ASP.NET Core MVC: adatbevitel és validáció

A harmadik gyakorlat célja, hogy az eddigi megjelenítési funkcionalitás mellé adatbeviteli és szerkesztési lehetőséget is kínáljunk: legyen lehetőség új teendő listák létrehozására, a meglévők átnevezésére vagy akár törlésére. A listákba vehessünk fel új elemeket (kép feltöltésével), a meglévőket szerkeszthessük, helyezhessük át másik listába vagy törölhessük.

1 Adatbeviteli és szerkesztő műveletek

Az előző gyakorlatban már megismertük az alábbi akciókat:

- *Index*: adott típusú entitás objektumok felsorolása
- *Details*: egyetlen entitás objektum részletes megjelenítése

Most kiegészítjük a `ListsController` és az `ItemsController` vezérlőket további akciókkal. A feladat megoldása során a következő műveletekkel fogunk találkozni:

- *Create*: új entitás objektum létrehozása
- *Edit*: létező entitás objektum szerkesztése
- *Delete*: létező entitás objektum eltávolítása

A szerkesztő műveletekhez (*Create*, *Edit*) két akció tartozik:

- az egyik, HTTP GET metóduson keresztül betölti a létrehozó vagy a szerkesztő űrlapot;
- a másik, HTTP POST metóduson keresztül fogadja az elküldött űrlap tartalmát és feldolgozza azt. Hiba esetén újból megjeleníti az űrlapot.

A törlés (*Delete*) akcióhoz egy kiegészítő, segédakció is tartozik: a *DeleteConfirmed* akció a törlés megerősítésére szolgál.

Megj.: az előző gyakorlat során a controllerek generálásakor létrejött minden akció a hozzájuk tartozó default kóddal. Ezekre az előző órán nem volt szükségünk.

Az űrlapokat célzó XSRF támadások ellen a már generált kód is *anti forgery token* alkalmazásával védekezik.

A műveletek tartalmazzák a modellek tulajdonságainak (*propertyk*) típusa és validációs annotációk szerinti ellenőrzéseket, a nem megfelelő bemenetet elutasítják és visszaküldik a felhasználónak az űrlapot javításra.

1.1 Nézet generálása

A vezérlőbeli akciókhoz generálhatunk önállóan is nézetet, ha erre lenne szükség. A metódus nevére jobbklikkelve válasszuk ki az *Add View* opciót, majd a felugró ablakban válasszuk a Razor View opciót.

