Háttéralkalmazások

REST jellegű API-k fejlesztése ASP.NET Core-ban

Simon.Gabor@vik.bme.hu 2022-től
Szabo.Gabor@vik.bme.hu 2021-ig

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Disclosure

Ez az oktatási segédanyag a Budapesti Műszaki és Gazdaságtudományi Egyetem oktatója által kidolgozott szerzői mű.
Kifejezett felhasználási engedély nélküli felhasználása szerzői jogi jogsértésnek minősül.

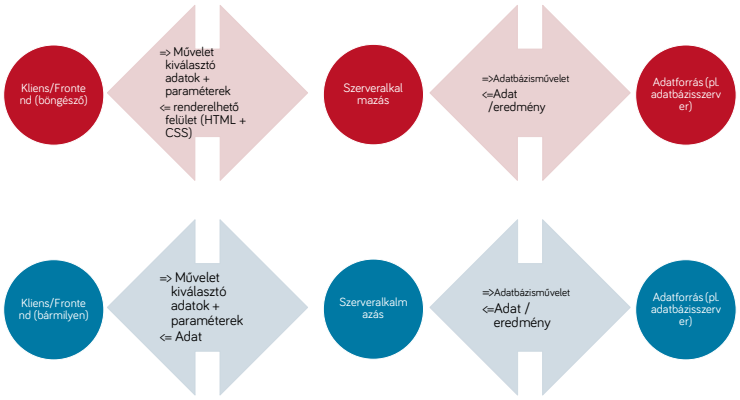
A szerző elérhetősége:
Simon.Gabor@vik.bme.hu

Backend-frontend architektúrák

Tipikus backend-frontend architektúrák

- Backend/Szerveroldal
  - Adatforrás
    - Pl. relációs adatbázis kiszolgáló (RDBMS) (adatbázis)szerveren
    - Adatot szolgáltat az RDBMS képességeinek megfelelően
  - Szerveralkalmazás
    - Valamilyen backend keretrendszerre írt szoftver
    - Általában külön gép(eke)n fut, mint az adatbáziszerver (alkalmazásszerver)
    - Műveleteket definiál, amiket a kliens meghívhat hálózaton keresztül
    - A műveletek végrehajtásához adatbázismműveleteket futtat(hat)
    - A műveletek eredményét visszaadja a kliensnek
- Frontend/Kliensoldal
  - Vagy böngésző
    - HTTP alapú protokollokon kommunikál hálózaton
    - HTML+CSS-t tud renderelni
    - JS és WebAssembly kódot tud végrehajtani, ezek módosíthatják a felületet, hálózaton is kommunikálhatnak
    - Ezek mind hálózatról, a backendről érkeznek, érkezhetnek
  - Vagy nem böngésző
    - Különféle felületi technológiák, leírók
    - Számos különféle nyelv, platform
    - Különféle kommunikációs protokollok
    - iOS app, Android app, Windows Store app, Linux app, stb.
    - A renderelendő felületet a frontend logika állítja össze
      - Ha ehhez kell valamilyen adat, azt a hálózati kommunikációval, a backendről szerzi be
      - Nem kész felületet vár, csak az adatokat hozzá

Szerver vs kliens renderelt



Szerver vs kliens renderelt

- Szerver renderelt
  - A szerveralkalmazás **felületleírót** (HTML+CSS) ad válaszul a kliensnek
  - Csak böngésző frontend
  - A felület szerkezetének előállításhoz szükséges logika a **szerveren** fut
- Kliens renderelt
  - A szerveralkalmazás a felület előállításához szükséges **adatot** adja
  - A felület szerkezetének előállításhoz szükséges logika a **kliensen** fut
  - Gyakorlatilag bármilyen klienstechnológiához illeszkedik
    - Feltétel: adatot tudjon küldeni/fogadni hálózaton keresztül

