ASP.NET MVC Web API

Reiter István

2013

Tartalomjegyzék

Web API bevezetés ... 3
Kliens oldal ... 8
Knockout ... 17
Knockout + Web API ... 22
Authentikáció ... 28
ToDoSite ... 30
Entity Framework Code First bevezető ... 31
A Repository minta ... 39
Entity Framework műveletek ... 44
ToDoSite – Model ... 46
Twitter Bootstrap ... 49
ToDoSite – View ... 51
ToDoSite – Controller ... 54
ToDoSite – Kliens oldal ... 58

Web API bevezetés

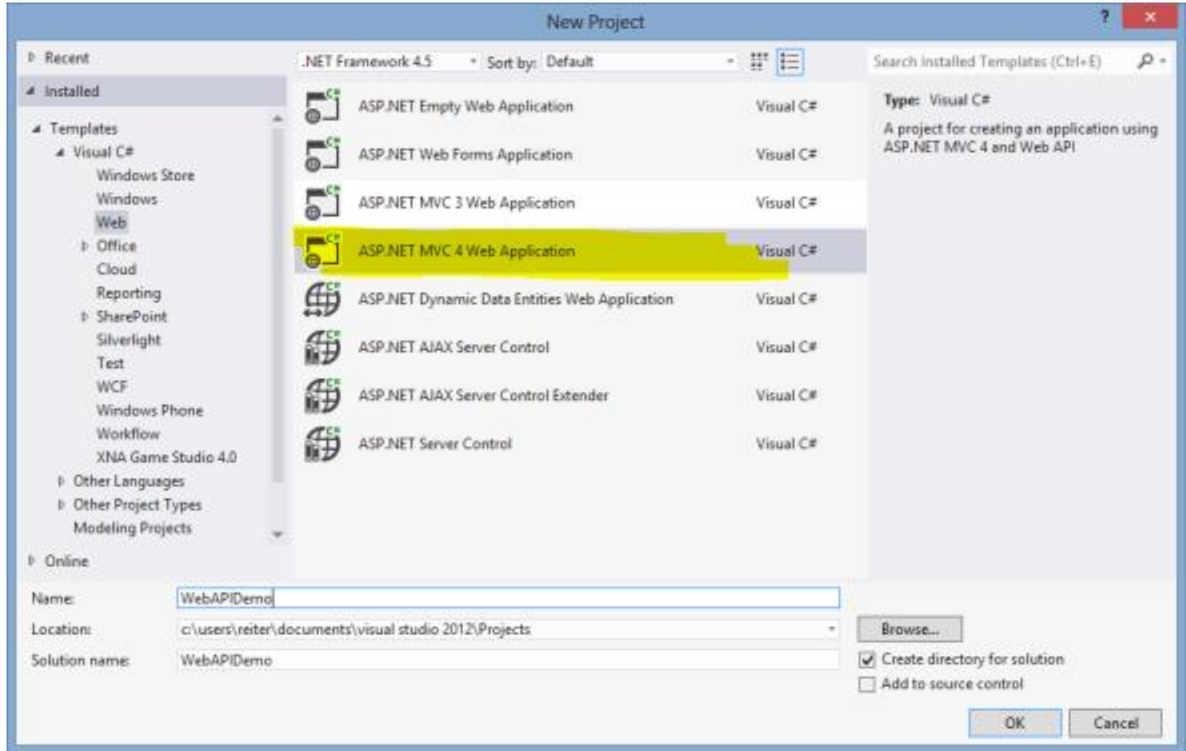
Legalább említés szintjén biztosan mindenki találkozott a WCF-fel (Windows Communication Foundation), amely a .NET szolgáltatás-orientált alkalmazások (SOA – Service Oriented Archtitecture) készítésére szolgáló platformja. Erről azt kell most csak tudni, hogy nagyjából minden transzportációs réteg felett képes működni (HTTP, TCP, rántott hús stb.) méghozzá elsődlegesen SOAP protokollal együttműködve.

Már nem fogok sokáig mindenféle rövidítésekkel dobálózni. Szóval, tegyünk egy nagy kitérőt: felmerült az igény, hogy ne csak SOAP támogatás legyen, és főleg HTTP protokoll felett, hanem egy jóval egyszerűbb HTTP kéréseken alapuló is, amely mondjuk XML vagy JSON adatformátumokat dolgoztat (így lényegében RESTful architektúrát alakítva ki, erről majd később). Így aztán a WCF egy melléktermélke lett a WCF Web API, ami végül beleolvadt az ASP.NET-be.

Nagy vonalakban a következőről van szó: én a böngészőmből elindítok, mondjuk egy POST vagy GET kérést, pont olyat amit mondjuk bármely más szerveroldali platform képes kezelni. Ezt a kérést a túloldalon a Web API fogadja, feldolgozza és visszalök egy JSON csomagot, amelyet én a kliens oldalon fel tudok dolgozni. Amit most leírtam az egyáltalán nem újdonság, ám van néhány dolog ami érdemes megemlíteni: elsősorban, azt, hogy ezt az egész folyamatot leprogramozni pokolian egyszerű, gyakorlatilag sima CLR metódusokat kell készítenünk, minden más teljes mértékben automatizálva van. Másodsorban nem csak böngészőkről van ám szó, gyakorlatilag bármilyen platform számára megnyílik az alkalmazásunk, amely képes HTTP kéréseket küldeni/fogadni.