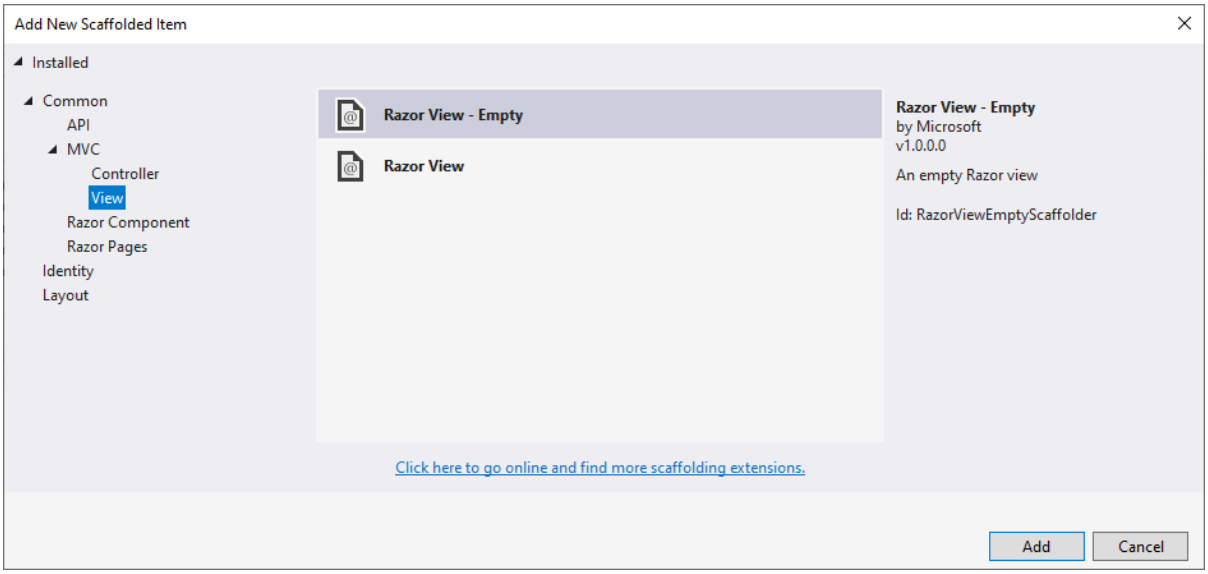

Figure 1: Nézet generálása

Ezután megadhatjuk a nézet nevét (alapértelmezetten ugyanaz, mint az akció neve), illetve kiválaszthatjuk, hogy mely entitás legyen a nézethez tartozó modell, és az adatbázis-kontextust is. Lehetőségünk van kiválasztani, hogy milyen sablon alapján generálja a VS a nézetet. Itt a lehetőségek megegyeznek az alapakciók neveivel, kivéve az `Index` akciót, amelyhez a `List` sablon tartozik. Megadhatjuk azt is, hogy milyen elrendező nézetet (*layout*) használjon a nézet. Amennyiben ezt üresen hagyjuk, a `_ViewStart.cshtml` fájlban megadott alapértelmezett beállítás jut majd érvényre.

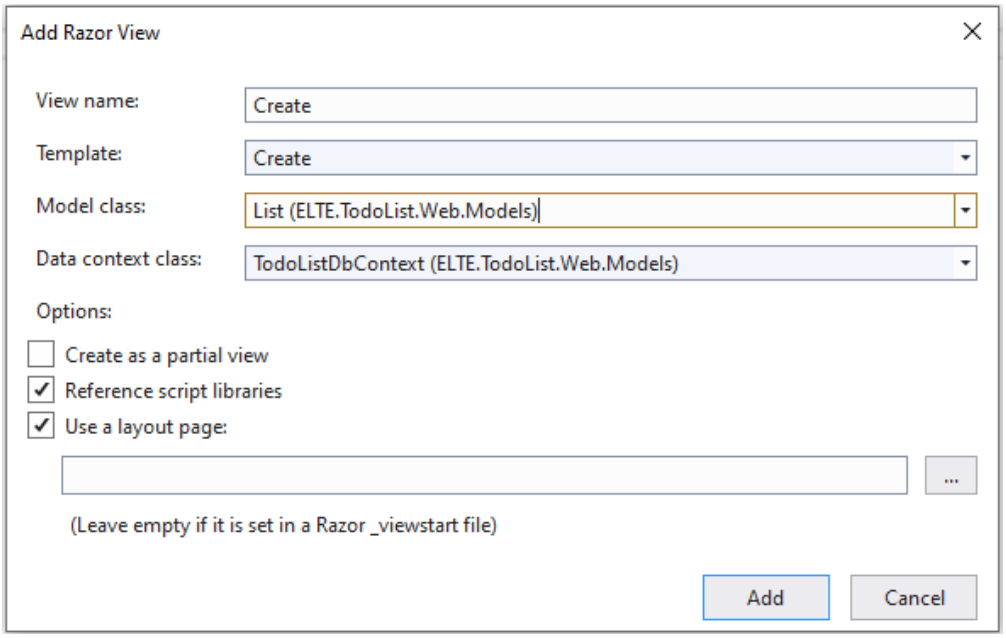

Figure 2: Nézet generálása

1.2 Listaentitás hozzáadása

Egészítsük ki a `TodoListService`-t a `CreateList` metódussal, amely hozzáadja a kapott listát az adatbázishoz (`_context.Add(list)`), majd elmenti a változtatásokat a `SaveChanges()` metódus segítségével. Figyeljünk rá, hogy a mentés `DbUpdateException`-t válthat ki, ezért a kódot helyezzük `try-catch` blokkba.

A vezérlőbeli akciók közül a `Create` felelős új lista hozzáadásáért, a nézetben pedig az azonos nevű nézet tartalmazza a lista létrehozásához szükséges űrlapot. A `Create` HTTP GET metódusa visszaadja a lista létrehozásához szükséges nézetet.

