

Mi az ASP.NET Core? Mit jelent, hogy egy alkalmazás monolitikus? Mi a „clean architecture”? Sorolj fel min. 3 általános ASP.NET Core által nyújtott infrastruktúra funkciót! Hogyan indul el egy ASP.NET Core projekt? Milyen lépései vannak, hogy egy Web API-n kiadjunk egy EF Core által nyújtott adatmodellt (lekérdezésre, módosításra)?

DIA 8 oldal 13

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
 - > Webszerver 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 ala

Az ASP.NET Core egy complex moduláris webalkalmazás-fejlesztési keretrendszer. Ami open source és ingyenes. Jelenleg elérhető rengeteg platformra mint például a Linux, Mac és Windows. Az ASP.NET Core ezen kívül dinamikus kiszolgálásra is képes attól függően, hogy kliens vagy szerver oldali a renderelés.

DIA 13 oldal 4

A Monolitikus alkalmazás egy olyan alkalmazás, amely telepítési szempontból egyetlen egységként jelenik meg. De ez nem jelenti azt, hogy nem bonthatjuk az alkalmazásunkat fejlesztés során modulokra bontsuk. A modulra bontás nem befolyásolja a build-et, mivel a build után ezek az alkalmazások egyetlen telepítési egység áll elő. A monolitikus kialakítást az alkalmazásban az üzleti logika telepítési egységeinek száma határozza meg.

Monolitikus alkalmazás

- Monolitikus (= egy tömbből álló): olyan alkalmazás, amely *telepítési szempontból* egyetlen egységként jelenik meg
 - > Nem zárja ki, hogy a fejlesztés során modulokra bontsuk, amelyek
 - Külön fejleszthetők, tesztelhetők, buildelhetők
 - Több alkalmazás között újrafelhasználhatók
 - > De ezekből a modulokból a teljes alkalmazás buildje során egyetlen telepítési egység áll elő
 - > A backend üzleti logika telepítési egységeinek száma határozza meg a monolit jellegét → hiába futnak a kliensek és a DB külön gépen, attól az még monolit marad, ha a szerver oldali üzleti logika egyetlen telepítési egység
- Telepítési egység tipikus megjelenése:
 - > ASP.NET web alkalmazás: zip fájl
 - > Java webalkalmazás: .war (=Web Archive) fájl (ez is zip)
 - Spring Boot esetén akár .jar is lehet

A clean architecture egy olyan elv ami a SOLID elvre épül és lényegében arról szól, hogy az architekruális tervezés egyik fő feladata a komponensek közötti határok meghúzása és komponensekre bontása.

A ASP.NET Core az alap szolgáltatásokon kívül nyújt plussz szolgáltatásokat. Ilyenek például a Végpontkiválasztás, Webszerver integráció, Felhasználókezelés

.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)

Először létrehozzuk a projektet ahol használhatunk sablonokat ha szeretnénk de ezeket mi magunk is kidolgozhatjuk. Ezután a projektet beállítjuk és konfiguráljuk a Program vagy Startup.cs -ben. Ezután futtatjuk az alkalmazásunkat esetleg más NuGet csomagokat telepítünk. Ezután teszteljük és végül hibát keressünk az alkalmazásban.

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.

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!

Először létrehozunk egy projektet az előbb felvetett módon. Ezután az adatmodellünket határozzuk meg. Ezzel együtt magát a DbContext-et is konfiguráljuk az adatmodellünk miatt. Ha ezzel megvoltunk akkor Startup.cs -ben beállítjuk a DbContext-et az adatbázis elérés miatt.

Ha ezzel megvoltunk készítünk egy controller-t a amiben meghatározzuk, hogy milyen címen szeretnénk kiszolgálni majd a kéréseket.

A controller-ben ha nem bontjuk jobban szét modulokra az alkalmazásunkat akkor már magából a controllerből kiszolgálhatunk. Itt a meghatározott módon lekérjük az adatot az adatbázisból a context segítségével. Majd ezt szerkeszthetjük törölhetjük vagy vissza adhatjuk. Ha szerkesztünk bármit akkor azt a context-ben visszakell iktatni majd azt mentenünk kell és csak utána less iktatva az adatbázisban.

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)

Legfontosabb REST ajánlások

- Kliens-szerver architektúra
- HTTP alapon
 - > GET, PUT, PATCH, POST, DELETE stb.
 - > Értelmszerűen GET lekérésre, 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
- Rétegzettség
 - > Proxy vagy terheléselosztó komponens könnyen hozzáadható

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();  
    }  
    // ...  
}
```