A renderelés helye szerinti előnyök

Kliens

- Teljes szeparáció a UI és backend csapatok között
  - Más programozói ismereteket igényelnek
- Jellemzően jóval gazdagabb felhasználói élmény
- A kliensalkalmazás használható lehet offline módban (PWA)
- Jellemzően kevesebb adat utazik az első betöltés után, a frissítés gyorsabb
- Szükség szerint egyszerűbben cserélhető a teljes UI
- Kíméli a szerver erőforrásait

Szerver

- Gyors prototipizálás, potenciálisan gyorsabb fejlesztési ciklusok
- Kevesebb kliensoldali ismeret szükséges
- Egyszerűbben generálható kód az üzleti modellek alapján (használható reflexió)
- Kevésbé törékeny a felhasználói felület változására
- Jellemzően kevesebb adat utazik az első betöltéskor, a betöltés gyorsabb
- Nem szükséges komplex kommunikációs réteg karbantartása
- Régi böngészőket is egyszerűen támogat
- Kíméli a kliens erőforrásait

Szerver renderelt

- Ha nincs JavaScript is mellé, akkor elég korlátozott felhasználói élmény
  - Minden felületi változáshoz kommunikáni kellene a szerverrel
  - Ma már a klienseszközök nagyon erősek, nem szükséges minden felületi logikának a szerveren futnia
- Vegytisztán ma már nem alkalmazzuk, több-kevesebb JavaScript logikát is mellékel a szerver, amit a böngésző futtat
  - Pl. jQuery – egyszerűbb felületi módosítások a böngészőn belül is végrehajtódnak
  - Egyes műveletek kliens renderelt módon működhetnek, pl. a JS kód adatot kérdez le a szerverről, ami alapján frissíti a felületet

Kommunikáció

- Kliens-szerveralkalmazás között
  - Szerver renderelt esetben: HTTP alapú protokoll(ok)
  - Kliens renderelt: amit a kliens és a szerver is támogat
    - Többféle kliens is lehet, a közös nevező tipikusan a HTTP
- Szerveralkalmazás – adatbázis között
  - Relációs DB: saját protokoll, a DB driver/provider intézi
  - NoSQL DB: HTTP alapú protokollok

Technológiák .NET-eseknek

- Adatelérési technológia a szerveralkalmazásba
  - .NET: EF, ADO.NET, stb.
- Szerver renderelt esetben
  - Webes keretrendszer a szerveralkalmazásba
    - HTTP kérés + sablon + adatbázis adat => HTTP válasz [HTML+CSS+JS]
    - .NET: **ASP.NET Core (MVC vagy Razor), Blazor Server**
  - Klienstechnológia
    - Egyszerű JavaScript logikák vagy egyszerűbb (lightweight) JS keretrendszer, pl. jQuery
- Kliens renderelt esetben (HTTP alapú kommunikáció)
  - Webes keretrendszer a szerveralkalmazásba
    - Webes API: HTTP kérés + adatbázis adat => HTTP válasz [adat, pl. JSON formátumban]
    - .NET: **ASP.NET Core (Web API)**
  - Klienstechnológia – böngészős kliens
    - Single Page Application (SPA): a böngészőbe csak az alkalmazás használatának elején töltődik le egy HTML oldal, utána szinte mindent az oldalhoz mellékelt JS/WebAssembly kód működtet
    - JS SPA: Angular, React, Vue, stb.
    - WebAssembly SPA: Blazor WebAssembly (C#/.NET kód fut!!!)
  - Klienstechnológia – nem böngészős kliens
    - .NET: **WinForms, WPF, UWP, WinUI, Xamarin/MAUI**
  - Klienstechnológia – tesztelésre
    - Postman

Szerver vs kliens renderelt – mikor melyiket 2023-ban?