A HTTP POST metódus paraméterül megkap egy lista objektumot, amelyhez már hozzákötöttünk (*bind*) a felhasználó által bevitt adatokat. A POST metódusnak ellenőriznie kell, hogy a bevitt adatok megfelelnek-e a modellben definiált formátum- és típusmegszorításoknak (pl. ki van-e töltve minden `Required` mező, megfelelő számú karaktert tartalmaz-e egy string stb.) Az ellenőrzést a `ControllerBase` absztrakt osztályból örökölt `ModelState` objektum `IsValid` *propertyjének* vizsgálatával tehetjük meg. Ha az `IsValid` hamis értéket ad, adjuk vissza az űrlapot tartalmazó nézetet, most már a kapott listaobjektummal. Amennyiben a modellünk helyes adatokat kapott, hívjuk meg a `CreateList` metódust. A `CreateList` eredményétől függően irányítsuk a felhasználót a listákat megjelenítő oldalra a `RedirectToAction` metódussal, ha sikeres volt a lista létrehozása, ha pedig sikertelen, adjunk vissza 404-es hibakódot a `NotFound` metódus hívásával.

Generáljunk nézetet az akcióhoz (ha még nincs). Az űrlapon nem kell változtatnunk semmit, az oldal egyéb szövegein tetszőlegesen változtathatunk.

Adjunk egy linket (`<a>`) a listák felsorolásához, amely a `Create` nézetére irányít.

1.3 Listaentitás átnevezése

Egészítsük ki a `TodoListService`-t az `UpdateList` metódussal. Ez lényegében ugyanazt csinálja, mint a `CreateList`, a különbség annyi, hogy `Add(list)` helyett `Update(list)` lesz a hívás.

Az `Edit` akció HTTP GET metódusa egy (nullable) lista id-t vár paraméterül. Ellenőrzi, hogy a kapott azonosító nem null-e, majd lekéri a listát a `TodoListService` `GetListByID` metódusával. Végül visszaadja az akcióhoz tartozó nézetet a lekért listával.

A HTTP POST metódus a korábbi id-n kívül egy teljes listát is megkap a felhasználó által bevitt adatokkal. Ellenőrizzük, hogy a kapott id megegyezik-e a lista ID-jával. Ha nem, térjünk vissza `NotFound`-dal. Hasonlóan a `Create` akcióhoz, ellenőrizzük a modellt, majd hívjuk meg az `UpdateList` metódust. Annyit változtassunk a `Create` akcióhoz képest, hogy az `UpdateList` sikertelen hívása esetén írjunk ki egy hibaüzenetet a felhasználónak a `ModelState` `AddModelError` metódusának meghívásával.

Generáljunk nézetet az akcióhoz (ha még nincs). Az űrlapon nem kell változtatnunk semmit, az oldal egyéb szövegein tetszőlegesen változtathatunk.

A listák felsorolásánál minden listához adjunk egy linket az átnevezéshez.

1.4 Listaentitás törlése

Egészítsük ki a `TodoListService`-t a `DeleteList` metódussal. Itt egy id-t vegyünk át, majd a kérjük le a kontextusból az id-hoz tartozó listát. Ha megtaláltuk a listát, az előző két metódushoz hasonlóan folytassuk, ezúttal a `Remove(list)` hívással.

A `Delete` akció HTTP GET metódusa egy nullable lista id-t kap paraméterül, törzse megegyezik az `Edit` akció megfelelő metódusának törzsével.

A `DeleteConfirmed` metódus megvizsgálja, hogy a kapott id egy létező listához tartozik-e. Ha igen, akkor meghívja a `TodoListService` osztály `DeleteList` metódusát, majd visszairányít a listák felsorolásához, egyébként `NotFound`-ot ad. Lássuk el ezt az akciót az `ActionName("Delete")` annotációval.

Generáljunk nézetet az akcióhoz (ha még nincs). Az űrlapon nem kell változtatnunk semmit, az oldal egyéb szövegein tetszőlegesen változtathatunk.

A listák felsorolásánál minden listához adjunk egy linket a törléshez.

1.5 Listaelem hozzáadása

Ha még nem tettük, töröljük az `ItemsController` osztály `Index` és `Details` akciót, ugyanis ezekre nem lesz szükségünk. Töröljük a vonatkozó nézet fájlokat is, továbbá a többi nézetből az esetlegesen rájuk mutató hivatkozásokat (`<a>`).

