Modulfejlesztés – Gyakorlati tanácsok

A DotNetNuke (DNN) modulfejlesztés során van néhány olyan speciális körülmény, ami miatt az ilyen jellegű fejlesztés eltér a megszokott, önálló alkalmazásfejlesztéstől. Más tekintetben viszont szabványos megoldásokkal dolgozhatunk. Ez a dokumentum leírja ezeket az eltéréseket, illetve nagy vonalakban bemutat a DNN modulfejlesztés során használható standard megoldásokat is.

Feltételezzük, hogy a modulfejlesztés alapjai (fejlesztőkörnyezet kialakítása, modulprojekt létrehozása) már ismertek, ezért ezekkel itt most nem foglalkozunk.

A kialakított modul forráskódja hozzáférhető az alábbi címen:
https://github.com/adam-halassy-stud/bce-irf

DNN (modul) architektúra

A DNN egy ASP.Net Framework 4.7.2 keretrendszeren futó, vegyesen WebForms és MVC megközelítést alkalmazó CMS rendszer, ami MsSQL adatbázis felett, Windows környezetben fut.

Amit mi „modul” néven emlegetünk, az valójában a DNN terminológiája szerint „extension” (kiterjesztés) Egy extension több modult tartalmazhat. A modulokat az egyes oldalak struktúrájába, a lapot meghatározó sablon szerint elérhető dobozokba tudjuk beilleszteni, és ott is fognak megjelenni. Egy lapon tetszőleges számú, akár egymással kooperáló modul is elhelyezhető.

A dokumentum további részében az egyértelműség kedvéért már extensionként (kiterjesztésként) fogjuk említeni a modulprojektet.

Extension-ök helye a fájlrendszerben

A modulok kötött helyen találhatóak a fájlrendszerben. A DNN telepítés könyvtárán belül a modulok a „DesktopModules” könyvtárban találhatóak. Ezen a mappán belül további mappák alá lehet bontani a modulkönyvtárakat. Kifejezetten ajánlott az esettanulmányok megoldása során egy közös könyvtárba gyűjteni a feladatok megoldása során létrehozott modulokat.

A „bin” könyvtárak

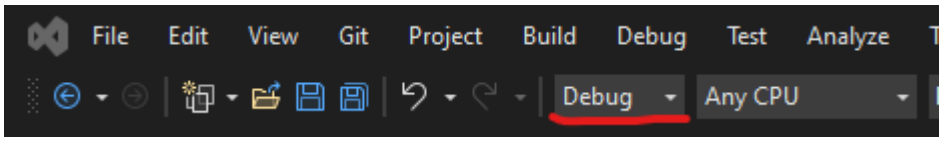
Az ASP.Net alkalmazások assembly-jei (szerelvényei), vagyis az alkalmazás MSIL (Microsoft Intermediate Language) kódját tartalmazó fájlok a „bin” könyvtárakban találhatóak. A DNN telepítés gyökérkönyvtárában is található egy ilyen bin könyvtár, ebben találhatóak a rendszer binárisai. A modulfejlesztés szempontjából azért fontos ez a könyvtár, mert ha a fejlesztés során szükségessé válik a DNN kódbázis hivatkozása, akkor azt az itt található assembly-k hivatkozásával lehet megtenni.

Fontos, hogy az assembly hivatkozások mindig relatív hivatkozások legyenek. A Visual Studio szerencsére alapértelmezés szerint relatív hivatkozásokat hoz létre.

Első lépések a projekt létrehozása után

A projekt létrehozása után az új kiterjesztést még telepíteni kell a DNN rendszerébe. Ehhez a következőket kell tenni.

1) Le kell fordítani a projektet „Debug” konfigurációban.
2) Le kell fordítani a projektet „Release” konfigurációban
3) A sikeres fordítások eredményeképpen, az „install” könyvtárban létrejött csomagot telepíteni kell a DNN adminisztrációs felületén.



Ez a solution most „Debug” konfigurációra van állítva

A „Debug” konfigurációs fordítás során kiderül, hogy minden projektbeállítás rendben van, illetve a fordítás eredményeképpen létrejönnek a nyomkövetési (debug) információkat tartalmazó binárisok.