- Szerver renderelt
  - Ha nem olyan fontos a tökéletes felhasználói élmény
  - Ha fontosabb a stabilitás, időtállóság
  - Ha elég csak böngészős klienst támogatni
  - Kiforrottabb, időtállóbb technológiák támogatják
  - Ha a célzott böngésző képességei nagyon korlátozottak (pl. régi böngésző, korlátozott JS futtatás)
  - Pl. céges belső, adminisztratív webalkalmazások
- Kliens renderelt
  - Az új fejlesztéseknél általában az elsődleges választás
  - Legmodernebb technológiák támogatják
  - Szinte mindenfajta klienshez (böngésző, mobil, IoT)

ASP.NET Core

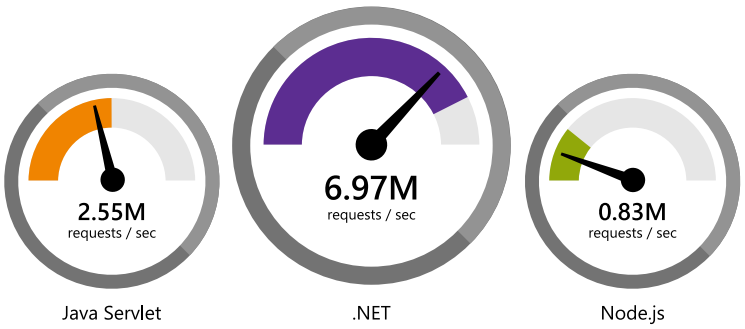
Mi az ASP.NET Core?

- Egy komplex, moduláris webalkalmazás-fejlesztési keretrendszer
  - Open source (https://github.com/dotnet/aspnetcore), 31k csillag
  - Ingyenes
  - Cross-platform (Windows, Linux, Mac)
  - Gyors, jól skálázódik
  - .NET Core/5/6 keretrendszerre épül rá
  - Elsődlegesen szerveralkalmazás írására
  - Webszervert is tartalmaz (Kestrel)
- Statikus tartalmak kiszolgálására
  - Fájlok, képek
  - SPA-k alkatrészei (HTML/JS/CSS/dll)
- Dinamikus kiszolgálásra
  - Szerveroldali rendereléshez
    - ASP.NET Core MVC
    - ASP.NET Core Razor
    - ASP.NET Core Blazor Server
  - Kliensoldali rendereléshez
    - ASP.NET Core Web API (szerver)
    - ASP.NET Core Blazor WebAssembly (kliens - böngésző)
    - ASP.NET Core gRPC (szerver)
  - SignalR (szerver és kliens): valós idejű kétirányú kommunikációra, WebSocket alapú

ASP.NET (+ Core)

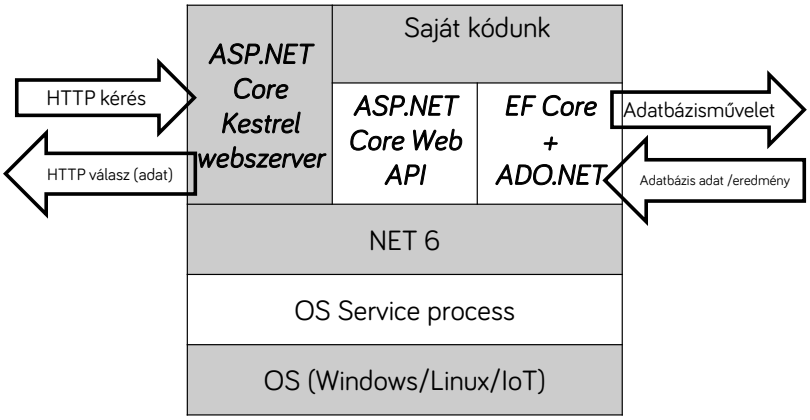
- Az ASP.NET 2002-ben jelent meg, az ASP (Active Server Pages) továbbfejlesztése .NET-re
- Korábban használt (legacy) technológiái:
  - > ASP.NET Web Forms
  - > ASP.NET Web Pages
- Jelenlegi verzió: ASP.NET Core v6
- Most csak ezzel foglalkozunk:
  - > ASP.NET Core v6 Web API
  - > Szerveralkalmazást készítünk, ami adatot ad vissza (kliens renderelt módszer)

Data sourced from official tests available at TechEmpower Round 16.