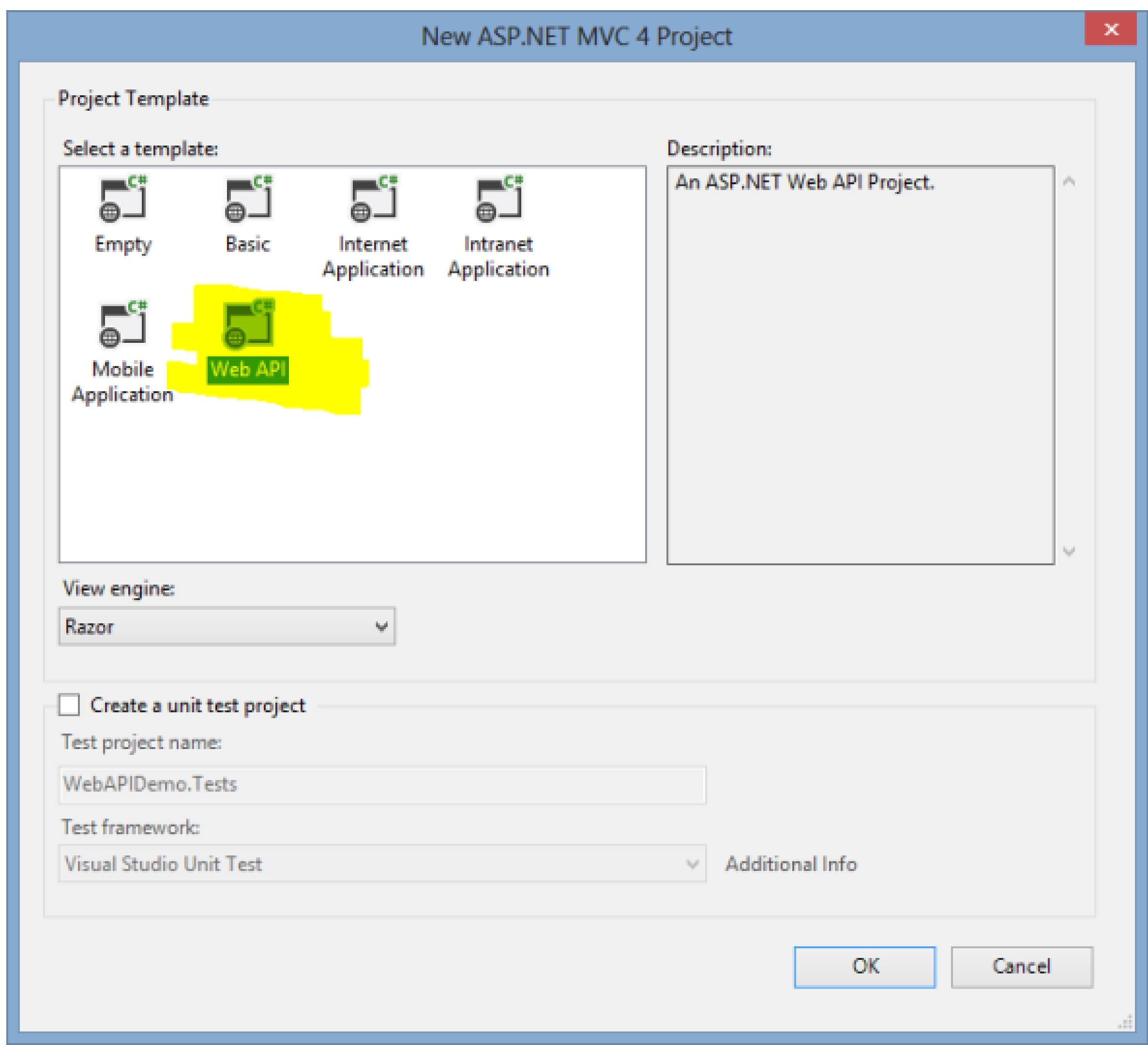
Elég a beszédből, ideje elkészíteni első Web API alkalmazásunkat. Ehhez szükségünk lesz az ASP.NET MVC 4 csomagra, ami Visual Studio 2012 esetén már fent van, VS 2010 esetén külön kell telepíteni a Web Platform Installer segítségével (ez az Expressekre is vonatkozik). Én a VS 2012-t ajánlom a jobb Javascript integráció miatt (is).

Tehát, New Project -> Web -> ASP.NET MVC 4 Web Application:



Majd a Web API sablont. Érdemes megjegyezni, hogy minden sablonhoz hozzáadhatunk később Web API controller-t. Maga a sima Web API sablon lényegében meztelen, az alapfunkciókon kívül nincs semmi megvalósítva benne. Amennyiben ennél többet szeretnénk, használjuk inkább az Internet Application sablont. Különösen akkor van ennek értelme, ha szeretnénk felhasználó azonosítást is az alkalmazásunkba. Az authentikációs szolgáltatások bekapcsolása meglehetősen hosszadalmas és favágó munka, így rengeteg időt megspórolhatunk a „teljes" sablon használatával. Az Internet Application sablon ráadásul előre be is van konfigurálva Web API controlllerek fogadására, ezért nagyjából mindig jobban járunk, ha ezt választjuk.

Ebben az esetben a kivételt erősítő szabály lép életbe, tanulási célokra tökéletesen megfelel a szűz Web API sablon is, később majd áttérünk a másikra is.



Létrejön a projekt. Vegyük észre, hogy a web-szolgáltatást tartalmazó osztályunk az ApiController osztályból származik, a Web API ezzel az osztállyal dolgozik. A másik érdekes dolog az előre generált metódusok.
A Web API gyakorlatilag négyféle funkciót tesz számunkra elérhetővé: GET (lekérdezés), POST (beillesztés), PUT (módosítás) és DELETE (törlés). Ezek megfelelnek a HTTP protokollban lévő függvényeknek. Amit viszont tudni kell, hogy a metódusok nevei esetében be kell tartanunk azt a szabályt, hogy a HTTP függvénynek megfelelő előtagot kell kapjanak. Tehát, ha van egy szolgáltatásom, ami egy listát vagy elemet ad vissza annak a neve Get-tel kezdődik, pl GetList.

Készítettem egy nagyon egyszerű alkalmazást a a platform képességeinek szemléltetésére, ebben a bejegyzésben a GET műveletekkel fogunk megismerkedni. Gyakorlatilag egy sima CLR listát fogunk használni, a műveletek pedig egy-egy sort foglalnak. Ennél több dolgunk egész egyszerűen nincs is most szerveroldalon.

```
public class ValuesController : ApiController
{
    static public List<string> Names = new List<string>()
    {
        "István", "Viktória", "Béla", "Balázs",
        "Judit", "Miklós", "Ágnes", "Edina"
    };

    // GET api/values
    public IEnumerable<string> Get()
    {
        return Names;
    }

    // GET api/values/5
    public string Get(int id)
    {
        return Names[id];
    }

    // POST api/values
    public void Post([FromBody]string value)
    {
    }

    // PUT api/values/5
    public void Put(int id, [FromBody]string value)
    {
    }

    // DELETE api/values/5
    public void Delete(int id)
    {
    }
}
```

Ugyan a program menetkész, de jó lenne azt is tudni, hogyan csalogathatjuk elő az értékeket. Az App_Start mappában megtalálható WebApiConfig.cs fájlban lévő forráskód erre is megadja a választ:
Például, ha én az összes nevet ki szeretném listázni akkor a következő címet használom:

http://localhost:38405/api/values

Az http://localhost:38405/api/ az alap amiből kiindulunk, ha a Web API szolgáltatásait akarjuk elérni. Ezután jön a controller neve. Amennyiben nem írunk mást, akkor az a metódus fut le, amely nevének előtagja Get, nincs paramétere és IEnumerable típussal tér vissza. Ha több ilyen metódus van, akkor kivétel keletkezik.
A fenti cím vagy megjeleníti a böngészőben az XML-t, vagy felajánlja letöltésre.

localhost:38405/api/values

This XML file does not appear to have any style information associated with it. The document tree is sho

```
▼<ArrayOfstring xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="htt
   <string>István</string>
   <string>Viktória</string>
   <string>Béla</string>
   <string>Balázs</string>
   <string>Judit</string>
   <string>Miklós</string>
   <string>Ágnes</string>
   <string>Edina</string>
 </ArrayOfstring>
```

Lehet azonban paraméterezni is a GET kérést, ekkor:

http://localhost:38405/api/values/3

A lista negyedik elemét adja vissza (ugye nullától indexelünk).

