

Tartalom

Tartalom.....	1
Dinamikus Razor Pages	1
Alapfogalmak	1
Dependency Injection (ismétlés)	1
Projekt előkészítése	2
Gyakorló feladat.....	3
Interakció az oldallal	5
Önálló feladat.....	6
Bónusz feladatok.....	6

Dinamikus Razor Pages

A mai laboron az előző alkalommal megismert Razor Pages technológia segítségével egy dinamikus weboldalt fogunk létrehozni. A weboldal témáján nem változtattunk, ugyanúgy egy rendelő webshopot tervezünk majd.


Ehhez két új megoldást fogunk használni: a pizzák adatait egy erre a célra létrehozott Service osztály segítségével tároljuk majd el, az oldalaink forráskódját pedig ún. HTTP request handler függvényekkel egészítjük ki.

Alapfogalmak

Dependency Injection (ismétlés)

A [dependency injection](#) (DI) egy általános szoftverfejlesztési módszertan, amely során egy osztály a működéséhez szükséges egyéb osztályokat paraméterként kapja, ahelyett, hogy helyben hozná létre. C#-ban a dependenciákat az adott osztály konstruktor paraméterként kapja meg.

Előnye, hogy ha egy osztály konstruktora megváltozik, akkor nem kell az összes rá dependáló osztályt átírni – pl. egy adatbázis technológia váltás esetén egy nagyobb projektben az adatbázis kezelő osztályra általában több száz másik osztály dependál.



```
/* DO NOT USE PATTERN! */  
/* Dependencies of a class should be  
    passed as constructor arguments*/  
public class PizzaService  
{  
    private DbContext db;  
  
    public PizzaService()  
    {  
        this.db = DbContext(/*arguments*/);  
    }  
}
```

A DI használatát a C# egy ún. DI container segíti, amelynek megadható, hogy milyen osztályokat kezeljen, majd a DI container automatikusan létrehozza a szükséges osztályokat. A DI container által ismert osztályokat ASP .NET Core környezetben Service-nek nevezzük.


A DI containerhez egy service-t 3 módon adhatunk hozzá:

1. Transient – minden alkalommal, amikor szükség van az osztályra, egy új példány keletkezik belőle.
2. Scoped – egy HTTP lekérdezés megválaszolása során egy példány jön létre az osztályból.
3. Singleton – az alkalmazás futása során egyetlen példány jön létre az osztályból.

Transient típusú objektumok csak másik Transient, Scoped és Singleton objektumokra dependálhatnak. Scoped típusú objektumok csak Scoped és Singleton objektumokra dependálhatnak. Singleton objektumok csak más Singleton objektumokra dependálhatnak.

```
/* This is the correct pattern! */
public class PizzaService
{
    private DbContext db;

    public PizzaService(DbContext _db)
    {
        this.db = DbContext(_db);
    }
}
```



Projekt előkészítése

A tantárgyhoz tartozó Teams csoportban a feltöltött fájlok között talál egy „lab –pizzashop.zip” nevű fájlt, ez egy előre elkészített projektet tartalmaz. Töltse le a fájlt, és csomagolja ki, majd nyissa meg Visual Studio-ban!

Egy újonnan létrehozott Razor Pages projektől eltérően ez a projekt egy Model és egy Services nevű mappát is tartalmaz. A Model mappában található a Pizza osztály, amely egy pizza adatait írja le (hasonlóan az előző labor anyagához). A Services mappa alatt egy IPizzaService nevű interfészt, és egy PizzaService nevű osztályt talál. A PizzaService interfész az alábbi függvényeket tartalmazza:

- Add(Pizza pizza)
 - Új pizzát hoz létre, és eltárolja a létrehozott pizzát a program memóriájában.
- List<Pizza> GetAllPizzas()
 - Visszaadja a program memóriájában lévő pizzákat.
- Pizza? GetPizzaById(int id)
 - A program memóriájában lévő pizzák közül megkeresi a megadott id-val rendelkező pizzát, és visszaadja azt.
 - Amennyiben nincs ilyen id-val rendelkező pizza, null értéket ad vissza.
- DeletePizza(Pizza pizza)
 - Eltávolítja a paraméterként kapott pizzát a program memóriájából.

A PizzaService.cs fájl egy üres implementációját tartalmazza ennek az interfésznek. Adjon hozzá egy List<Pizza> típusú statikus változót ehhez az osztályhoz, és implementálja az egyes függvényeket. Az Add függvény hívásakor az újonnan hozzáadott Pizza kapjon egy új ID-t, ami még nem volt használatban korábban (így, ha 10 pizza van a listában, törölünk kettőt, majd hozzáadunk egy újat, annak a 11-es ID-t kell kapnia). Ezt a legegyszerűbben úgy éri el, ha egy privát változóban tárolja a következő ID értéket, amit minden hozzáadáskor növel.