Emeljük át a 2. gyakorlat megoldásában elkészített teendőelem-listázást. Ehhez egyszerűen a cseréljük le a `ListsController` osztály `Details` akcióját és a hozzá tartozó nézetet a korábban elkészítettre.

Helyezzünk el egy új linket a `ListsController` vezérlő `Details.cshtml` nézetében, amellyel egy új elem vehető fel a listába. Ezt irányítson a vezérlő (még létrehozandó) `CreateItem` akciójára és adja át a lista azonosítóját `id` argumentumaként.

```
<a asp-action="CreateItem" asp-route-id="@Model.Id">Add item</a>
```

Egészítsük ki a `TodoListService`-t egy `CreateItem` metódussal, amely a paraméterként átvett elemet elmenti az adatbázisba.

Egészítsük ki a `ListsController` osztályt egy `CreateItem(int id)` akcióval, amely a paraméterként átvett listához ad hozzá egy új listaelemet. Ehhez egyszerűen irányítsunk át az `ItemsController` vezérlő `Create` akciójára, de a `TempData`* tárolón keresztül adjuk át a lista azonosítóját.

```
public IActionResult CreateItem(int id)
{
    TempData["ListId"] = id;
    return RedirectToAction("Create", "Items");
}
```

Az `ItemsController` vezérlő `Create` akcióját módosítsuk, hogy a konstruált `SelectList`-ben az alapértelmezetten kiválasztott elem a megfelelő lista legyen. Ilyen módon új listaelem hozzáadásakor a legördülő menüben alapértelmezetten a megfelelő lista kerül kiválasztásra, nem a legelső.

```
ViewData["Lists"] = new SelectList(_service.GetLists(), "Id", "Name", TempData["ListId"]);
```

* *Míg a `ViewData` és a `ViewBag` konténerek a vezérlők és a nézetek közötti adatátadásra szolgálnak (a nézetmodell mellett), addig a `TempData` konténer az akciók közötti adat átadást támogatja, akár vezérlők között is. Hasznos lehet akciók közötti átirányítás esetén, vagy amennyiben két oldalbetöltés között kell adatokat ideiglenesen megőrizni. (A háttérben rövid idejű munkamenetekkel (*session*) dolgozik. A munkamenetekkel és az állapotmegőrzéssel később foglalkozunk részletesen.)*

1.5.1 Nézetmodell létrehozása és használata

Az `Item` modell `List` propertyjét korábban elláttuk a `Required` annotációval, ezért ez a mező nem lehet `null` a listaelem létrehozásakor és szerkesztésekor, Az űrlapon keresztül azonban ezt a propertyt nem tudjuk kitölteni. Jó megoldás lehet egy *nézetmodell* (*viewmodel*) elkészítése. A nézetmodell egy adatátviteli interface-t képez az entitásmodell és a controller között.

Hozzunk létre egy `ItemViewModel` nevű osztályt a `Models` mappában! Ebbe az osztályba csak azokat a propertyket emeljük át a modellből, amelyekre szükségünk van, a többit pedig más módon töltjük ki (pl. a controller akcióban). A `List` propertyn kívül mindenre szükségünk lesz az eredeti modellből, ezeket annotációkkal együtt emeljük át az `ItemViewModel`be.

Egészítsük ki az `ItemViewModel` propertyjeit a megszorításokra vonatkozó hibaüzenetekkel! Ezt az annotációk paramétereként megadott `ErrorMessage` stringgel adhatjuk meg. Jelezzük, ha valamely kötelező mező nincs kitöltve, illetve ha valahol nem megfelelő a megadott adat formátuma. Pl.

```
[Required(ErrorMessage = "A név megadása kötelező.")]
[MaxLength(30, ErrorMessage = "A listaelem neve maximum 30 karakter lehet.")]
public String Name { get; set; }
```

A nézetmodellben definiálhatunk metódusokat és operátorokat is. A későbbiekben szükségünk lesz a nézetmodell és a modell közötti konverzióra, ezért definiáljunk mindkét irányban egy-egy konverziós operátort! Ezekben feleltessük meg a kapott paraméter *propertyjeit* a visszaadott objektum *propertyjeinek*.