A „Release” konfigurációs fordítás során optimalizált és nyomkövetési (debug) információkat nem tartalmazó binárisok jönnek létre. Ez a lépés a DNN template-ből létrehozott projekt során kiegészül a kiterjesztéscsomagok létrehozásával. Mindig két csomag jön létre: az egyik csomag csak a modul futtatásához szükséges fájlokat tartalmazza, a másik pedig a projekt összes forrásállományát (ez a csomag „source” végződésű)

Maga a csomag egy ZIP archívum. Általában elegendő a kisebb, csak a futtatható állományokat tartalmazó fájl használata.

Fejlesztési folyamat

Egy DNN kiterjesztés fejlesztési folyamata igényel némi manuális munkát az egyéb fejlesztésekkel összehasonlítva.

Strukturális módosítások (adatbázis struktúra, vagy .dnn fájl módosítása) után mindig újra kell telepíteni a kiterjesztést.

Ha csak a kiterjesztés kódja változik (C#, cshtml, css, js, stb fájl), akkor nem szükséges újratelepíteni a kiterjesztést. Ha C# forrás változik, akkor mindenképpen újra kell fordítani a projektet. Ilyenkor a bináris változásait detektálja az IIS, és automatikusan újraindítja a weboldalt. Emiatt a fordítás utáni első request kiszolgálása lehet szignifikánsan lassú, viszont a friss kód fog futni. Minden más esetben elegendő újratölteni az oldalt, az ismételt betöltéskor már a friss css/js/cshtml (egyéb fájl) tartalmával fog dolgozni az IIS.

MVC modell

A DNN támogatja a régebbi WebForms, és az MVC megközelítést is. Mivel mára az MVC az elterjedtebb (és a .Net Core már nem is támogatja a WebForms keretrendszert), ezért mi is ennek a használatát mutatjuk be.

Az MVC modell leírása

Az MVC modell a Modell-View-Controller szavak kezdőbetűiből áll össze, amik a struktúra három komponensét jelentik.

View

A három komponens közül a View (Nézet) neve magáért beszél: ez az, amit a felhasználó lát maga előtt. Kicsit tágabban értelmezve a View tartalmazza azokat a komponenseket is, amelyek a felhasználói interakciókért felelősek (pl. gombok és beviteli mezők) A mi esetünkben a view-k cshtml fájlok lesznek majd.

Fontos, hogy a view-k kódjában csak a megjelenítéshez szigorú értelemben vett kódok lehetnek. Például egy lista megjelenítésekor a View számára szűrt, rendezett és feldolgozott adatsort kell átadni, a View ebben az esetben legfeljebb az adatokat táblázatban megjelenítő ciklust tartalmazhat.

Model

A Model a view-n megjelenített adatokat tartalmazó adatstruktúra. Egyszerű esetben ez megegyezhet az ORM entity osztályával, de általában komplexebb adatokat kell átadni a view számára, hogy az az elvárásoknak megfelelő kimenetet tudja renderelni. (pl. egy „Order” entity-hez biztosan kapcsolni kell a „Customer” tábla releváns rekordját, és ezeket az adatokat egy model osztályba kell szervezni)

ASP.Net Framework használata esetén a View számára két módon tudunk model adatokat átadni. Egyrészt az @model annotáció segítségével lehet átadni a szigorú értelemben vett model objektumot. Kiegészítő adatokat a „ViewBag”-on keresztül lehet átadni. Ilyen kiegészítő adat lehet egy legördülő mezőhöz használt lista például. A ViewBag maga egy ún. dinamikus objektum, ami azt jelenti, hogy ennek a tulajdonságai nem fordítási időben jönnek létre, hanem futásidőben, statikusan. Ennek megfelelően a fordító nem tud figyelmeztetni, ha olyan tulajdonságokat próbálunk használni, amik nem léteznek, illetve a Visual Studio sem tud segíteni, hogy mik létezhetnek.

Objektum vagy osztály? Az OOP paradigma során objektumnak nevezzük a memóriában létező, példányosított objektumot, osztálynak pedig az osztály statikus struktúráját. Ebben a dokumentumban elsősorban futásidejű megközelítéssel dolgozunk, ezért többnyire objektumokról fogunk beszélni, de ezek leírt formájukban természetesen osztályok.

