Szerver oldali .Net programozás

PEKÁRDY MILÁN – PEKARDY@DCS.UNI-PANNON.HU
HORVÁTH ÁDÁM – HORVATH@DCS.UNI-PANNON.HU

Óra ütemezés

- 2019.09.13. 13:50-17:00 I.fsz.I2 – Bevezetés, ASP.NET Core/EF core alapok
- 2019.09.20. 13:50-17:00 I.fsz.I2 – EF Core/Repository
- 2019.10.04. 13:50-17:00 I.fsz.I2 – UnitOfWork, szolgáltatási réteg
- 2019.10.18. 13:50-17:00 I.fsz.I2 – Autentikáció/autorizáció
- 2019.11.08. 13:50-17:00 I.fsz.I2 – ASP.NET Core kiegészítő témák
- 2019.11.15. 13:50-17:00 I.fsz.I2 – Konzultáció/gyakorlati ZH

Követelmények

- Aláírás feltétel: 1 db gyakorlati ZH megírása
- Jegy kialakítása: elégséges gyakorlati ZH min. 50%-tól

Technológiák, eszközök

- .NET Core 2.2 SDK (https://dotnet.microsoft.com/download)
- Visual Studio Code (https://code.visualstudio.com/download)
  - C# for Visual Studio Code extension
- Git for Windows (https://git-scm.com/download/win)
- SmartGit (https://www.syntevo.com/smartgit/download/)
- MS SQL Server Developer Edition (https://www.microsoft.com/en-us/sql-server/sql-server-downloads)
- MS SQL Server Management Studio (https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-2017)

Irodalom, források

- C#:
  - https://docs.microsoft.com/en-us/dotnet/csharp/
  - Joseph Albahari and Ben Albahari: C# 7.0 in a Nutshell
- ASP.NET Core:
  - https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.2
  - Andrew Lock: ASP.NET Core in Action
- EF Core:
  - https://docs.microsoft.com/en-us/ef/core/
  - Jon Smith: Entity Framework Core in Action
- Git:
  - https://desoft.hu/oktatas/git/tartalom
- Github repository:
  - https://github.com/pekmil/szerveroldalidotnet

ASP.NET Core alapok

https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.2

ASP.NET Core

- ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-based, Internet-connected applications. With ASP.NET Core, you can:
  - Build web apps and services, IoT apps, and mobile backends.
  - Use your favorite development tools on Windows, macOS, and Linux.
  - Deploy to the cloud or on-premises.
  - Run on .NET Core or .NET Framework

ASP.NET Core vs. ASP.NET

ASP.NET Core	ASP.NET 4.x
Build for Windows, macOS, or Linux	Build for Windows
Razor Pages is the recommended approach to create a Web UI as of ASP.NET Core 2.x. See also MVC, Web API, and SignalR.	Use Web Forms, SignalR, MVC, Web API, WebHooks, or Web Pages
Multiple versions per machine	One version per machine
Develop with Visual Studio, Visual Studio for Mac, or Visual Studio Code using C# or F#	Develop with Visual Studio using C#, VB, or F#
Higher performance than ASP.NET 4.x	Good performance
Use .NET Core runtime	Use .NET Framework runtime

.NET Core vs. .NET Framework

.NET Core	.NET Framework
You have cross-platform needs	Your app currently uses .NET Framework (recommendation is to extend instead of migrating)
You are targeting microservices	Your app uses third-party .NET libraries or NuGet packages not available for .NET Core
You are using Docker containers	Your app uses .NET technologies that aren't available for .NET Core
You need high-performance and scalable systems	Your app uses a platform that doesn’t support .NET Core
You need side-by-side .NET versions per application	

Startup class

- The Startup class configures services and the app's request pipeline.
- ASP.NET Core apps use a Startup class, which is named Startup by convention. The Startup class:
  - Optionally includes a ConfigureServices method to configure the app's services. A service is a reusable component that provides app functionality. Services are configured—also described as registered—in ConfigureServices and consumed across the app via dependency injection (DI) or ApplicationServices.
  - Includes a Configure method to create the app's request processing pipeline.
- ConfigureServices and Configure are called by the ASP.NET Core runtime when the app starts:

