions);
```

```
SELECT "dtoSession"."Code", "dtoSession"."Title"
FROM "Sessions" AS "dtoSession"
```

Step 11

Completing the CRUD methods

DbContext as a Unit-of-Work

- The `DbContext` tracks changes to entities for as long as it exists
  - All entities queried through the `DbContext` will be tracked automatically
  - Entities created outside of the `DbContext` can be added to it for tracking
- The `DbContext` can persist all changes in a single transaction using `SaveChangesAsync()`
- The default behavior of the dependency injection system will give you a single `DbContext` instance per request
- The `DbContext` will thus automatically function as a Unit-of-Work, ideal for simple use cases

Adding a POST method

- In the SessionController add the following

```
[HttpPost]
[ProducesResponseType(typeof(ViewModels.SessionDetail[]), StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status409Conflict)]
public async Task<IActionResult> CreateSession([FromBody] ViewModels.SessionDetail newSession)
{
    var duplicateCodeExists = await _dbContext.Sessions.AnyAsync(s => s.Code == newSession.Code);
    if (duplicateCodeExists)
        return this.Problem(StatusCodes.Status409Conflict, nameof(newSession.Code),
            $"Session with code {newSession.Code} already exists");

    var session = _mapper.Map<Session>(newSession);
    await _dbContext.AddAsync(session);
    await _dbContext.SaveChangesAsync();

    return CreatedAtAction(nameof(GetSession), new { code = session.Code },
        _mapper.Map<ViewModels.SessionDetail>(session));
}
```

Oops! What is this.Problem()?

- We will add a small extension method to make returning problems easier
  - The source code can be found in the “After” folder under Controllers/Extensions/ProblemDetailsExtensions.cs
- To use it we add a using statement for the `DevCruise.Controllers.Extensions` namespace
- This extension method will format our problems to conform to IETF RFC 7807
  - https://tools.ietf.org/html/rfc7807

Adding a PUT method

- In the SessionController add the following

```
[HttpPut("{code}")]
[ProducesResponseType(typeof(ViewModels.SessionDetail), StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public async Task<IActionResult> UpdateSession(string code,
    [FromBody] ViewModels.SessionDetail updatedSession)
{
    var session = await _dbContext.Sessions.SingleOrDefaultAsync(s => s.Code == code);
    if (session == null)
        return this.Problem(StatusCodes.Status404NotFound, nameof(code),
           $"Session with code {code} not found");

    _mapper.Map(updatedSession, session);

    await _dbContext.SaveChangesAsync();
    return Ok(_mapper.Map<ViewModels.SessionDetail>(session));
}
```

Wait a minute, we still need to prevent duplicate codes!

- The unique index on the database will take care of that, but to present a nicer error to the caller we can add the following before the first `_mapper.Map()`

```
if (!string.Equals(updatedSession.Code, code))
{
    var duplicateCodeExists = await _dbContext.Sessions.AnyAsync(s =>
        s.Code == updatedSession.Code);
    if (duplicateCodeExists)
        return this.Problem(StatusCodes.Status409Conflict, nameof(code),
            $"Session with code {code} already exists");
}
```

- Remember we now also have to indicate that a 409 response is possible

```
[ProducesResponseType(StatusCodes.Status409Conflict)]
```

Finally add the DELETE method

- In the SessionController add the following

```
[HttpDelete("{code}")]
[ProducesResponseType(typeof(ViewModels.SessionDetail), StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[ProducesResponseType(StatusCodes.Status409Conflict)]
public async Task<IActionResult> DeleteSession(string code)
{
    var session = await _dbContext.Sessions.SingleOrDefaultAsync(s => s.Code == code);
    if (session == null)
        return this.Problem(StatusCodes.Status404NotFound, nameof(code),
            $"Session with code {code} not found");

    _dbContext.Sessions.Remove(session);
    await _dbContext.SaveChangesAsync();
    return Ok(_mapper.Map<ViewModels.SessionDetail>(session));
}
```

Is that it?

- Not quite, if we run now and POST a new session we will get:
  `AutoMapper.AutoMapperMappingException: Missing type map configuration or unsupported mapping`
- We still have to add the reverse mapping to our mapping profile

```
CreateMap<ViewModels.SessionDetail, Model.Session>();
```

- But basically that's it!
- Now try doing the same for the Speaker entity yourself

Step 12

Authentication and Authorization

Adding JWT Bearer Token Authentication

- Add the Microsoft.AspNetCore.Authentication.JwtBearer Nuget package
- Add the authentication to dependency injection (Azure AD in this case)

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwtBearer(o =>
{
    o.Authority =
        "https://login.microsoftonline.com/0b53d2c1-bc55-4ab3-a161-927d289257f2/v2.0";
    o.Audience = "4be39034-c55a-4ab3-bd3a-38fa0664bb53";
});
```

- Add the authentication middleware to the pipeline

```
app.UseAuthentication();
```

› The order is very important here: add after Routing and before Endpoints

But wait, everything still works and I didn’t authenticate!