Az eredményt alapértelmezés szerint XML-ben kaptuk vissza, de mi a teendő, ha mondjuk JSON-t szeretnénk, ami ugye tisztább, szárazabb érzés? Két lehetőségünk van: ha a végfelhasználóra bízzuk akkor ő a HTTP kérés felépítésekor beállíthatja az Accept vagy ContentType adatokat és tetszőleges eredményt kérhet vissza.

Ha alapértelmezett JSON a vágyunk akkor módosítsuk a WebApiConfig.cs fájlt:

```
public static void Register ( HttpConfiguration config )
{
    config.Routes.MapHttpRoute(
        name: "DefaultApi" ,
        routeTemplate: "api/{controller}/{id}" ,
        defaults: new { id = RouteParameter.Optional }
    );

    var appXmlType = config.Formatters
        .XmlFormatter
        .SupportedMediaTypes
        .FirstOrDefault(item => item.MediaType == "application/xml");
    config.Formatters.XmlFormatter.SupportedMediaTypes.Remove(appXmlType);

}
```

Kliens oldal

Az előző fejezetben eljutottunk odáig, hogy a szerver oldal szépen muzsikál és böngészőből le is tudjuk kérni az adatokat. Most megmutatom, hogy kliens oldalról hogyan tudunk kommunikálni a szolgáltatással.

Az első dolgunk, hogy takarítunk kicsit. A Web API sablon alapértelmezetten “teleszemeteli” az oldalt, szedjük ki a felesleget. A Views/Home/Index.cshtml fájlban töröljük ki a “body” id-vel rendelkező div elem tartalmát, és készen is vagyunk. A többi maradhat, megtartjuk a fejlécet és a szürke hátteret (marad egy fekete sáv az oldal tetején, CSS-t kell turkálni, ha valakit zavar). Helyezzünk el a div-be egy táblát, legyen az id-je dataTable:

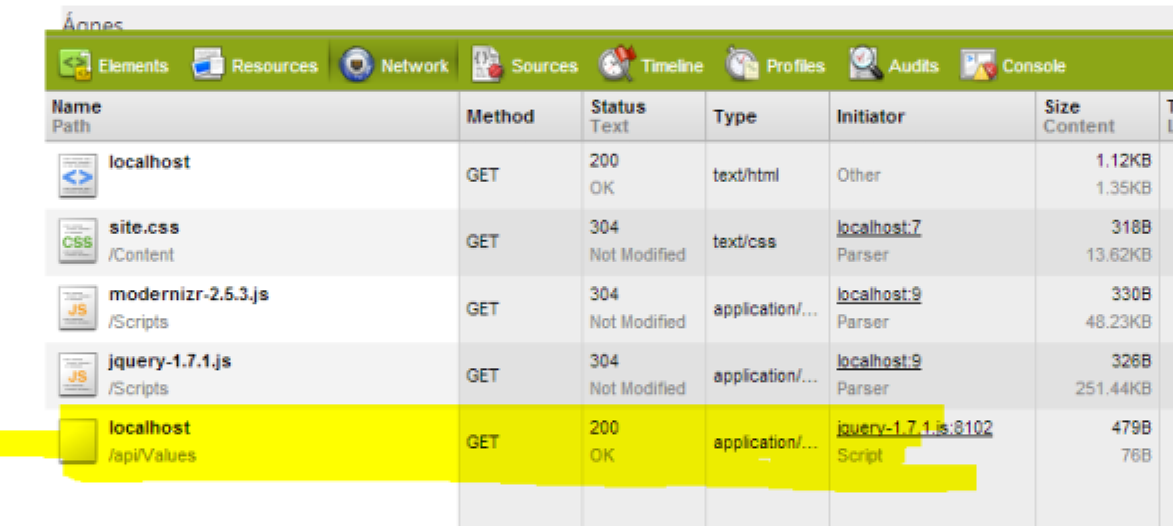
```
<header>
    <div class="content-wrapper">
        <div class="float-left">
            <p class="site-title">
                <a href="~/">ASP.NET Web API</a></p>
        </div>
    </div>
</header>
<div id="body" style="margin: 10px;">
    <table id="dataTable">

    </table>
</div>
```

Ezután navigáljunk át a Views/Shared/_Layout.cshtml fájlba és a *head* szekcióban helyezzünk el egy *script* elemet, amelyben lekérdezzük majd az adatokat. Itt a kód:

```
$(document).ready(function () {
    $.ajax({
        url: "api/Values/",
        dataType: "json",
        cache: false,
        statusCode:
            {
                200: function (response) {
                    $.each(response, function (key, value) {
                        $("#dataTable").append("<tr><td>" + value + "</td></tr>");
                    });
                },
                401: function () {
                    alert("Hiba");
                }
            }
    });
});
```

A jQuery *ajax* függényét használtuk, de természetesen tetszőleges javascript könyvtár is használható a kérés felépítéséhez. Az url-ben megadtuk a gyökeret (api/) és a szükséges controller nevét (Values, cask az előtag kell). Utána pedig, hogy milyen formátumban szeretnénk eredményhez jutni. Most a kérés, típusától függően (ha nem adunk meg mást, akkor ez egy GET lesz) a megfelelő controller akcióhoz kerül. Amennyiben sikeres GET kérést hajtottunk végre 200-as státuszkódot és az eredményt kapjuk vissza, ellenkező esetben 401 kód a jutalmunk.
Előbbi esetben már csak annyi a dolgunk, hogy végigmegyünk a listán és hozzáfűzzük az adatokat a táblához. Ez itt most elég fapados megoldás, de majd lesz jobb is.

Nézzük meg, hogy mi történt, én most a Chrome fejlesztői eszközeit használom. Alul látszik, hogy elment a kérés és vissza is adta a 200-as kódot:



Mutatom a kérés részleteit:

```
Headers  Preview  Response  Cookies  Timing

Request URL: http://localhost:38405/api/Values/?_=1357295237742
Request Method: GET
Status Code: 200 OK
▼Request Headers    view source
  Accept: application/json, text/javascript, */*; q=0.01
  Accept-Charset: ISO-8859-2,utf-8;q=0.7,*;q=0.3
  Accept-Encoding: gzip,deflate,sdch
  Accept-Language: hu-HU,hu;q=0.8,en-US;q=0.6,en;q=0.4
  Connection: keep-alive
  Cookie: szabalyzat=ok; Theme=Default
  Host: localhost:38405
  Referer: http://localhost:38405/
  User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.11 (K
  X-Requested-With: XMLHttpRequest
▼Query String Parameters    view URL encoded
  _: 1357295237742
▼Response Headers    view source
  Cache-Control: no-cache
  Content-Length: 76
  Content-Type: application/json; charset=utf-8
  Date: Fri, 04 Jan 2013 10:27:17 GMT
  Expires: -1
  Pragma: no-cache
```