Startup class

```
public class Startup
{
    // Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        ...
    }

    // Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app)
    {
        ...
    }
}
```

Startup class

- The Startup class is specified when the app's host is built. The Startup class is typically specified by calling the WebHostBuilderExtensions.UseStartup<TStartup> method on the host builder:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

- The host provides services that are available to the Startup class constructor. The app adds additional services via ConfigureServices. Both the host and app services are then available in Configure and throughout the app.

ConfigureServices method

- Optional.
- Called by the host before the Configure method to configure the app's services.
- Where configuration options are set by convention.
- Adding services to the service container makes them available within the app and in the Configure method.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>()
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();


    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
}
```

Configure method

- The Configure method is used to specify how the app responds to HTTP requests.

- The request pipeline is configured by adding middleware components to an IApplicationBuilder instance.

- Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline or short-circuiting the chain, if appropriate.

Configure method

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();

    app.UseMvc();
}
```

Dependency injection

- ASP.NET Core supports the dependency injection (DI) software design pattern, which is a technique for achieving Inversion of Control (IoC) between classes and their dependencies.
- A *dependency* is any object that another object requires.
- The use of an interface or base class to abstract the dependency implementation.
- Registration of the dependency in a service container. ASP.NET Core provides a built-in service container, IServiceProvider. Services are registered in the app's Startup.ConfigureServices method.
- Injection of the service into the constructor of the class where it's used. The framework takes on the responsibility of creating an instance of the dependency and disposing of it when it's no longer needed.
- (Hollywood Principle: Don't Call Us; We'll Call You)

DI example: interface and implementation

```
public interface IMyDependency
{
    Task WriteMessage(string message);
}
```

```
public class MyDependency : IMyDependency
{
    private readonly ILogger<MyDependency> _logger;

    public MyDependency(ILogger<MyDependency> logger)
    {
        _logger = logger;
    }

    public Task WriteMessage(string message)
    {
        _logger.LogInformation(
            "MyDependency.WriteMessage called. Message: {MESSAGE}",
            message);

        return Task.FromResult(0);
    }
}
```

DI example: registration

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddScoped<IMyDependency, MyDependency>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));

    // OperationService depends on each of the other Operation types.
    services.AddTransient<OperationService, OperationService>();
}
```

DI example: usage

```
public class IndexModel : PageModel
{
    private readonly IMyDependency _myDependency;

    public IndexModel(
        IMyDependency myDependency,
        OperationService operationService,
        IOperationTransient transientOperation,
        IOperationScoped scopedOperation,
        IOperationSingleton singletonOperation,
        IOperationSingletonInstance singletonInstanceOperation)
    {
        _myDependency = myDependency;
        OperationService = operationService;
        TransientOperation = transientOperation;
        ScopedOperation = scopedOperation;
        SingletonOperation = singletonOperation;
        SingletonInstanceOperation = singletonInstanceOperation;
    }

    public OperationService OperationService { get; }
    public IOperationTransient TransientOperation { get; }
    public IOperationScoped ScopedOperation { get; }
    public IOperationSingleton SingletonOperation { get; }
    public IOperationSingletonInstance SingletonInstanceOperation { get; }