Controller

A View-n manifesztálódott felhasználói input-ok feldolgozásáért és a válasz rendereléséért a kontroller objektumok felelősek. A kontroller feladata, hogy az inputnak megfelelő műveletet végrehajtsa. Ez lehet egyszerűen egy lap renderelése (pl. megnyitjuk az index lapot), de lehet egy űrlap feldolgozása, vagy válaszadás egy REST API hívásra.

Modul action hozzáadása

Mivel maga az ASP.Net ún. „convention over configuration” (konvenció a konfiguráció helyett) megközelítést alkalmaz, ezért általában egy új kontroller vagy action hozzáadása az osztály vagy metódus implementálásánál többet nem igényel. A DNN számára azonban deklarálni kell az új végpontokat (egy kontroller egy metódusát)

A kiterjesztéseket egy „.dnn” kiterjesztésű fájl konfigurálja. Ebben a fájlban vannak leírva a kiterjesztések legfontosabb tulajdonságai, a metaadatok, az adatbázis migrációk és a kiterjesztés által kiajánlott modulok. Minden modul egy-egy végpont (controller action-je) Az egyes végpontokat le kell írni a .dnn fájlban. Egy ilyen végpont leírása sokféle metaadatot tartalmaz. Nekünk jelenleg a következőek kiemelten fontosak:

„controlKey” – Végpont azonosítója.
Ennek a kiterjesztésen belül mindenképpen egyedinek kell lennie.

„controlSrc” – Végpont útvonal leírása.
Ez egy speciális útvonalleírás, ami tartalmazza a végpontot megvalósító kontrollerosztály névterét, az osztály és az action metódus nevét. A kontrollerosztály nevének végéről a „Controller” utótagot el kell hagyni. A formátuma a következő:

```
{Controller.Class.Namespace}/{ControllerClassName}/{ActionMethodName}.mvc
```

Példa:
`RF.Modules.TestFlightAppointment.Controllers/TestFlightGrid/Create.mvc`

Ez az útvonal „RF.Modules.TestFlightAppointment.Controllers” névtérben található „TestFlightGridController” osztály „Create” metódusára mutat.

„controlType” – Típusa
A mi esetünkben ez mindig „View”.

„supportsPopUps” (opcionális) – Popup megjelenés támogatása
Akkor kell megadni true értékkel, ha egy popup ablakot akarunk megjeleníteni a DNN infrastruktúráján keresztül.

Fontos! A .dnn fájl módosítása után mindig újra kell telepíteni a csomagot.

Modul view hozzáadása

A view-ket leíró .cshtml fájlokat az ASP.Net MVC-ben megszokott módon, a View/{ControllerName}/ könyvtárba kell elhelyezni. Layout-ok és partial view-k ugyanúgy használhatóak.

Fontos, hogy a View-k ősosztálya a „`DotNetNuke.Web.Mvc.Framework.DnnWebViewPage<T>`” osztály. Ennek a használatával elérhetővé válik a DNN keretrendszere által biztosított környezet (pl. aktuálisan bejelentkezett felhasználó adatai)

Felugró ablakok

A DNN-ben a felugró ablakok iFrame-n belül jelennek meg. (Ilyen végpont esetén meg kell adni a „supportsPopUps” értéket a .dnn fájlban)

Popup megjelenítéséhez elegendő a „supportsPopUps” értékét „true”-ra állítani a .dnn fájlban. Ilyen típusú URL feldolgozásakor a DNN automatikusan tudja, hogy egy popup-ot kell megjelenítenie.

Példa:

```
<a href="@Url.Action("Detail", "TestFlightGrid", new {ctl = "Detail", bookingID =
booking.BookingID })">
```

A releváns modulban a „Detail” action popup-ként van megjelenítve. Ez a példa egy olyan linket renderel, ami a megjelenő popup számára átadja egy foglalás azonosítóját, így a megjelenő popup a kiválasztott foglalás adatait jeleníti meg. Vegyük észre, hogy standard ASP.Net MVC eszközöket használtunk.