És az eredményt:

```
Headers  Preview  Response  Cookies  Timing

▼[István, Viktória, Béla, Balázs, Judit, Miklós, Ágnes, Edina]
  0: "István"
  1: "Viktória"
  2: "Béla"
  3: "Balázs"
  4: "Judit"
  5: "Miklós"
  6: "Ágnes"
  7: "Edina"
```

Lekérdezéssel kész is lennénk, most nézzük meg, hogyan tudunk feltölteni adatokat. Nagyon hasonló lesz, mint amit eddig csináltunk, de igényel némi előkészületet. Elsősorban helyezzünk el az oldalon egy input mezőt és egy gombot, valahogy így:

```
<header>
    <div class="content-wrapper">
        <div class="float-left">
            <p class="site-title">
                <a href="~/">ASP.NET Web API</a></p>
        </div>
```

```
    </div>
</header>
<div id="body" style="margin: 10px;">
    <table id="dataTable">

    </table>

    <br />

    <input type="text" id="postText" />
    <button onclick="post();">POST</button>
</div>
```

Majd készítsük el a gomb által hivatkozott függvényt:

```
function post() {
    var postData = JSON.stringify($("#postText").val());

    $.ajax({
        url: "api/Values/",
        dataType: "json",
        cache: false,
        type: "POST",
        contentType: "application/json; charset=utf-8",
        data: postData,
        statusCode:
            {
                201: function (response) {
                    $("#dataTable").append("<tr><td>" + response + "</td></tr>");
                }
            }
    }).fail(function (xhr, textStatus, err) {
        alert(err);
    });
}
```

Három dolgot kell észrevenni:

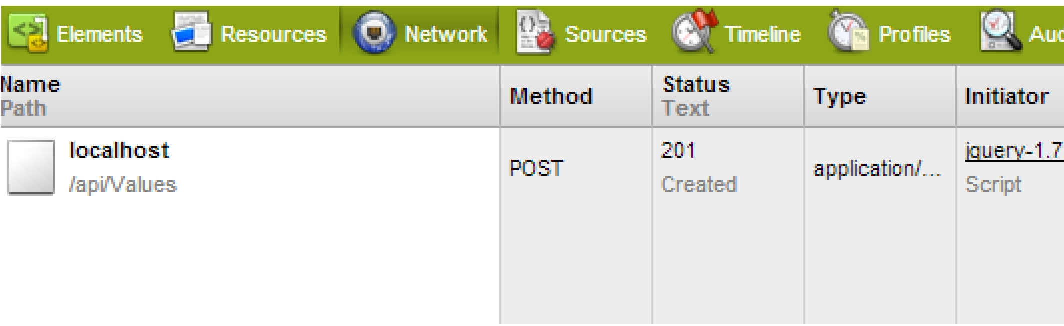
1. Átalakítottuk JSON formátumba a feltöltendő adatot
2. A kérés típusát explicite POST-ra állítottuk
3. A státuszkód ezúttal 201, vagyis "Created"

A kérésre kapott válaszban visszakapjuk a feltöltött elemet, amelyet hozzáfűzünk a listához. Ez a 201 kód sajátossága, azonban nem kötelező használni. Alapértelmezés szerint a Web API POST műveletei is 200-as kódot adnak vissza, ám ekkor más módot kell találni az új elem beillesztésére. A 201 azért is jó, mert így biztosan tudjuk, hogy az új adat rendben megérkezett. Van azonban egy spec. eset, amikor ugyan minden rendben van, de valamilyen oknál fogva még nem mentette el az alkalmazás. Ekkor lehet a 202-es kódot használni, ami az Accepted, vagyis eddig minden rendben, de még nincs minden kész. Viszont a 202 igényli az aktuális státusz megadását.

Kliens oldalon kész vagyunk, de szerveren is van dolgunk. Mivel nincs igazi adatbázisunk, ezért csak szimplán visszaküldjük a feltöltött stringet a státuszkód kíséretében:

```
// POST api/values
public HttpResponseMessage Post([FromBody]string value)
{
    return Request.CreateResponse<string>(HttpStatusCode.Created, value);
}
```

A visszatérési érték egy HttpResponseMessage objektum lesz, amelynek megadjuk, hogy milyen státuszt akarunk visszaküldeni, illetve a csatolt adatokat, ami ez esetben a feltöltött string. A metódus paraméterének típusa meg kell egyezzen azzal a típussal amit fel akarunk tölteni, illetve a CreateResponse-ra ugyanez igaz a generikus paraméterére vonatkozóan. Mutatom az eredményt, valóban ott a 201-es kód:



Egyszerű stringekkel már tudunk bánni, de mi a helyzet, ha ennél összetettebb adatszerkezeteket akarunk kezelni? Tulajdonképpen semmiféle helyzet nincs, a Web API gyönyörűen oda-vissza konvertál JSON-t, meg is mutatom.

Első lépésünk, hogy a Model mappában hozzunk létre egy Person osztályt, keresztnév, vezetéknév, ennyi:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Ezután a controller kódjában egy statikus listát készítünk, gyakorlatilag ugyanúgy, mint a stringes példában, még mindig nincs adatbázisunk ugye.

```
static public List<Person> Persons = new List<Person>()
{
    new Person() { FirstName = "Istvan", LastName = "Reiter" },
    new Person() { FirstName = "Frank", LastName = "Einstein" },
    new Person() { FirstName = "Sue", LastName = "Ellen" }
};
```

Írjuk át villámgyorsan a két GET és a POST függvényt, mindössze a típusokat kell kicserélni:

```
// GET api/values
public IEnumerable<Person> Get()
{
    return Persons;
}

// GET api/values/5
public Person Get(int id)
{
    return Persons[id];
}

// POST api/values
public HttpResponseMessage Post([FromBody]Person value)
{
    return Request.CreateResponse<Person>(HttpStatusCode.Created, value);
}
```

Kész vagyunk, teljesen fájdalommentes beavatkozás volt. Kliens oldalon hasonló nehézségekkel kell számolni, a GET kódja:

```
$(document).ready(function () {
    $.ajax({
        url: "api/Values/",
        dataType: "json",
        cache: false,
        statusCode:
            {
                200: function (response) {
                    $.each(response, function (key, value) {
                        $("#dataTable")
                            .append("<tr><td>"
                                + value.FirstName
                                + "</td><td>"
                                + value.LastName
                                + "</td><</tr>");
                    });
                },
                401: function () {
                    alert("Hiba");
                }
            }
    });
});
```

A válasz feldolgozásakor kell csak egy kis extra munkát elvégeznünk, micsoda szerencse, hogy a JSON stringek JavaScript alatt natív objektumok. Lefelé megvagyunk, nézzük a visszautat! Ehhez elsősorban egy újabb szövegmezőt kell elhelyeznünk az oldalon, hiszen most vezeték és keresztnevet küldünk fel. Az Index.cshtml a következőképpen alakul át:

```html
<header>
    <div class="content-wrapper">
        <div class="float-left">
            <p class="site-title">
                <a href="~/">ASP.NET Web API</a></p>
        </div>
    </div>
</header>
<div id="body" style="margin: 10px;">
    <table id="dataTable">

    </table>

    <br />

    <input type="text" id="firstNameText" /> <br />
    <input type="text" id="lastNameText" />
    <button onclick="post();">POST</button>
</div>
```