    public async Task OnGetAsync()
    {
        await _myDependency.WriteMessage(
            "IndexModel.OnGetAsync created this message.");
    }
}
```

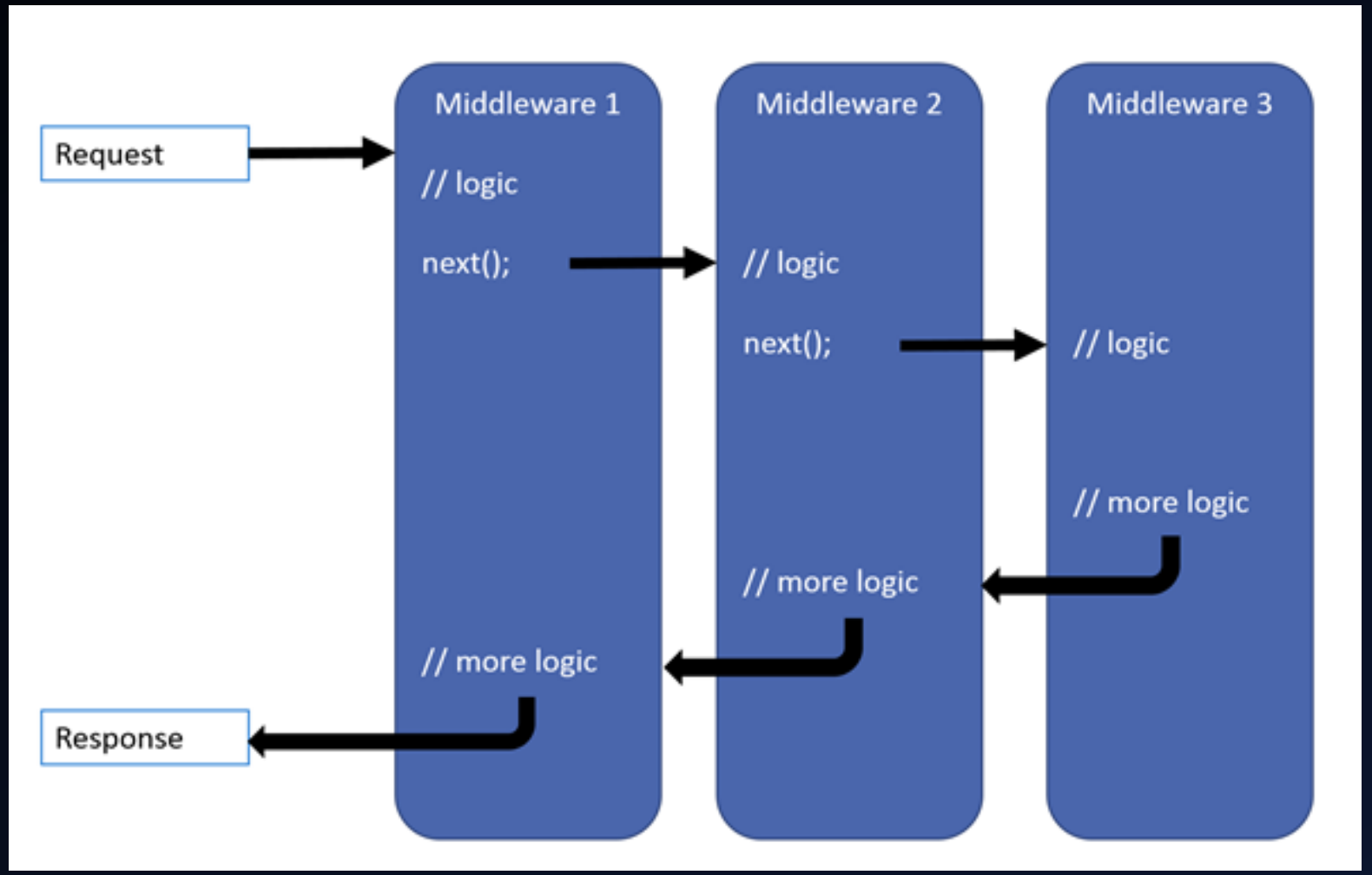
DI: service lifetimes

- **Transient**
  - Transient lifetime services (AddTransient) are created each time they're requested from the service container. This lifetime works best for lightweight, stateless services.
- **Scoped**
  - Scoped lifetime services (AddScoped) are created once per client request (connection).
- **Singleton**
  - Singleton lifetime services (AddSingleton) are created the first time they're requested (or when Startup.ConfigureServices is run and an instance is specified with the service registration). Every subsequent request uses the same instance. If the app requires singleton behavior, allowing the service container to manage the service's lifetime is recommended. Don't implement the singleton design pattern and provide user code to manage the object's lifetime in the class.

Middleware

- Middleware is software that's assembled into an app pipeline to handle requests and responses. Each component:
- Chooses whether to pass the request to the next component in the pipeline.
- Can perform work before and after the next component in the pipeline.
- Request delegates are used to build the request pipeline. The request delegates handle each HTTP request.
- Request delegates are configured using Run, Map, and Use extension methods.

Middleware


Request
Middleware 1
// logic
next();
// more logic
Response
Middleware 2
// logic
next();
// more logic
Middleware 3
// logic
// more logic

Middleware

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            // Do work that doesn't write to the Response.
            await next.Invoke();
            // Do logging or other work that doesn't write to the Response.
        });

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from 2nd delegate.");
        });
    }
}
```