Java Servlet .NET Node.js

ASP.NET Core szerveralkalmazás tipikus kiépítése



ASP.NET Core szerveralkalmazás tipikus kiépítése

- Fontos, hogy a szerveralkalmazás szolgáltatásként fusson, ne csak simán el legyen indítva
  - Minden OS támogatja, csak máshogy
  - Windows service, Linux daemon
  - Fejlesztői környezetben nem (mindig) foglalkozunk vele
- Kell egy webszerver, ami fogadja a HTTP kéréseket és előállítja a választ
  - Az ASP.NET Core is tartalmaz (Kestrel)
- ASP.NET Core Web API - támogatja a kliens által meghívható műveletek kényelmes megírását
- EF Core – kommunikáció az adatbázissal

ASP.NET Core szerveralkalmazás – alternatív kiépítések

- A legtöbb webszerver eleve szolgáltatásként fut
  - Az alkalmazásunk futhatna ezen process részeként
  - Csak az IIS támogatja
  - Nem kell az alkalmazásunkat felkészíteni szolgáltatásként való futásra
- Két webszerveres kiépítések
  - Ipari webszerverek: nginx, apache, IIS
  - Az ipari webszerver proxy-ként működik
  - Az ipari webszerver fogadja a hálózatról a kéréseket
  - Bonyolultabb, de sokan megbízhatóbbnak tartják
  - Fel kell készíteni az alkalmazásunkat szolgáltatásként való futásra
- Felhős platformszolgáltatásokba telepítés
  - Pl. Azure App Service
  - Minden telepítést, karbantartást elintéz, nekünk csak a
    - kódot vagy a lefordított projektet kell feltölteni
    - és konfigurálni kell (mennyi erőforrás kell, connection string, stb.)
  - Nem kell az alkalmazásunkat felkészíteni szolgáltatásként való futásra

ASP.NET Core Web API ígéret

Alapvetően sima C# programozási konstrukciókban kell megírnunk az üzleti logikát (osztályokat, függvényeket írunk).

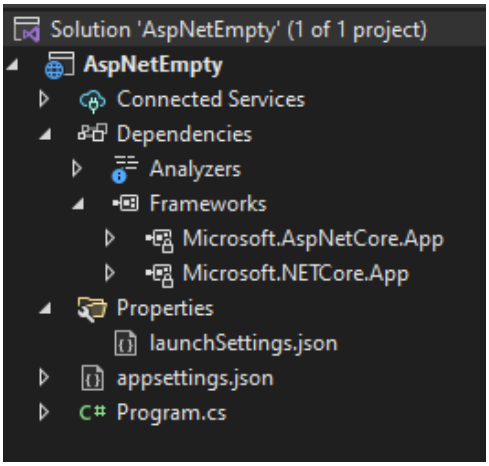
A webes kommunikáció részleteit elfedi előlünk, vagy csak konfigurálnunk kell.

Beizzítás

Projekt létrehozás

- Releváns beépített sablonok
  - ASP.NET Core Empty (.NET 6)
    - Egyetlen művelet
    - Minimális ahhoz, hogy a „/” címre HTTP GET kérést küldve ***Hello World*** szöveget kapjunk
    - Minimal API-t használ
  - ASP.NET Core Web API (.NET 6)
    - Alapvetően kliens renderelt alkalmazásokhoz
    - ***Minimal API*** vagy ***Controller API*** alapú is lehet a kérés kiszolgálás
    - A művelet már bonyolultabb adatot ad vissza
- NuGet csomagot nem kel telepíteni (legfeljebb az EF-höz)
    - Az OS-re telepített .NET 6 SDK/runtime tartalmazza a szükséges ASP.NET Core komponenseket (shared frameworks), a Kestrelt is!