A Post függvényben pedig csak nevesítenünk kell az adatokat:

```javascript
function post() {
    var postData = JSON.stringify(
        { FirstName: $("#firstNameText").val(), LastName: $("#lastNameText").val() }
        );

    $.ajax({
        url: "api/Values/",
        dataType: "json",
        cache: false,
        type: "POST",
        contentType: "application/json; charset=utf-8",
        data: postData,
        statusCode:
            {
                201: function (response) {
                    $("#dataTable")
                        .append("<tr><td>" + response.FirstName + "</td><td>"
                            + response.LastName + "</td></tr>");
                }
            }
    }).fail(function (xhr, textStatus, err) {
        alert(err);
    });
}
```

Ígértem, hogy megnézzük a törlést is, egyelőre csak vázlatosan, de egy későbbi bejegyzésben rendesen kivesézem. Ha szerveroldalon megnézzük a Delete függvényt, akkor láthatjuk, hogy egy azonosítót vár, ami teljesen ésszerű, itt nem kell az egész objektumot feljuttatnunk. Persze adatbázis még mindig nincs, szóval a cselekmény maga ebben a cikkben értelmetlen. A törlés műveletet úgy indíthatjuk be, hogy az alábbi url-re egy DELETE HTTP kérést küldünk: api/controller_neve/azonosító. A kérés felépítése így néz ki, biztosan nem okoz meglepetést:

```
function del(key) {
    $.ajax({
        url: "api/Values/" + key,
        datatype: "json",
        cache: false,
        type: "DELETE",
        success: function () { alert("Deleted"); }
    }).fail(function (xhr, textStatus, err) {
        alert(err);
    });
}
```

Én itt most kényelmi okokból a success-t használtam a válasz kezelésére, de háromféle választ kaphatunk: 200 (OK) és 202 (Accepted), ezekről már volt szó a harmadik pedig a 204 (No Content), vagyis minden rendben, semmit nem kell visszaküldenem, a függvény visszatérési típusa pedig void.

Már csak azt kell kitalálni, hogy adjuk oda az azonosítót a kérésnek, ehhez kicsit módosítottam a GET függvényt:

```
$(document).ready(function () {
    $.ajax({
        url: "api/Values/",
        dataType: "json",
        cache: false,
        statusCode:
            {
                200: function (response) {
                    $.each(response, function (key, value) {
                        $("#dataTable")
                            .append("<tr><td>"
                                + value.FirstName
                                + "</td><td>"
                                + value.LastName
                                + "</td><td>"
                                + "<button onclick='del(" + key + ")' >DELETE</button>"
                                + "</td></tr>");
                    });
                },
                401: function () {
                    alert("Hiba");
                }
            }
    });
});
```

Ha megnyomjuk a gombot, akkor a kérés elmegy, de persze semmit nem csinál. Tartozom még a PUT vagyis módosítás függvénnyel, de úgy döntöttem, hogy a DELETE-hez hasonlóan ezt is csak röviden érintem. A PUT két részből áll: egyrészt egy azonosítóra van szükségünk, amely segítségével szerveroldalon megtaláljuk a szükséges objektumot, másrészt a módosított adatokat is át kell juttatni. Előbbi a DELETE-hez hasonlóan az url-ben, utóbbi a kérésbe ágyazva érkezik:

```
function update(key, firstName, lastName) {
    $.ajax({
        url: "api/Values/" + key,
        datatype: "json",
        cache: false,
        type: "PUT",
        data: JSON.stringify({FirstName: firstName, LastName: lastName}),
        success: function () { alert("Updated"); }
    }).fail(function (xhr, textStatus, err) {
        alert(err);
    });
}
```

Knockout

Kis kitérőt teszünk, abszolút mellőzve a Web API-t megismerkedünk egy igencsak hasznos Javascript könyvtárral: Knockout.
Amiért nagyon szeretjük őt az az, hogy lehetővé teszi adatkötések létrehozását, vagyis gyakorlatilag csak egy sablont kell megírnunk és a Knockout behelyettesíti az adatainkat. Ezzel nagyon leegyszerűsítettem a könyvtár tudását, megmutatom, hogy miről van szó.
A példánkban egy menü megjelenítő/karbantartó űrlapot fogunk létrehozni, hasonló bonyolultsággal, mint az eddigi részek példái. A kész program megtekinthető az alábbi linken:

http://dl.dropbox.com/u/6133419/index.html

Készítsük el először az oldal vázát, és linkeljük a Knockout illetve jQuery könyvtárakat:

```
<html>
    <head>
        <title>Knockout</title>

        <script type="text/javascript"
            src="http://knockoutjs.com/downloads/knockout-2.2.0.js"></script>
        <script type="text/javascript"
            src="http://code.jquery.com/jquery-1.8.3.min.js"></script>

        <script type="text/javascript">

        </script>
    </head>

    <body>

    </body>
</html>
```

A Knockout ún. viewmodel-ekre (és a Model-View-ViewModel mintára) épül, ami gyakorlatilag ugyanaz mint ami WPF/SL/… alatt van, bár közel sem kell annyit dolgozni rajta. Készítsük el első viewmodelünket:

```
function FoodViewModel(name, price) {
    this.name = name;
    this.price = price;
}
```

Egyszerű mint a faék, következő lépésünk, hogy megjelenítjük a viewmodel adatait az oldalon. A Knockout kihasználja, hogy a HTML5 megengedi tetszőleges attribútum kezelését a data-előtaggal. Ebben az esetben a data-bind attribútumot kell használnunk, amelyen keresztül megadjuk, hogy mit akarunk kötni, hová és hogyan. Ha az étel adatait akarom megjeleníteni, akkor az így fog kinézni:

```
Név: <strong data-bind="text: name"></strong> <br />
Ár: <strong data-bind="text: price"></strong>
```

A “text” lesz a hová, utána pedig a megfelelő viewmodel tulajdonság következik (vagyis a mit). Van már viewmodel, meg is tudjuk jeleníteni, már csak össze kell kötni ezt a kettőt. Az eddigi kód kiegészítve a Knockout beindításával:

```
<html>
    <head>
        <title>Knockout</title>

        <script type="text/javascript"
            src="http://knockoutjs.com/downloads/knockout-2.2.0.js"></script>
        <script type="text/javascript"
            src="http://code.jquery.com/jquery-1.8.3.min.js"></script>

        <script type="text/javascript">

            function FoodViewModel(name, price) {
                this.name = name;
                this.price = price;
            }

            $(document).ready(function () {
                var vm = new FoodViewModel("Rántott hús", 600);
                ko.applyBindings(vm);
            });

        </script>
    </head>

    <body>
        Név: <strong data-bind="text: name"></strong> <br />
        Ár: <strong data-bind="text: price"></strong>
    </body>
</html>
```