Middleware

- The order that middleware components are added in the Startup.Configure method defines the order in which the middleware components are invoked on requests and the reverse order for the response.
- The order is critical for security, performance, and functionality.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseAuthentication();
    app.UseSession();
    app.UseMvc();
}
```

Middleware

- Configure the HTTP pipeline using Use, Run, and Map
- The Use method can short-circuit the pipeline (that is, if it doesn't call a next request delegate).
- Run is a convention, and some middleware components may expose Run[Middleware] methods that run at the end of the pipeline.
- Map extensions are used as a convention for branching the pipeline. Map branches the request pipeline based on matches of the given request path. If the request path starts with the given path, the branch is executed.

Middleware

```
public class Startup
{
    private static void HandleMapTest1(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 1");
        });
    }

    private static void HandleMapTest2(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 2");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Map("/map1", HandleMapTest1);

        app.Map("/map2", HandleMapTest2);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}
```

Web host

- ASP.NET Core apps configure and launch a *host*.
- The host is responsible for app startup and lifetime management.
- At a minimum, the host configures a server and a request processing pipeline. The host can also set up logging, dependency injection, and configuration.
- The Web Host is for hosting web apps.
- Create a host using an instance of IWebHostBuilder. This is typically performed in the app's entry point, the Main method.

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

Web host

CreateDefaultBuilder performs the following tasks:

- Configures Kestrel server as the web server using the app's hosting configuration providers.
- Sets the content root to the path returned by Directory.GetCurrentDirectory.
- Loads host configuration from:
  - Environment variables prefixed with ASPNETCORE_ (for example, ASPNETCORE_ENVIRONMENT).
  - Command-line arguments.
- Loads app configuration in the following order from:
  - appsettings.json.
  - appsettings.{Environment}.json.
  - Secret Manager when the app runs in the Development environment using the entry assembly.
  - Environment variables.
  - Command-line arguments.
- Configures logging for console and debug output. Logging includes log filtering rules specified in a Logging configuration section of an appsettings.json or appsettings.{Environment}.json file.
- When running behind IIS with the ASP.NET Core Module, CreateDefaultBuilder enables IIS Integration, which configures the app's base address and port. IIS Integration also configures the app to capture startup errors.
- Sets ServiceProviderOptions.ValidateScopes to true if the app's environment is Development.

Web host

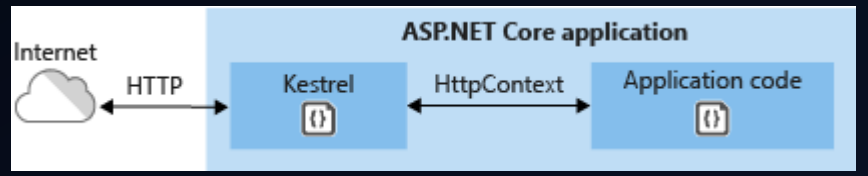
```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args)
    {
        var config = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("hostsettings.json", optional: true)
            .AddCommandLine(args)
            .Build();