Empty projektsablon



Program.cs – Empty projektsablon

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", () => "Hello World!");
app.Run();
```

Builder

- Az alapszolgáltatásokat és azok konfigurációját fogja össze
  - Új szolgáltatás regisztrálása
    - Builder.Services.Add[Szolgáltatásnév]
      - builder.Services.AddControllers()
      - DI konténerbe regisztrál
  - Szolgáltatás konfigurálása
    - builder.[Szolgáltatás].[Konfiguráló függvény]
      - builder.Logging.AddConsole();
  - Mindig az adott szolgáltatás dokumentációja alapján konfiguráljunk!
- WebApplication.CreateBuilder egy kiinduló buildert ad, amiben már sok minden eleve be van állítva
  - Pl. konfigurációs fájlok kezelése, környezetkezelés, naplózás
  - Csak azt állítjuk át, amit ehhez képest nem úgy akarunk
  - Új szolgáltatásokat, függőségeket adhatunk hozzá a DI konténerhez

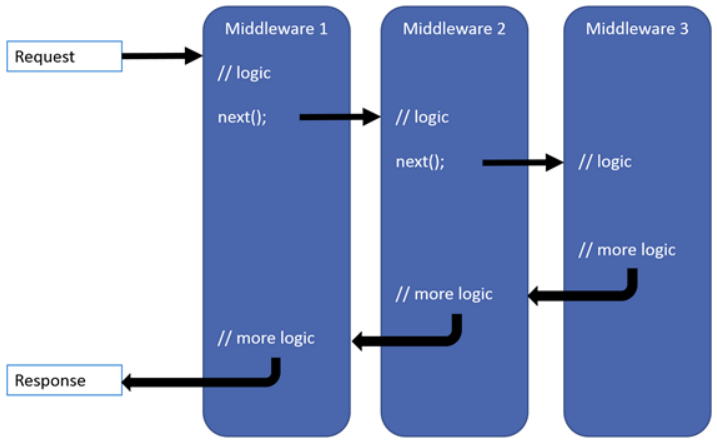
.NET 6 alapszolgáltatások

- Nem csak webes alkalmazásoknál használhatjuk
  - Függőséginjektálás (Dependency Injection)
  - Konfigurációmenedzsment (Configuration Mgmt.)
    Alkalmazásbeállítások (Application settings)
    - User Secrets
  - Naplózás (Logging)
  - Hosztolás (Hosting)
  - Gyorsítótárazás (Caching)
  - Többnyelvűség (Localization)

ASP.NET Core 6 szolgáltatások

- Végpontok, műveletek (endpoints)
  - Kontrollerek
- Végpontkiválasztás (routing)
- Exception handler (Developer Exception Page)
- Webszerver integráció
- Felhasználókezelés (authentication)
- Hozzáférés szabályozás (authorization)
- Metaadat, dokumentáció publikálás (swagger)
- Stb.

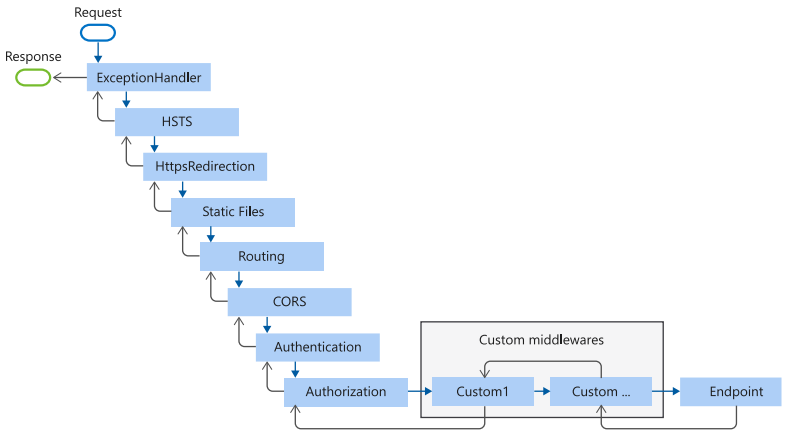
ASP.NET Core alkalmazásszerkezet – A csővezeték



Alkalmazásszerkezet felállítása

- Az alkalmazásszerkezet logikailag egy csővezeték
  - Middleware-ekből (MW) áll, melyek megszabott sorrend szerint láncba vannak fűzve
  - Egy middleware két helyen is beavatkozhat a folyamatba
    - Megkapja a kérést az elölről számított előző MW-től, logikát futtathat, majd
      - Vagy meghívja a sorrendben következő MW-t
      - Vagy maga adja vissza a választ (terminál)
    - Megkapja a választ a hátulról számított előző MW-től, logikát futtathat
      - Pl. megváltoztathatja a választ (titkosít)
- *Builder.Build*: kiinduló alkalmazásszerkezet felállítása
  - Bővít(het)jük a csővezetéket middleware-ekkel
    - Általában beépített MW-ket teszünk bele, de írhatunk saját MW-t is
    - A sorrend fontos!!! Az adott MW dokumentációja alapján járjunk el.
    - `app.Use[MW neve]`
  - A már bennelévő middleware-eket konfigurálhatjuk
    - pl. `app.MapXXX` az endpoint MW-t konfigurálja

ASP.NET Core alkalmazásszerkezet – Példa csővezeték



Kiinduló csővezeték

1. Developer exception page MW: fejlesztői környezetben részletes információt tesz a válaszba a keletkezett, nem kezelt kivételről
2. Routing MW: a bejövő kérés alapján eldönti, hogy a csővezeték további részén definiált végponti logikák közül melyiket kell hívni
3. Endpoint MW: a kiválasztott logika futtatásáért felel

Minimal API kéréskezelés (Request Handling)

- Egyik lehetőség végponti logikák megadására
- Az Endpoint MW-t konfiguráljuk
  - Több MapXXX függvényhívással konfiguráljuk, melyik kérésre mi történjen
  - `app.Map[HTTP ige]([útvonal/route],[delegát/lambda])`
  - Példa:

```
app.MapGet(
      "/users/{userId}/books/{bookId}",
      (int userId, int bookId) =>
      $"The user id is {userId} and book id is {bookId}"
);
```

Kiinduló projekt

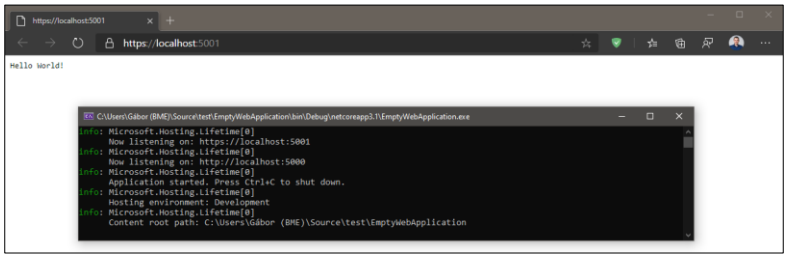
- Konfigurálva van benne minden, amit a kiinudló builder tud
- Fel van állítva egy kiinduló csővezeték (3 MW-rel)
- Egy végponti logika van benne definiálva
  - A weboldal gyökércímére küldött HTTP GET kérésre „Hello World!” szöveg a válasz

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", () => "Hello World!");
app.Run();
```

ASP.NET Core szolgáltatás hozzáadása, konfigurációja

- Mindig az adott szolgáltatás dokumentációja alapján járjunk el! Ez alapján lehet, hogy
  - hozzá kell adnunk egy új szolgáltatás(oka)t a DI konténerhez
    - builder.Services.AddXXX
  - módosítanunk kell valamit a builderben
    - Pl. builder.Logging.ClearProviders()
  - hozzá kell adnunk MW-(ke)t a csővezetékhez
    - app.AddXXX
  - meglévő MW-t kell konfigurálnunk
    - Pl. app.MapXXX
  - a fenti műveletek tetszőleges kombinációját kell alkalmaznunk

Indulás


https://localhost:5001
https://localhost:5001
Hello World!
C:\Users\Gábor (BME)\Source\test\EmptyWebApplication\bin\Debug\netcoreapp3.1\EmptyWebApplication.exe
info: Microsoft.Hosting.Lifetime[0]
Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
Content root path: C:\Users\Gábor (BME)\Source\test\EmptyWebApplication

Controller API

Controller API kéréskezelés (Request Handling)