A Program.cs fájlban adja hozzá a PizzaService objektumot a DI konténerhez a frissen létrehozott PizzaService osztályt:

```
builder.Services.AddTransient<IPizzaService, PizzaService>();
```

Most már el tudja érni az IPizzaService osztályt az egyes Razor oldalakból, és más service-ekből. Természetesen egy valódi weboldalon az adatokat nem egy listában tárolnánk el, hanem egy adatbázisban – ezt a következő laboron fogjuk kipróbálni. A kiadott interfész tartalmaz egy Seed függvényt, amely példa adatokkal tölti fel a listát. Ezt már a Program.cs fájlban meg tudja hívni:

```
app.Services.GetRequiredService<IPizzaService>().Seed();
```

Gyakorló feladat

Hozzon létre egy Pizzas nevű oldalt a Pages mappán belül, és ott egy listában írja ki a program memóriájában lévő pizzák adatait.

- Az oldal létrehozásához kattintson jobb egérgombbal a Pages mappán, és válassza ki az Add → Razor Page ... opciót! Az oldal neve legyen „Pizzas”!
- Adja hozzá az IPizzaService interfészt a frissen létrehozott oldalhoz! Az IPizzaService interfészt két módon adhatja át az oldalnak:
 1. Ha csak az oldal megjelenítéséhez van szükség a service-re, a Pizzas.cshtml fájlban a Razor kódban az „@inject IPizzaService pizzaService” utasítás segítségével tudja „injektálni” a service objektumot. Ezután az objektum a @ karakter után pizzaService néven érhető el, és bármely függvény meghívható.
 2. A modell osztályban felvehet egy publikus property-t, és konstruktorparaméterként átveheti az IPizzaService interfész egy példányát. Mi most ezt a megoldást fogjuk használni, mivel a megjelenítésen túl később egyéb műveleteket is akarunk majd végezni (pizzák hozzáadása, törlése). Ehhez vegyen fel egy publikus IPizzaService típusú property-et a modell osztályba, és a konstruktorban vegyen át egy ugyanilyen típusú paramétert. A konstruktorban kapott paramétert tárolja el a property-ben.
- Módosítsa az Pizzas.cshtml fájlt úgy, hogy kilistázza az elérhető pizzák adatait, és jelenítsen meg egy hivatkozást az Index.cshtml oldalon, ami erre a fájlra mutat!

Az eddigi feladatok megoldását a következő oldalon találja.

Pizzas.html.cs

```
namespace PizzaShop.Pages
{
    public class IndexModel : PageModel
    {
        private readonly ILogger<IndexModel> _logger;

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

Pizzas.html (Bootstrap keretrendszer segítségével)

```
<div class="container">
    <div class="row">
        @foreach (var pizza in Model.PizzaService.GetAllPizzas())
        {
            <div class="col-4 mb-4">
                <div class="card">
                    @* You can find images in the www-root folder of the project.
                       The images are named after the pizzas, with small-case letters
                       and spaces replaced with - characters.
                    *@
                    <div class="card-body">
                        <h5 class="card-title">@pizza.Name</h5>
                        <h6 class="card-subtitle">@(pizza.Price)$</h6>
                        <p class="card-text">@pizza.Ingredients</p>
                        <p class="card-text fs-6 fst-italic">Allergens:
@pizza.Allergens</p>
                    </div>
                </div>
            </div>
        }
    </div>
</div>
```

Pizzas.cshtml (natív HTML)

```
<ul>
    @foreach (var pizza in Model.PizzaService.GetAllPizzas())
    {
        <li>
            <p>@pizza.Name (@(pizza.Price)$)</p>
            <p>@pizza.Ingredients</p>
            <p><i>Allergens: @pizza.Allergens</i></p>
        </li>
    }
</ul>
```

Interakció az oldallal

Razor Pages használata esetén a weblappal való felhasználói interakciót ún. request handler metódusokkal valósíthatjuk meg. A nevéből látható, hogy ezek olyan függvények, amelyek egy-egy HTTP request beérkezését követően a request feldolgozását végzik el. A modell osztályban ezeket a függvényeket az OnGet vagy OnPost szóval kell kezdeni, az OnPost után a handler-nek egy nevet adhatunk. Példák:

- OnGet()
 - Normál GET HTTP request esetén hívódik meg. Tipikusan olyan kódot tartalmaz, ami az oldal megjelenítéséhez szükséges információkat inicializálja.
- OnPost()
 - Normál POST HTTP request esetén hívódik meg. Tipikusan egy HTML űrlap (form) elküldésekor hívódik meg.
- OnPostRefresh()
 - Névvvel ellátott POST request – a nevet a HTTP request címében egy ?handler=Refresh paraméter jelzi. Ha egy oldalon több űrlapot akarunk kezelni (pl. törlés, hozzáadás, rendelés), akkor különböző néven kell hozzáadnunk a request handler metódusokat.

A request handler függvényeknek természetesen paramétereket is átadhatunk, alapértelmezésként a webszerver mindegyik függvényparamétert a HTTP request query paraméterei közül próbálja majd meg feltölteni (így pl. egy „int id” típusú és nevű függvényparamétert a query végén található ?id=10 string-ből kap majd értéket).

Ha egy HTML űrlapból akarjuk a paraméter értékét megkapni, akkor a függvényparaméter neve előtt egy [FromForm] attribútummal jelezhetjük ezt. Ha egy függvényparaméternek egy űrlap egy mezőjéből akarunk értéket adni, akkor a függvényparaméter nevének meg kell egyeznie a HTML űrlapon lévő mező name attribútumának értékével. A HTML űrlapon két különleges attribútumot kell még megadni: az asp-page attribútum az oldal nevét tartalmazza, ami majd kezeli a request-et, az asp-page-handler attribútum pedig a metódus nevét.

Ennek mintájára létezik egy [FromQuery] paraméter is, ami az alapértelmezett működéssel egyezik, ezt akkor érdemes használni, ha a query paramétereket és az űrlapból érkező mezőket vegyesen használjuk egyetlen függvényben.

Egyszerű példa:

Modell osztály request handler függvény