A vezérlőben a `Create` akció HTTP POST metódusa egy `ItemViewModel` objektumot kapjon paraméterként, amit alakítsunk `Item` objektummá. Ezután a listák létrehozásánál látott módon járjunk el. Ügyeljünk rá, hogy invalid modell esetén a adjunk a `ViewBag`-nek egy új `SelectList`-et, amelynek kiválasztott eleme az átvett paraméter `ListId` *propertyje*.

Generáljunk nézetet is az akcióhoz (ha korábban kitöröltük). A nézetben írjuk át a legördülő menü elemeinek forrását (`asp-items`) `ViewBag.ListId`-ról `ViewBag.Lists`-re.

1.6 Listaelem szerkesztése

Egészítsük ki a `TodoListService`-t az `UpdateItem` metódussal. Ez lényegében ugyanazt csinálja, mint a `CreateItem`, a különbség annyi, hogy `Add(item)` helyett `Update(item)` lesz a hívás.

Az `Edit` akció HTTP GET metódusa működjön ugyanúgy, mint a listaentitások megfelelő akciója. Itt se felejtsük el ellenőrzi, hogy a kapott azonosító nem null-e, továbbá, hogy a listaelem létezik-e?

A HTTP POST metódus a korábbi ID-n kívül egy `ItemViewModel`-t is megkap a felhasználó által bevitt adatokkal. Ez a metódus is hasonlóan működik a listák `Edit` akciójához. Érvényes modell esetén hívjuk meg a `TodoListService` osztály `UpdateItem` metódusát. Ennek eredményétől függően irányítsuk a felhasználót a listaelemek felsorolásához, vagy adjunk hibaüzenetet. Ha a modell invalid volt, tegyünk új `SelectList`-et a `ViewBag`-be.

Generáljunk nézetet az akcióhoz (ha még nincs). Az űrlapon csak a legördülő menü forrását kell megváltoztatnunk az előző akcióhoz hasonlóan.

A listaelemek felsorolásánál (listák `Details` nézete) minden elemhez adjunk egy linket, ami a szerkesztő nézethez vezet.

1.7 Listaelem törlése

Egészítsük ki a `TodoListService`-t a `DeleteItem` metódussal. A törlést itt is hasonlóan végezzük el, mint a listáknál: keressük meg a paraméterként kapott id-hoz tartozó listaelemet, majd töröljük. Ne felejtsük el elmenteni a változtatásokat.

A `Delete` akció HTTP GET metódusa egy nullable listaelem id-t kap paraméterül, törzse megegyezik az `Edit` akció megfelelő metódusának törzsével.

A `DeleteConfirmed` metódus megvizsgálja, hogy a kapott id egy létező listaelemhez tartozik-e. Ha igen, akkor meghívja a `TodoListService DeleteItem` metódusát, majd visszairányít a listaelemek felsorolásához, egyébként `NotFound`ot ad. Lássuk el ezt az akciót az `ActionName("Delete")` annotációval.

Generáljunk nézetet az akcióhoz (ha még nincs). Az űrlapon nem kell változtatnunk semmit, az oldal egyéb szövegein tetszőlegesen változtathatunk.

A listaelemek felsorolásánál (listák `Details` nézete) minden listelemnél adjunk egy linket a törléshez.

2 Képek feltöltése

Valósítsuk meg, hogy a listaelemekhez létrehozáskor és szerkesztéskor legyen lehetőség egy képet feltölteni. Mivel a megjelenítéskor PNG állományokra számítunk, ezért garantáljuk, hogy felölteni is csak ilyen formátumú képeket lehessen.

Az `ItemsController` vezérlő `Create` és `Edit` akcióit, valamint a vonatkozó nézeteket kell módosítanunk. Egészítsük ki a két metódus paraméterlistáját egy `IFormFile? image` paraméterrel. Amennyiben a feltöltött `IFormFile? image` objektum nem `null` és a feltöltött fájl nem üres (a `Length` property értéke pozitív), akkor nyissunk egy új `MemoryStream` típusú adatfolyamat, amelybe betölthetjük a feltöltött fájlt. Innen a bájttömböt már könnyedén az `item` entitásobjektumba másolhatjuk.

```
using (var stream = new MemoryStream())
{
    image.CopyTo(stream);
    item.Image = stream.ToArray();
}
```