        return WebHost.CreateDefaultBuilder(args)
            .UseUrls("http://*:5000")
            .UseConfiguration(config)
            .Configure(app =>
            {
                app.Run(context =>
                    context.Response.WriteAsync("Hello, World!"));
            });
    }
}
```

Servers

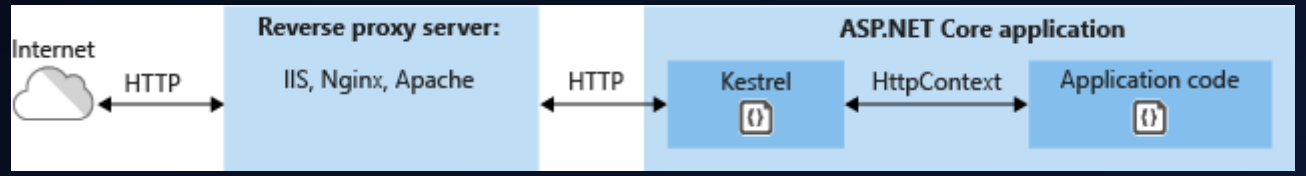
- An ASP.NET Core app runs with an in-process HTTP server implementation.
- The server implementation listens for HTTP requests and surfaces them to the app as a set of request features composed into an HttpContext.
- ASP.NET Core ships with the following:
  - Kestrel server is the default, cross-platform HTTP server implementation.
  - IIS HTTP Server is an in-process server for IIS.
  - HTTP.sys server is a Windows-only HTTP server based on the HTTP.sys kernel driver and HTTP Server API.
- When using IIS or IIS Express, the app either runs:
  - In the same process as the IIS worker process (the in-process hosting model) with the IIS HTTP Server. *In-process* is the recommended configuration.
  - In a process separate from the IIS worker process (the out-of-process hosting model) with the Kestrel server.
- The ASP.NET Core Module is a native IIS module that handles native IIS requests between IIS and the in-process IIS HTTP Server or Kestrel. For more information, see ASP.NET Core Module.

Servers - Kestrel

- Kestrel is the default web server included in ASP.NET Core project templates.
- Use Kestrel:
  - By itself as an edge server processing requests directly from a network, including the Internet.



  - With a *reverse proxy server*, such as Internet Information Services (IIS), Nginx, or Apache. A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel.



Routing

- Routing is responsible for mapping request URIs to endpoint selectors and dispatching incoming requests to endpoints.
- Routes are defined in the app and configured when the app starts.
- A route can optionally extract values from the URL contained in the request, and these values can then be used for request processing.
- Most apps should choose a basic and descriptive routing scheme so that URLs are readable and meaningful. The default conventional route {controller=Home}/{action=Index}/{id?}

Routing – Routing middleware

```
app.UseMvc(routes =>
{
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
```

```
routes.MapRoute(
    name: "us_english_products",
    template: "en-US/Products/{id}",
    defaults: new { controller = "Products", action = "Details" },
    constraints: new { id = new IntRouteConstraint() },
    dataTokens: new { locale = "en-US" });
```

Routing – Conventional routing

```
routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
```

- This style is called conventional routing because it establishes a convention for URL paths:
  - the first path segment maps to the controller name
  - the second maps to the action name.
  - the third segment is used for an optional id used to map to a model entity

Routing – Attribute routing

- Attribute routing uses a set of attributes to map actions directly to route templates.

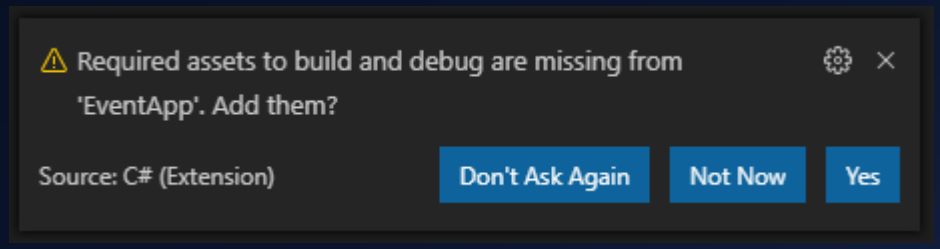
- With attribute routing the controller name and action names play **no** role in which action is selected.