Kiterjesztés specifikus stílus (CSS) integrálása

A DNN a kiterjesztések könyvtárában található „module.css” fájlt automatikusan csatolja a megjelenített oldalhoz, ha az tartalmaz olyan modult, amit a kiterjesztés tartalmaz. Ezért a moduljainkhoz szükséges CSS stílusokat a „module.css” fájlban kell megadni. A fájl módosítása után nem szükséges újratelepíteni a kiterjesztést, vagy manuálisan újraindítani az IIS-t.

JavaScript használata

A moduljainkban használhatunk CSS-t. A DNN alapértelmezés szerint rendelkezésünkre bocsátja a saját keretrendszere mellett a jQuery keretrendszert.

Egy view-n használt JS kódot mindenképpen regisztrálnunk kell a DNN számára. Lent látható egy ilyen példa.

```
@using DotNetNuke.Web.Client.ClientResourceManagement
@using DotNetNuke.Framework.JavaScriptLibraries

...

@{
    ClientResourceManager.RegisterScript(
        Dnn.DnnPage,
        "~/DesktopModules/MVC/RF.Modules.TestFlightAppointment/Scripts/BookingGrid.js"
    );
```

A ClientResourceManager osztály RegisterScript metódusával az „Rf.Modules.TestFlightAppointment” modulban található, a „Scripts/BookingGrid.js” JavaScript fájlt hivatkoztuk be arra az oldalra, ahol a modulunk megjelenik.

A modulban használt további JavaScript kódokat célszerű a forrás végére, <script></script> tag-ek közé tenni, illetve az inicializálást igénylő műveleteket (pl. eseményfeliratkozás) a jQuery „onRead” eseménykezelőjében megírni.

JavaScript a popup ablakokban

Mivel a popup ablakokat iframe-n belül rendereli a DNN, ezért külön kérés nélkül a popup ablakokban nem érhetőek el a JavaScript kódok, a keretrendszeren keresztül kérnünk kell, hogy megjelenítse őket. A lenti példa a jQuery-t, a DNN saját keretrendszerét és az ajax hívások támogatásához szükséges JS library-ket igényli. (Az ajax hívásokhoz használt JS library-ra van szükségünk REST API-k hívásához is!)

```
DotNetNuke.Framework.JavaScriptLibraries.JavaScript.RequestRegistration(CommonJs.jQuery);
DotNetNuke.Framework.JavaScriptLibraries.JavaScript.RequestRegistration(CommonJs.DnnPlugins);
ServicesFramework.Instance.RequestAjaxScriptSupport();
```

REST API kontroller

A DNN architektúrája lehetővé teszi, hogy REST API végpontokat implementáljunk. Ezeket, a modul kontrollerektől eltérően nem kell regisztrálni a .dnn fájlban.

A valamilyen API végpontot megvalósító kontrollerosztályok ősosztálya mindig a DnnApiController osztály, amit a DNN egyedi módon kezel. Egy API végpont URL-jét a modul nevének és azonosítójának ismeretében tudjuk elkérni.

```
        var sf = $.ServicesFramework('@Dnn.ActiveModule.ModuleID');
        var serviceUrl = sf.getServiceRoot('TestFlightBooking');
```

Ez a példakód az éppen renderelt modultól kéri el a „TestFlightBooking” nevű modul API végpontjainak az elérését. A tényleges híváskor az így megkapott URL mögé kell fűzni a kontroller és az action nevét.

```
var sf = $.ServicesFramework('@Dnn.ActiveModule.ModuleID');
var serviceUrl = sf.getServiceRoot('TestFlightBooking');
var url = serviceUrl + ”Booking/Detail”;
```

Itt az utolsó sor azzal egészíti ki a felső kódrészletet, hogy a serviceUrl változóban levő érték után fűzi a kontroller és action pontos nevét és így elérési útvonalát.

Adatbázis kezelés a DAL2 infrastruktúrájával

A DNN az adatbázis eléréséhez az EntityFramework-höz (EF) hasonló, de attól lényeges pontokon eltérő, saját fejlesztésű ORM modellt használ, ezt hívja DAL2-nek.