Első lépés kipipálva, lépjünk tovább. Természetesen módosítani is szeretnénk az adatokat, mi lenne, ha a viewmodel-t egy textbox-ra kötjük?

```
Név: <strong data-bind="text: name"></strong> <br />
Ár: <strong data-bind="text: price"></strong> <br />
<input type="text" data-bind="value: name" />
```

Megjelenik a szöveg, szerkeszteni is tudjuk, de jó lenne, ha az oldalon lévő összes kötés is változna. Ehhez ún. observable tulajdonságokat fogunk használni, amelyek “értesítik” a felhasználói felületet a változásokról. Írjuk át a viewmodel-t:

```
function FoodViewModel(name, price) {
    this.name = ko.observable(name);
    this.price = ko.observable(price);
}
```

Ha most átírjuk a szövegdoboz tartalmát és Enter-t nyomunk vagy elvesszük onnan a fókuszt akkor az oldalon lévő másik kötött adat is megváltozik.

Mivel menüt csinálunk, ezért egynél több viewmodel-t is tudnunk kell kezelni. Kézenfekvő, hogy készítünk egy új viewmodel-t és ez tárolja majd a többi objektumot:

```
function MenuViewModel() {
    var self = this;

    self.foods = ko.observableArray([
        new FoodViewModel("Rántott hús", 600),
        new FoodViewModel("Palacsinta", 800),
        new FoodViewModel("Csokitorta", 1000),
        new FoodViewModel("Almás pite", 1200)
    ]);
}
```

Az első sorban elkövetett “var self = this;” az ún. binding loss effektus ellen véd, ha valakit érdekel akkor látogasson el ide:

http://www.alistapart.com/articles/getoutbindingsituations

Máskülönben kell és kész. Az observableArray könnyen kitalálható, hogy a sima observable listásított változata, amely maga is viewmodel-eket tárol. Persze a Knockout inicializálása is változik kicsit:

```
$(document).ready(function () {
    var vm = new MenuViewModel();
    ko.applyBindings(vm);
});
```

Ok, most kössük rá a listánkat valamire. Ugye itt két kötés is lesz: egyszer magát a listát, másodszor a lista elemeit is külön kötnünk kell. Ehhez a foreach binding-et használjuk, aminek megadhatunk egy sablont, ami alapján a lista elemeit megjeleníti. Kell viszont még egy olyan HTML elem, amely képes egyszerre több elemet megjeleníteni, mi most táblázatot használunk, de pl. lista is megfelelő erre a célra.

```
<table>
    <thead>
        <tr>
            <th>Név</th><th>Ár</th>
        </tr>
    </thead>
    <tbody data-bind="foreach: foods">
        <tr>
            <td data-bind="text: name"></td>
            <td data-bind="text: price"></td>
        </tr>
    </tbody>
</table>
```

A táblázat törzsében beállítottuk a foreach kötést, hogy használja a viewmodel foods tulajdonságát, vagyis a listánkat.

A következő funkció a törlés lesz. Azt akarjuk, hogy a lista minden eleméhez tartozzon egy törlés gomb. Elsősorban készítsük el a törlés függvényt a viewmodel-ben, vagyis:

```
self.remove = function (food) {
    self.foods.remove(food);
};
```

Most pedig egészítsük ki a táblázat sablonunkat:

```
<tbody data-bind="foreach: foods">
    <tr>
        <td data-bind="text: name"></td>
        <td data-bind="text: price"></td>
        <td><button data-bind="click: $root.remove">Delete</button></td>
    </tr>
</tbody>
```

Az első amit észre kell venni, hogy a gomb click eseményéhez kötöttünk, vagyis egyúttal bevezettük magunkat a Knockout eseménykezelésének világába is. A másik érdekes dolog az az amit kötöttünk. Az ugye egyértelmű, hogy a frissen elkészített remove függvény lesz az áldozat, ám mit keres előtte a $root? Gyorsan gondoljuk át a dolgot! A táblázathoz hozzákötöttük a "fő" viewmodel egy tulajdonságát, viszont a táblázaton/sablonon belül egy új kötés van érvényben, mégpedig a listaelemé. Innen nem tudjuk elérni a MenuViewModel-t, ezért meg kell kérnünk a Knockout-ot, hogy a FoodViewModel helyett a gazda viewmodel-en belül keressen.

Már csak két napirendi pontról kell gondoskodnunk, az első a hozzáadás lesz. Kicsit talán szokatlan megoldást fogunk alkalmazni, ugyanis el akarjuk kerülni, hogy kézzel kelljen lekérni az új adatokat. Azt csináljuk, hogy a viewmodel-ben felveszünk egy változót, amely mindig az aktuális új elemet tárolja és ezt kötjük hozzá egy űrlaphoz. Amikor pedig hozzáadjuk a listához az új elemet, akkor "lenullázzuk" ezt az átmeneti változót, hogy újra munkába állhasson.

```
self.itemToAdd = ko.observable(new FoodViewModel("Étel neve", 0));
```

Persze observable lesz és adtunk egy kis segítséget is a felhasználónak. Készítsük el a hozzáadás függvényt is:

```
self.add = function () {
    self.foods.push(self.itemToAdd());
    self.itemToAdd(new FoodViewModel("Étel neve", 0));
};
```

Végül az űrlapot. Felhasználjuk, hogy a Knockout képes kezelni a submit eseményt. Az adatkötéseknél figyeljünk a zárójelekre!

```
<form data-bind="submit: add">
    <input type="text" data-bind="value: itemToAdd().name" /> <br />
    <input type="text" data-bind="value: itemToAdd().price" /> <br />
    <button type="submit">OK</button>
</form>
```

Utolsó lépés következik! Már csak valamiféle ellenőrzést szeretnénk az űrlapra rakni, hogy ne lehessen akármit felvinni. Legyenek, a feltételeink mondjuk: 1. az ár legyen nullánál nagyobb, 2. az étel neve ne legyen üres string, 3. az étel neve ne egyezzen meg azzal amit mi magunk adtunk meg. Ez önmagában egyáltalán nem bonyolult, viszont azt akarjuk, hogy ezeket a feltételeket a gombhoz köthessük, vagyis olyan mechanizmust kell találnunk, ami folyamatosan követni tudja a hozzáadásra kijelölt elem állapotát. Erre a célra ún. számított tulajdonságokat (computed property eredetiben, jobban is hangzik) használunk, a miénk ilyen lesz:

```
self.canAddNewItem = ko.computed(function () {
    return (
        self.itemToAdd().name() !== "" &&
        self.itemToAdd().name() !== "Étel neve" &&
        self.itemToAdd().price() > 0);
});
```

Mindössze annyi dolgunk maradt, hogy hozzákössük a gomb enable tulajdonságához:

```
<form data-bind="submit: add">
    <input type="text"
        data-bind="value: itemToAdd().name, valueUpdate: 'keyup'" />
    <br />
    <input type="text"
        data-bind="value: itemToAdd().price, valueUpdate: 'keyup'" />
    <br />
    <button type="submit"
        data-bind="enable: canAddNewItem">OK</button>
</form>
```

Egy aprósággal azért kiegészítettem: ha nem adunk meg mást, akkor a kötött elemek értéke akkor módosul a viewmodel-en belül, ha a vezérlőről elkerül a fókusz, vagy Enter-t ütünk. A “valueUpdate”-nek azonban megadhatjuk, hogy minden billentyűlenyomás után tegye ezt meg, így elegánsabb megoldást kapunk.