Az `Edit` akcióban ezen felül gondoskodnunk kell arról, hogy ha egy listaelem módosításakor nem töltünk fel új képet, a frissítés ne írja felül az elemhez tartozó korábbi képet (ha volt ilyen), mivel ilyen esetben az `IFormFile` paraméter értéke `null` lesz. Egészítsük ki a `TodoListService`-ben az `UpdateItem` metódust, hogy az elem frissítésekor kihagyja a képet a *propertyk* közül, ha nem adtunk meg új képet.

```
_context.Update(item);
if (item.Image == null)
{
    _context.Entry(item).Property("Image").IsModified = false;
}
_context.SaveChanges();
```

A nézetekben a `<form>` elemet ki kell egészítenünk a `enctype="multipart/form-data"` attribútummal, hogy a fájlfeltöltést az űrlap számára engedélyezzük. A fájltallózó mezőt a következő módon definiálhatjuk:

```
<input asp-for="Image" type="file" accept="image/png" />
```

Valósítsuk meg, hogy szerkesztéskor jelenjen meg az aktuális kép a fájltallózó beviteli mező felett, hasonlóan a `ListsController` osztály `Details.cshtml` nézetéhez. Ügyeljünk arra, hogy a kép megléte opcionális.

> *Megjegyzés:* a beviteli mezőben az `accept="image/png"` attribútum csak kliensoldali megszorítást jelent a feltölthető fájlok típusára, és ilyen módon könnyen megkerülhető. Amennyiben mindenképpen garantálni szeretnénk, hogy csak PNG formátumú fájlt lehessen feltölteni, szerver oldali ellenőrzésre is szükség van, de ez nem triviális. Ellenőrizhetjük például, hogy a fájl a PNG állományoktól elvárt fejléccel kezdődik-e (az első 8 bájt rendre: `137 80 78 71 13 10 26 10`), azonban ez sem garantálja, hogy kép például nem sérült-e.

3 Erőforrások kötegelése és méretcsökkentése

Nagyobb webalkalmazások számos CSS és JavaScript állományt betölthetnek a weboldallal együtt. A probléma ezzel, hogy minden egyes fájl lekéréséhez új HTTP kapcsolatot kell felépíteni a kliens (a böngésző) és a szerver között. Így tulajdonképpen nem is a fájlok mérete, hanem azok mennyisége ronthatja már a weboldal betöltésének idejét.

A megoldást ezen erőforrás állományok kötegelt, egyetlen állományként történő letöltése jelenti (angol szakszóval *bundling*). További optimalizációt jelenthet, ha a CSS és a JavaScript kódfájlokból a fölösleges whitespace karaktereket eltávolítjuk, ezzel csökkentve a méretüket (*minifying*).

Ez a feladat jól automatizálható, egy ASP.NET Core webalkalmazás esetében a `BuildBundlerMinifier` NuGet csomag lehet ebben segítségünkre, amely a projekt minden fordításakor a megadott konfiguráció szerint kötegeli és "tömöríti" a CSS és JavaScript állományainkat.

Adjuk hozzá a nevezett NuGet csomagot a projektünkhöz, valamint egy `bundleconfig.json` konfiguráció állományt a projektkönyvtár gyökerébe az alábbi tartalommal:

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
      "wwwroot/css/site.css"
    ]
  }
]
```

Ezzel megadtuk, hogy a `site.css` állományból kell elkészíteni a `site.min.css` *minified* CSS fájlt (több, mint 50%-os méret csökkenés). Több input fájl megadásával kötegelést is végezhetnénk.

Megadhatjuk azt is, hogy fejlesztés közben az eredeti CSS és JavaScript erőforrás állományokkal dolgozzunk (a forrásukat könnyen át tudjuk tekinteni hibakeresés esetén), de éles környezetben az optimalizált változatot használjuk. Ehhez a `_Layout.cshtml` elrendező nézetben a következő módon hivatkozzuk a fájlt:

```
<environment include="Development">
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
```

```
<environment exclude="Development">
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

Megjegyzés: hasonló elven a képfájlok betöltése is optimalizálható, amennyiben sok kis kép helyett az összes képet egy nagyobb állományra másoljuk és úgy töltjük le, ezeket *sprite*-oknak nevezzük. Erre azonban a CSS kódot is fel kell készíteni, így ezt most nem tárgyaljuk részletesebben.