- Másik lehetőség végponti logikák megadására
- A végponti logikákat, mint függvényeket osztályokba szervezzük, ezek a kontrollerek
  - MVC mintából kapja a nevét
  - Egy kontroller ~ egy üzleti entitáshoz kapcsolódó műveleteket fog össze
  - ControllerBase beépített típusból származtatjuk
  - ApiController attribútumot is teszünk rá
- Paraméterek kitöltésével általában nem kell foglalkoznunk
  - A routing szabályok alapján felöltődik
- Visszatérési értékek sorosításával általában nem kell foglalkoznunk
  - C# objektumokat adhatunk vissza -> HTTP 200-as válasz, a törzsben az objektum sorosítva
  - a böngésző és a szerveralkalmazás letárgyalja a formátumot (XML vs. JSON) (*content negotiation*)
  - kényelmi függvényekkel megadhatjuk a HTTP válasz státuszkódot
- Szerver renderelt és kliens renderelt működésben is használható!
- Itt is az Endpoint MW-t konfiguráljuk, de általában tömegesen (vagy csoportonként) regisztráljuk a kontrollereket
  - vagy app.MapControllers – attribútum alapú routing
  - vagy app.MapControllerRoute – konvenció alapú routing
- A szükséges szolgáltatásokat regisztrálni kell a DI konténerbe
  - `builder.Services.AddControllers();`

ControllerBase, ApiController

- Az ASP.NET Core biztosít ősosztályokat kontrollerek írásához
  - Szerver renderelt: ***Controller***
  - Kliens renderelt: ***ControllerBase***
  - ***Controller*** származik a ***ControllerBase***-ből
- ***ControllerBase*** ősosztály
  - Request, Response, User objektumok
  - Kényelmi függvények a válasz előállításához
    - HTTP 200 OK válaszhotz Ok() függvény
- ***ApiController*** attribútum
  - API-kra jellemző működést automatikusan megvalósít a keretrendszer, pl.
    - Validációs hibák-> HTTP 400-as válasz
    - A kontroller függvényei csak attribútum routing-gal érhetőek el

Végpontkiválasztás kontrollereknél

- HTTP kérés alapján meg kell határozni, hogy ***melyik*** kontroller ***melyik*** függvénye hívódjon meg
- Attribútum alapú routing
  - Főleg kliens renderelt működésre
  - C# attribútumokat teszünk a kontroller
    - osztályra
    - függvényeire
    - függvényparaméreire
- Konvenció alapú routing
  - Főleg szerver renderelt működésre
  - Útvonalsablon(oka)t adunk meg az `app.MapControllerRoute`-ban

```
app.MapControllerRoute(
name: "default",
pattern: "{controller=Home}/{action=Index}/{id?}");
```

- Bővebben: Routing to controller actions in ASP.NET Core | Microsoft Docs

Kontrollertámogatás bekapcsolása

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();
var app = builder.Build();
app.MapControllers();
app.Run();
```

Kontroller – Attribútum alapú routing

```
[Route("api/[controller]")]
[ApiController]
public class DummyController : ControllerBase
{

    // GET: api/Dummy
    [HttpGet]
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }

    // GET: api/Dummy/5
    [HttpGet("{id}")]
    public string Get(int id)
    {
        return "value";
    }
}
```

Kontroller – Attribútum alapú routing

- Route attribútum
  - Többet is rárakhatunk egy osztályra
  - A kontroller osztály milyen cím(ek)re reagál
  - Speciális tokeneket használhatunk benne
    - Pl. [controller] behelyettesítődik a kontroller nevére (a Controller-t levágva a végéről) pl. DummyController => Dummy
- HttpGet, HttpPost, HttpPut, stb.
  - HTTP ige neve
  - Az adott függvény (művelet) milyen HTTP igére reagál
  - Extra URL szegmens(eket)t is definiálhat
    - {szegmensnév}
    - A szegmensnév jelentősége, hogy a név alapján képződnek le függvényparaméter(ek)re
    - Ilyenkor csak akkor reagál az adott művelet, ha a megfelelő számú szegmens van az URL-ben

Web API építés

REST vs RPC

- Hogyan nézzen ki az API? Milyen címek? Milyen paraméterezés? Milyen műveletek?
- RPC stílus
  - Remote Procedure Call
  - A függvényhíváshoz hasonló
  - HTTP példa:
    - GET /api/DogService/GetDogById?id=5,
    - GET /api/DogService/UpdateDog?id=5&name=Bodri
- REST stílus (REpresentational State Transfer)
  - Irányelvek halmaza
    - Nem csak az API kinézetre, de működésre is
    - Nem szabvány!
    - Nem alkalmazza mindenki mindegyik elvet!
  - A generálódó ASP.NET Core kontrollerek többé-kevésbé követik ezeket az irányelveket
  - RESTful Web Services
    - A REST által definiált kritériumoknak eleget tevő webszolgáltatás, ami rendszerek közötti együttműködést tesz lehetővé hálózaton keresztül
  - HTTP példa:
    - GET /api/Dog/5
    - PATCH /api/Dog + a törzsben a módosított Dog

Legfontosabb REST ajánlások

- Kliens-szerver architektúra
- HTTP alapon
  - GET, PUT, PATCH, POST, DELETE stb.
  - Értelemszerűen GET lekérdezésre, DELETE törlésre, stb.
  - Bővebben: https://www.restapitutorial.com/lessons/httpmethods.html
- Erőforrás-orientáltság
- Állapotmentesség
  - A kliensnek nem kell azzal foglalkoznia, hogy a szervernek épp mi az állapota, mi volt az előző hívása, stb.
- Gyorsítótárazhatóság
- Rétegezettség
  - Proxy vagy terheléselosztó komponens könnyen hozzáadható

EF Core adatelérés Web API-ban

1. Szükséges NuGet csomagok telepítése:
   - Microsoft.EntityFrameworkCore.SqlServer
   - Microsoft.EntityFrameworkCore.SqlServer.Design
2. DbContext, entitások, mapping definiálása
3. DbContext regisztrációja a DI konténerbe

```
builder.Services.AddDbContext<DogFarmDbContext>(
        o => o.UseSqlServer("connectionstring")
);
```

4. DbContext használata DI segítségével (kontrollerben vagy külön DAL rétegbeli osztályban)

Migrációk:

1. Migráció létrehozása (Add-Migration)
2. Migráció futtatása az adatbázison (Update-Database)

DogsController.cs: GET

```
[Route("api/[controller]")]
[ApiController]
public class DogsController : ControllerBase
{
    private readonly DogFarmDbContext _context;

    public DogsController(DogFarmDbContext context)
    {
        _context = context;
    }

    // GET: api/Dogs
    [HttpGet]
    public async Task<ActionResult<IEnumerable<Dog>>> GetDogs()
    {
        return await _context.Dogs.ToListAsync();
    }
    // ...
}
```

DogsController.cs: POST

```
// POST: api/Dogs
[HttpPost]
public async Task<ActionResult<Dog>> PostDog([FromBody]Dog dog)
{
    _context.Dogs.Add(dog);
    await _context.SaveChangesAsync();

    return CreatedAtAction("GetDog", new { id = dog.Id }, dog);
}
```

ASP.NET Core API irányelvek

- Ne adjunk vissza entitás típusokat
  - Vagy nem lesz kitöltve a navigációs property, vagy sorosítási problémáink lesznek, pl. körkörös hivatkozások miatt
  - Helyette csináljunk olyan típusokat, amik
    - Csak azokat a propertyket tartalmazzák, amire a kliensnek szüksége van. Ez lehet olyan adat is, ami egy kapcsolódó entitásban van
    - Ezek az ún. DTO-k
    - Lehetőleg lapítottak, azaz nem tartalmaznak navigációs property-t vagy a navigációs property-k típusa is DTO
- Lehetőleg válasszuk le az üzleti logikát a kontroller kódról
  - Az üzleti logikát, adatbázis kommunikációt külön osztály(ok)ba szervezzük
  - Az osztályokat DI segítségével injektáljuk a kontrollerbe (a DbContext helyett)
  - A kontroller áthív az üzleti logikába