```
public class MyDemoController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    public IActionResult MyIndex()
    {
        return View("Index");
    }
    [Route("Home/About")]
    public IActionResult MyAbout()
    {
        return View("About");
    }
    [Route("Home/Contact")]
    public IActionResult MyContact()
    {
        return View("Contact");
    }
}
```

Example appplication

Visual Studio Code Terminal

- List app templates: `dotnet new -l`
- Create ASP.NET Core Web API application: `dotnet new webapi`
- When you first open the C# project, you'll get a warning that Code needs to build required assets to the project.
- After you click Yes, you'll see a .vscode folder added to your project, which includes the following:
  - launch.json - where Code keeps debugging configuration information
  - tasks.json - where you can define any tasks that you need run - for example, dotnet build or dotnet run
- Run the app: `dotnet run`



Entity Framework Core

BASICS

EF Core

- Entity Framework (EF) Core is a lightweight, extensible, open source and cross-platform version of the popular Entity Framework data access technology.
- EF Core can serve as an object-relational mapper (O/RM), enabling .NET developers to work with a database using .NET objects, and eliminating the need for most of the data-access code they usually need to write.
- EF Core supports many database engines (ex.: Microsoft.EntityFrameworkCore.SqlServer)
- `dotnet add package Microsoft.EntityFrameworkCore.SqlServer`

EF Core: the model

- With EF Core, data access is performed using a model.
- A model is made up of entity classes and a context object that represents a session with the database, allowing you to query and save data.
- You can generate a model from an existing database, hand code a model to match your database, or use EF Migrations to create a database from your model, and then evolve it as your model changes over time.

EF Core: querying

- Instances of your entity classes are retrieved from the database using Language Integrated Query (LINQ).

```
using (var db = new BloggingContext())
{
    var blogs = db.Blogs
        .Where(b => b.Rating > 3)
        .OrderBy(b => b.Url)
        .ToList();
}
```

EF Core: saving data

- Data is created, deleted, and modified in the database using instances of your entity classes.

```
using (var db = new BloggingContext())
{
    var blog = new Blog { Url = "http://sample.com" };
    db.Blogs.Add(blog);
    db.SaveChanges();
}
```

EF Core: the model

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(
            @"Server=(localdb)\mssqllocaldb;Database=Blogging;Integrated Security=True");
    }
}
```

```
{
  "ConnectionStrings": {
    "BloggingDatabase": "Server=(localdb)\\mssqllocaldb;Database=EFGetStarted.ConsoleApp.NewDb;Trusted_Connection=True;"
  },
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<BloggingContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("BloggingDatabase")));
}
```

EF Core: the model

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    public int Rating { get; set; }
    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

Migrations

- A data model changes during development and gets out of sync with the database.
- You can drop the database and let EF create a new one that matches the model, but this procedure results in the loss of data.
- The migrations feature in EF Core provides a way to incrementally update the database schema to keep it in sync with the application's data model while preserving existing data in the database.
- Tools: `dotnet add package Microsoft.EntityFrameworkCore.Tools`

Migrations: create

- After you've defined your model, it's time to create/update the database.
- `dotnet ef migrations add MigrationName`
- Three files are added to your project under the Migrations directory:
  - XXXXXXXXXXXXXX_*MigrationName*.cs--The main migrations file. Contains the operations necessary to apply the migration (in Up()) and to revert it (in Down()).
  - XXXXXXXXXXXXXX_*MigrationName*.Designer.cs--The migrations metadata file. Contains information used by EF.
  - MyContextModelSnapshot.cs--A snapshot of your current model. Used to determine what changed when adding the next migration.
- The timestamp in the filename helps keep them ordered chronologically so you can see the progression of changes.