Migráció

A DNN Database first megközelítést alkalmaz, ami azt jelenti, hogy az ORM modell létező adatbázis modellre csatlakozik. Mivel a modulok telepítésének teljesen automatizáltnak kell lenniük – a modult telepítő felhasználó esetleg nem is rendelkezik közvetlen adatbázis-eléréssel – ezért a modulcsomagnak tartalmaznia kell az adatbázis séma létrehozásához szükséges szkripteket.

A DNN esetében ezek interpretált SQL szkriptek. Az interpretáció ebben az esetben azt jelenti, hogy az SQL szkriptekben az SQL objektumazonosítók (táblák, tárolt eljárások, view-k, stb) előtt a „{databaseOwner}{objectQualifier}” makrót kell szerepeltetni. Emiatt az SQL Server Manager Studio (más más SQL kliensek) nem tudják közvetlenül futtatni ezeket a szkripteket, csak a DNN telepítője.

A kiterjesztés telepítésekor az SQL szkriptek a kiterjesztés első telepítésekor és a javítótelepítés futtatásakor futnak meg. Ha valamilyen hiba lép fel, a kiterjesztés telepítése sikertelen, illetve az SQL hibaüzenet megjelenik a telepítés naplójában.

A lenti példában egy tábla létrehozását láthatjuk. Figyeljük meg a „*{databaseOwner}{objectQualifier}*” makrókat a tábla neve előtt.

```
CREATE TABLE {databaseOwner}{objectQualifier}TestFlightBookings
    (
        [BookingID] INT NOT NULL IDENTITY (1, 1)
                CONSTRAINT [PK_TestFlightBookings] PRIMARY KEY CLUSTERED,
        [CreatedByUserID] [int] NULL,
        [CreatedOnDate] [datetime] NULL,
        [IsCancelled] BIT,
        [DepartureAt] [datetime],
        [Duration] INT,
        [FlightPlanID] INT

    )  ON [PRIMARY]
GO
```

A DAL2 ORM

A DAL2 terminológiája szerint az adatbázist „DataContext”-nek nevezzük. Az adatbázis egy tábláját „Repository”-nak. A repository-kat a táblához létrehozott entity rekordok segítségével lehet azonosítani. Az entity rekordok felépítésekor az EF-hez hasonló, de attól eltérő annotációk használhatóak.

A lenti példa a „TestFlightPlans” táblához készült DTO osztály. Figyeljük meg a következőket:

- A [TableName] attribútum deklarálja a táblanevet

- A [PrimaryKey] attribútum deklarálja, hogy autoincrement elsődleges kulcsot használunk
- A TotalDuration tulajdonság [IgnoreColumn] attribútuma azt jelenti, hogy nincs megfelelője az adatbázisban.

```
[TableName("TestFlightPlans")]
[PrimaryKey(nameof(FlightPlanID), AutoIncrement = true)]
[Cacheable("TestFlightPlan", CacheItemPriority.Default, 20)]
[Scope("ModuleId")]
public class TestFlightPlan
{
    public int FlightPlanID { get; set; }

    public string Name { get; set; }

    public int Duration { get; set; }

    public string Description { get; set; }

    public bool IsPublic { get; set; }

    [IgnoreColumn]
    public int TotalDuration => Duration + 1;

}
```

Lekérdezések és keresés

Az EF-el ellentétben a LinqSQL keresései nem használhatóak, ehelyett a .Find() metódust kell használni. Ez a metódus a Linq-ban használt „.Where()” megfelelője, azzal a lényegi különbséggel, hogy a paramétere nem lambda kifejezés, hanem egy valid SQL WHERE feltétel.

FONTOS! A feltételt szövegesen és paraméterezetten kell megadni. Nem szabad a keresési értékeket konkatenálni a feltétel kifejezésbe, mert azzal potenciálisan lehetővé válik az alkalmazás támadása SQL injection segítségével.

A lenti példában láthatjuk, hogyan lehet egy rekordot (TestFlightPlan) azonosító alapján megkeresni, illetve a hozzá tartozó „TestFlightParticipant” rekordok paraméterezett keresését.

```
var plan = ctx.GetRepository<TestFlightPlan>()
    .GetById(booking.FlightPlanID);

var participants = ctx.GetRepository<TestFlightParticipant>()
    .Find("WHERE BookingID = @0", bookingID)
    .ToArray();
```