Knockout + Web API

Ismerjük a Web API-t, ismerjük a Knockout-ot, ideje összehozni a kettőt! Azt a projektet fogjuk használni, amit a harmadik részig összeraktunk. Első dolgunk, hogy adjuk hozzá az oldal kódjához a Knockout könyvtárat. Szerencsénk van, mert a Web API sablon alapértelmezésben belerakja a Script mappába, mindössze hozzá kell adnunk a bundle-höz. Ez az MVC 4 újítása, ami lehetővé teszi, hogy összecsomagoljuk a szükséges Javascript könyvtárakat. Nyissuk meg az App_Start mappában lévő BundleConfig.cs fájlt és a RegisterBundles függvényen belül a jQuery és társai után illesszük be az alábbi sort:

```
bundles.Add(new ScriptBundle("~/bundles/knockout").Include(
    "~/Scripts/knockout-*"));
```

Ezután a _Layout.cshtml fájlban a többi definíció után írhatjuk:

```
@Scripts.Render("~/bundles/knockout");
```

Most készítsük el a viewmodel vázát! A Person osztályunkat fogjuk használni szerveroldalon, ezért ilyen adatszerkezetre kell felkészülnünk. Adjunk hozzá a projekthez egy ClientScripts nevű könyvtárat, azon belül pedig egy Client.js nevű fájlt. Ezek persze tetszés szerint elnevezhetőek. A Javascript fájlban lesz az összes kód, ami egyelőre a következő:

```
function PersonViewModel(firstName, lastName) {
    this.firstName = ko.observable(firstName);
    this.lastName = ko.observable(lastName);
}

function PersonsViewModel() {
    var self = this;

    self.itemToAdd = ko.observable(new PersonViewModel("", ""));
    self.persons = ko.observableArray();

    self.get = function () { };

    self.add = function () { };

    self.delete = function (person) { };

    self.update = function (person) { };
}
```

Az előző fejezet után semmi újdonságot nem jelenthet. Most a GET és POST függvényeket készítjük el, következő alkalommal pedig jön az UPDATE és DELETE. Kezdjük a get függvényünkkel:

```
self.get = function () {
    self.persons.removeAll();

    $.ajax({
        url: "api/Values/",
        datatype: "json",
        cache: false,
        statusCode:
            {
                200: function (data) {
                    $.each(data, function (key, value) {
                        self.persons.push(
                            new PersonViewModel(value.firstName, value.lastName)
                            );
                    });
                }
            }
    });
};
```

Két dolgot kell észrevenni: az első, hogy az ajax hívás előtt kitöröltük a tömb tartalmát. Ez első ránézésre kissé értelmetlen, viszont később hasznosnak fog bizonyulni. Vegyük például a törlést, megkérjük a szerveroldalt, hogy törölje az adatbázisból az adott elemet, ám ezt a kliens oldalára is le kell képezni. Vagy manuálisan törlünk, viszont ekkor nincs biztosítékunk, hogy a kliens is ugyanazokat az adatokat birtokolja, vagy meghívjuk a get függvényt, ekkor pedig pontos adataink lesznek. Ilyenkor viszont értelemszerűen ki kell üríteni az aktuális elemeket a listából.

A másik érdekesség, hogy bejövő objektumok kisbetűs tulajdonságait használtuk. Ez elsőre banálisnak tűnhet, ám van értelme. Ha emlékeztek, akkor akkor eddig a szerveroldalról nagybetűs tulajdonságok érkeztek, tehát LastName, etc… Viszont, azt figyelembe kell venni, hogy a Javascript és a C# elnevezési konvenciói különböznek: JS esetén tulajdonságok és adattagok kisbetűvel kezdődnek, C# nyelven nagybetűvel. Vagy keresünk középutat és kizárólagosan az egyik formát használjuk, rábírjuk – mondjuk – a szerveroldalt, hogy alkalmazkodjon a kliens igényeihez. Persze nem szeretnénk átírni a létező osztályainkat sem, ezért azt a valamit kell megcéloznunk ami a kettő között van: a JSON-t.

A .NET lehetővé teszi, hogy megadjuk a JSON formátumot a JsonProperty attribútum segítségével, tehát nem kell mást tenni, mint kidekorálni a kérdéses osztályokat (kell a Newtonsoft.Json névtér):

```
public class Person
{
    [JsonProperty("firstName")]
    public string FirstName { get; set; }

    [JsonProperty("lastName")]
    public string LastName { get; set; }
}
```

Persze ha csak annyi hasznunk lenne az egészből, hogy a lekérés könnyebb akkor nem lenne sok értelme, viszont van ám más is a tarsolyomban. Jöjjön a POST függvény:

```
self.add = function () {
    var person = ko.toJSON(self.itemToAdd());

    $.ajax({
        url: "api/Values/",
        datatype: "json",
        contentType: "application/json; charset=utf-8",
        type: "POST",
        cache: false,
        data: person,
        statusCode:
            {
                201: function (data) {
                    self.persons.push(
                        new PersonViewModel(data.firstName, data.lastName)
                        );
                    self.itemToAdd(new PersonViewModel("", ""));
                }
            }
    });
};
```

Rögtön az első sorban ott van a lényeg. Ahelyett, hogy bűvészkedni kellene a JSON-nal a kisbetű/nagybetű miatt egyszerűen megkérem a Knockout motort, hogy csináljon egy JSON objektumot az éppen hozzáadott elemből, átküldöm a szervernek, ami pedig szó nélkül kezeli.