Migrations

- **Update the database**: apply the migration to the database to create/update the schema
  - `dotnet ef database update`
- **Remove migration**: sometimes you add a migration and realize you need to make additional changes to your EF Core model before applying it, with this command you can remove the last migration
  - `dotnet ef migrations remove`
- **Revert a migration**: if you already applied a migration (or several migrations) to the database but need to revert it, you can use the same command to apply migrations, but specify the name of the migration you want to roll back to
  - `dotnet ef database update LastGoodMigration`

Exercise #1

EVENTAPP

Exercises

- Create an Event model class to represent an event that people can attend with the following properties:
  - Id (int)
  - Name (string)
  - Description (string)
  - Start (DateTime)
  - End (DateTime)
- Create the appropriate DbContext, Service and Controller classes to support basic CRUD functions
- Add and apply a migration to create the database schema

Supporting tools

AUTOMAPPER & SWAGGER

AutoMapper (https://automapper.readthedocs.io/en/latest/Getting-started.html)

- AutoMapper is a convention-based object-to-object mapper that requires little configuration
- Install: `dotnet add package AutoMapper.Extensions.Microsoft.DependencyInjection`
- Configure: `services.AddAutoMapper();`
- Profiles:
  - Profiles in AutoMapper are a way of organizing your mapping collections.
  - To create a Profile, we create a new class and inherit from Profile. We can then add our mapping configuration inside our new classes constructor.
  - When our application starts up and adds AutoMapper, AutoMapper will scan our assembly and look for classes that inherit from Profile, then load their mapping configurations

AutoMapper: Profiles

```
public class EventAppMapperProfile : Profile {
    0 references
    public EventAppMapperProfile(){
        CreateMap<EventCreateDto, Event>();
        CreateMap<EventUpdateDto, Event>();
        CreateMap<Event, EventReadDto>();
    }
}
```

```
public class EventDto {
    0 references
    public string Name { get; set; }

    0 references
    public string Description { get; set; }

    0 references
    public DateTime Start { get; set; }

    0 references
    public DateTime End { get; set; }
}

1 reference
public class EventCreateDto : EventDto { }

1 reference
public class EventUpdateDto : EventDto {
    0 references
    public int Id { get; set; }
}

1 reference
public class EventReadDto : EventDto {
    0 references
    public int Id { get; set; }
}
```

AutoMapper: Usage

- Inject the IMapper interface
- Use the IMapper implementation's Map method

```
private readonly IEventService _eventService;
5 references
private readonly IMapper _mapper;

0 references
public EventsController(IEventService eventService, IMapper mapper){
    _eventService = eventService;
    _mapper = mapper;
}

// GET api/events/getall
[HttpGet]
0 references
public ActionResult<IEnumerable<Event>> GetAll()
{
    var events = _eventService.GetEvents();
    var eventDtos = events.Select(e => _mapper.Map<EventReadDto>(e));
    return Ok(events);
}
```

Swagger (https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-2.2)

- Swagger is a language-agnostic specification for describing REST APIs.
- It allows both computers and humans to understand the capabilities of a service without any direct access to the implementation (source code, network access, documentation).
- One goal is to minimize the amount of work needed to connect disassociated services. Another goal is to reduce the amount of time needed to accurately document a service.

Swagger: Usage

- Install: `dotnet add package Swashbuckle.AspNetCore -v 5.0.0-rc2`
- Add the Swagger generator to the services collection in the Startup.ConfigureServices method:

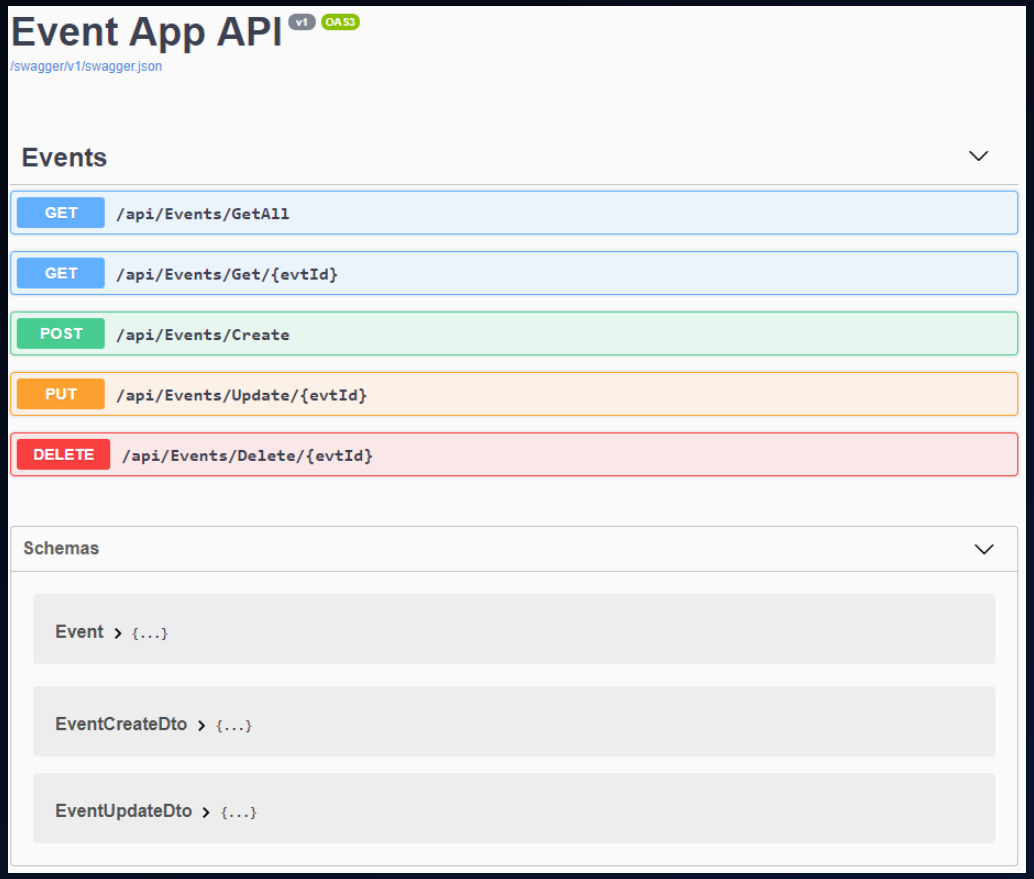
```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "My API", Version = "v1" });
});
```

- Add the Swagger generator to the services collection in the Startup.ConfigureServices method:

```
// Enable middleware to serve generated Swagger as a JSON endpoint.
app.UseSwagger();

// Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
// specifying the Swagger JSON endpoint.
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
});
```

Swagger: Usage

- Browse: https://localhost:5001/swagger/index.html



Exercise #2

EVENTAPP

Exercises

- Create a Place model class to represent a place where an event can take place with the following properties:
  - Id (int)
  - Name (string)
  - Address (string)
- Create/extend the appropriate DbContext, Service and Controller classes to support basic CRUD functions
- Extend the Event model (and other supporting classes) to reflect the one-to-many relationship with the Place model
- Add and apply a migration to modify the database schema

Exercises

- Create a person model class to represent a person who can attend an event with the following properties:
  - Id (int)
  - Name (string)
  - DateOfBirth (DateTime)
- Create/extend the appropriate DbContext, Service and Controller classes to support basic CRUD functions
- Add and apply a migration to modify the database schema

Exercises

- Create an invitation model class to represent an invitation to one of the events with the following properties:
  - PersonId (int) – invitee (Person – navigational property)
  - EventId (int) (Event – navigational property)
  - Status (enum) – Created, Accepted, Declined
- Create/extend the appropriate DbContext, Service and Controller classes to support basic manipulation functions (create, list, accept, decline)
- Add and apply a migration to modify the database schema