- True, we have stated that we would accept authentication by AAD if offered, but we have not set up security barriers to our API yet.

- For this we use the second part: Authorization
  - With Authorization we will declare policies that will block unauthorized users

Add Authorization policies

- First we add a new folder called Security and a class called Scopes

```
namespace DevCruise.Security
{
    public static class Scopes
    {
        public const string ReadAccess = "readAccess";
        public const string WriteAccess = "writeAccess";

        public const string AadScopePrefix = "api://devCruiseApi/";
        public const string AadReadAccess = AadScopePrefix + ReadAccess;
        public const string AadWriteAccess = AadScopePrefix + WriteAccess;

        public const string ScopeClaimType = "http://schemas.microsoft.com/identity/claims/scope";

        public static bool HasScope(this ClaimsPrincipal user, string scopeName)
            => user.FindFirst(Scopes.ScopeClaimType)?.Value.Split(' ')
                .Contains(scopeName) ?? false;
    }
}
```

Add Authorization policies

- Add the policies to dependency injection

```
services.AddAuthorization(o =>
{
     o.AddPolicy(Scopes.ReadAccess, builder => builder.RequireAssertion(context =>
         context.User.HasScope(Scopes.ReadAccess)));
     o.AddPolicy(Scopes.WriteAccess, builder => builder.RequireAssertion(context =>
         context.User.HasScope(Scopes.WriteAccess)));
});
```

- Add the authorization middleware

```
app.UseAuthorization();
```

- Now we apply the policy where we want using the `Authorize` attribute
  - The attribute can be applied to the controller and overridden on any method

```
[Authorize(Scopes.ReadAccess)]
```

Authentication now works!

- Great, any method we try now returns a 401!
  - … but now our Swagger UI is useless ☹
- We still have to
  - tell Swagger about our authentication requirements
  - and tell Swagger UI where to get it’s tokens
  - … easy ☺

Adding scope requirements to Swagger

- For this we use an `IOperationFilter`
  - Copy over the `SecurityRequirementsOperationFilter` class from the “After” folder
- We then add the new filter to the SwaggerGen registration from earlier

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1.0", new OpenApiInfo { Title =
        "DevCruise API Documentation", Version = "1.0" });
    c.OperationFilter<SecurityRequirementsOperationFilter>();
});
```

Adding scope requirements to Swagger

- Last we add the full declaration of our security to the Swagger document so clients know what the need to do to authenticate

```
c.AddSecurityDefinition("oauth2", new OpenApiSecurityScheme
{
    Type = SecuritySchemeType.OAuth2,
    Flows = new OpenApiOAuthFlows
    {
        Implicit = new OpenApiOAuthFlow
        {
            AuthorizationUrl = new Uri(
  "https://login.microsoftonline.com/0b53d2c1-bc55-4ab3-a161-927d289257f2/oauth2/v2.0/authorize"),
            Scopes = new Dictionary<string, string>
            {
                { Scopes.AadReadAccess, "Access read operations" },
                { Scopes.AadWriteAccess, "Access write operations" }
            }
        }
    }
});
```

Just one last piece missing

- The Swagger document is now complete
  - There is a securitySchemes element at the bottom
  - Every method has a security element
  - Every method declares 401 and 403 results
- Now Swagger UI needs to know it's ClientID for Azure AD to authenticate

```
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1.0/swagger.json", "DevCruise API v1.0");
    c.OAuthClientId("4be39034-c55a-4ab3-bd3a-38fa0664bb53");
});
```

Want to know what all of this stuff means?

See you this afternoon at 18:55!


"That's all Folks!"

Resources

Docs

- https://docs.microsoft.com/en-us/aspnet/core/getting-started/?view=aspnetcore-3.0
- https://docs.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-3.0
- https://docs.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-3.0
- https://docs.microsoft.com/en-us/ef/core/

Learn

- https://docs.microsoft.com/en-us/learn/modules/build-web-api-net-core/?view=aspnetcore-3.0

Now have fun exploring ASP.NET Core for yourself!