```
public void OnPostOrderPizza([FromForm] int id)
{
    /* handle ordering logic here */
}
```

Razor kód

```
<form asp-page="Pizzas" asp-page-handler="OrderPizza">
  <input type="number" name="id" />
  <input type="submit" />
</form>
```

Az így létrehozott oldal HTML kódjában a webszerver automatikusan generálja a form működéséhez szükséges adatokat, így ha megnézi az oldal HTML kódját, láthatja hogy a form kapott egy method="post" attribútumot, illetve egy action="/Pizzas?handler=OrderPizza" attribútumot.

Önálló feladat

- A letöltött projekt `wwwroot\images` mappájában talál egy-egy .jpg kiterjesztésű képet minden pizzához. Egészítse ki a Pizzas oldal kódját úgy, hogy látszódjanak a pizzák képei!
 - A képek fájlneve a pizzák nevéből úgy jön létre, hogy a szóköz karaktereket kötőjelre cseréli, majd minden karaktert kisbetűssé konvertál. Így pl. a „Buffalo Chicken” nevű pizzához tartozó kép a „buffalo-chicken.jpg” fájlban található.
 - A konverziót elvégző függvényt a modell osztályban valósítsa meg.
 - Nem létező pizzákhoz található egy not-found nevű képet. Az előzőleg létrehozott függvényt egészítse ki úgy, hogy ellenőrizze, hogy létezik-e a képfájl. Ehhez a `Directory.GetCurrentDirectory()` függvényt tudja használni, ebben kell a `wwwroot` mappa tartalmát ellenőriznie.
- Adjon hozzá egy admin oldalt a weblapjához, `Admin.cshtml` néven. Ezen az oldalon új pizzákat lehet majd felvenni, illetve törölni.
 - Adjon hozzá az oldalhoz egy HTML űrlapot, ami egy új pizza felvételét teszi lehetővé.
 - Adjon hozzá a modell osztályhoz egy event handler függvényt `OnPostAddPizza` néven, és kösse össze ezt a metódust az előzőleg létrehozott űrlappal.
 - A modell osztály konstruktorában vegyen át egy `IPizzaService` objektumot, és hívja meg az objektum `Add` függvényét az event handler függvényben! Ha mindent jól csinált, a pizzának meg kell jelennie az elérhető pizzák listájában, és a not-found.jpg képet kell látnia.
- Adjon hozzá egy event handlert `OnPostRemovePizza` néven, ami egy pizza törlését teszi lehetővé, hozzon létre ehhez egy űrlapot, és implementálja a mögötte lévő logikát! Próbálja ki, hogy tud-e törölni pizzákat.
- Adjon hozzá egy event handlert `OnPostUpdatePizza` néven, és ezt is implementálja egy megfelelő űrlappal!
 - Feltűnő lehet, hogy az `IPizzaService` interfész nem tartalmaz pizzák adatainak frissítésére szolgáló függvényt. Ennek az az oka, hogy jelenleg a pizzákat a program memóriájában egy listában tároltuk el, így ha egy pizzát lekérdez a `GetPizzaById` függvény segítségével, és annak módosítja az adatait, akkor a listában lévő adatok is módosulnak (hacsak nem hozott létre egy másolatot a lekérdezett pizzáról). Ez jelenleg nem okoz problémát, de ha az adatokat hosszú távon akarja tárolni (pl. JSON fájlban vagy adatbázisban), akkor szüksége lesz egy `UpdatePizza` függvényre.

Bónusz feladatok

- Implementálja az utolsó pontban leírt `UpdatePizza` függvényt az `IPizzaService` és `PizzaService` fájlokban.
- Oldja meg, hogy az eltárolt pizzákon végzett változtatásokat mentse el a program, és a következő indulás alkalmával olvassa vissza. Ehhez használhat például adatbázist, JSON fájlt, vagy XML fájlt.