Folytassuk a DELETE és PUT műveletekkel! Meglehetősen kevés dolgunk lesz, első lépésben egészítsük ki az oldalunk HTML kódját, azaz adjuk hozzá a megjelenített elemekhez a törlés és mentés gombokat. Azt mondtam mentés, vagyis valamiféleképpen tudnunk kell módosítani az adatokat. Az eddigi “sima” megjelenítést kicseréltem, input elemeket fogunk használni. A végleges HTML tehát:

```
<header>
    <div class="content-wrapper">
        <div class="float-left">
            <p class="site-title">
                <a href="~/">ASP.NET Web API</a></p>
        </div>
    </div>
</header>
<div id="body" style="margin: 10px;">
        <table>
            <thead>
                <tr>
                    <th>First Name</th><th>Last Name</th>
                </tr>
            </thead>
            <tbody data-bind="foreach: persons">
                <tr>
                    <td><input type="text" data-bind="value: firstName" /></td>
```

```
                    <td><input type="text" data-bind="value: lastName" /></td>
                    <td><button data-bind="click: $root.remove">Delete</button></td>
                    <td><button data-bind="click: $root.update">Save</button></td>
                </tr>
            </tbody>
        </table>

    <br />

    <form data-bind="submit: add">
        <input type="text"
            data-bind="value: itemToAdd().firstName, valueUpdate: 'keyUp'" /> <br />
        <input type="text"
            data-bind="value: itemToAdd().lastName, valueUpdate: 'keyUp'" />
        <button type="submit">OK</button>
    </form>
</div>
```

Kezdjük a törléssel! Szerveroldalon lesz egy kis dolgunk, a forráskód:

```
// DELETE api/values/5
public void Delete(int id)
{
    Persons.RemoveAt(id);
}
```

A DELETE vár egy azonosítót az url részeként. Amikor a nagyon egyszerű példát kitaláltam nem gondoltam a jövőre, így a Person osztályunknak nincsen egyértelmű azonosítója. Még. Amikor a GET műveletet kezeltük emlékezzünk, hogy két értéket kapunk minden listaelemhez: a data tárolta az objektumot, a key pedig tulajdonképpen egy index, ami jelöli a listabeli pozíciót. Egészítsük ki gyorsan a PersonViewModel-t:

```
function PersonViewModel(firstName, lastName, id) {
    this.firstName = ko.observable(firstName);
    this.lastName = ko.observable(lastName);
    this.id = ko.observable(id);
}
```

És a Get függvényt is:

```
self.get = function () {
    self.persons.removeAll();

    $.ajax({
        url: "api/Values/",
        datatype: "json",
        cache: false,
        statusCode:
            {
                200: function (data) {
                    $.each(data, function (key, value) {
                        self.persons.push(
                            new PersonViewModel(value.firstName, value.lastName, key)
                            );
                    });
                }
            }
    });
};
```

A Delete paraméterének típusa lehet egész számtól különböző, ám a neve mindenképpen “id” kell maradjon, mert így van beállítva a routing (átírható persze, de nem éri meg a fáradtságot).
A kliens oldali kód:

```
self.remove = function (person) {
    $.ajax({
        url: "api/Values/" + person.id(),
        type: "DELETE",
        cache: false,
        success: self.get
    });
};
```

Lényegesen egyszerűbb az eddigieknél, mindössze az url-hez kell hozzáfűzni az azonosítót, illetve a sikeres művelet után megadni a végrehajtandó műveletet. A DELETE alapbeállítás szerint 204-es státuszkódot ad vissza és nem küld semmilyen adatot.
Jöjjön a PUT! Ismét szerveroldal:

```
// PUT api/values/5
public void Put(int id, [FromBody]Person value)
{
    Persons[id] = value;
}
```

A PUT szintén azonosítót vár az url-ben, illetve fel kell küldenünk a megváltoztatott objektumot is. A kliens oldali kód nem jelenthet problémát:

```
self.update = function (person) {
    $.ajax({
        url: "api/Values/" + person.id(),
        datatype: "json",
        contentType: "application/json; charset=utf-8",
        type: "PUT",
        cache: false,
        data: ko.toJSON(person),
        success: self.get
    });
};
```

Authentikáció

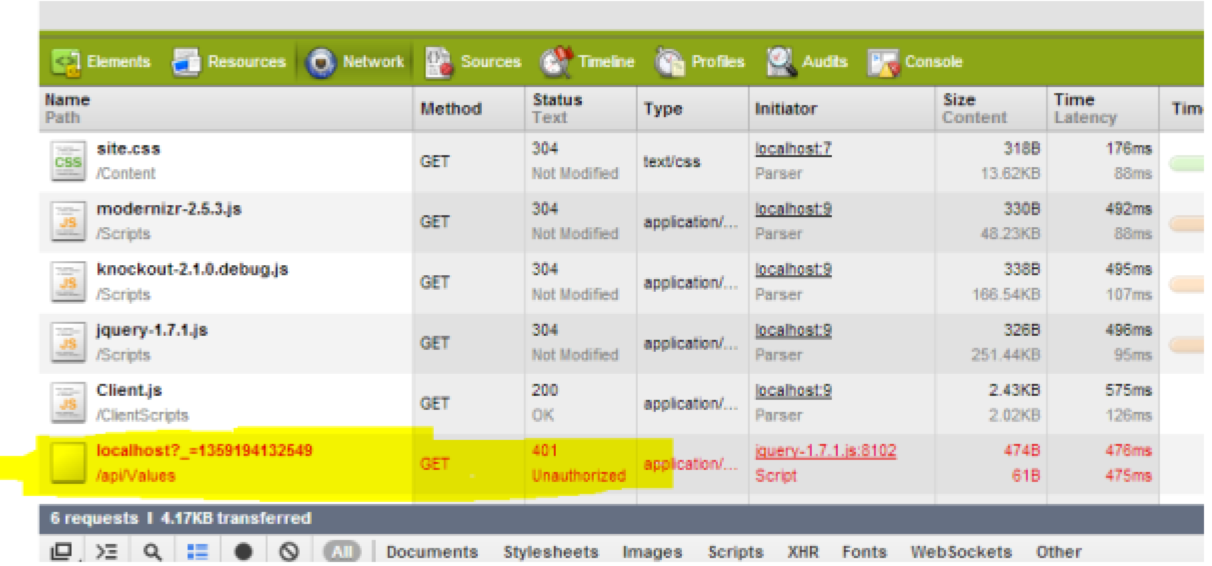
Tegyük fel, hogy nem szeretnénk, hogy a személy-listánkat akárki láthassa. Az ugye nyilvánvaló, hogy ilyenkor valamiféle beléptetős okosságot kellene használni, de milyet? Azt tudni kell, hogy az MVC 4 vadiúj authentikációs csomagot kapott, ezt fogjuk beüzemelni.
Na most, van egy apró probléma. A sima Web API projektsablonban nincsen előre beállítva a felhasználóazonosításhoz szükséges infrastruktúra, így két lehetőségünk van:

a) Megírjuk kézzel az egészet
b) Web API sablon helyett az MVC Internet Application sablon-t használjuk, amihez minden további nélkül lehet Web API controller-eket adni.

Én a b. lehetőséget ajánlom, kicsit dolgozni kell rajta, de lényegesen egyszerűbb mint az a. opció.
Ha minden kész, akkor egyetlen dolgot kell tennünk: ellátni a megfelelő műveleteket vagy akár magát a controller-t az Authorize attribútummal:

```
[Authorize]
public class ValuesController : ApiController
{
    static public List<Person> Persons = new List<Person>()
    {
```

Ezután ha megpróbáljuk meghívni, mondjuk a GET műveletet 401-es státuszkódot kapunk vissza, azaz nincs jogosultságunk.

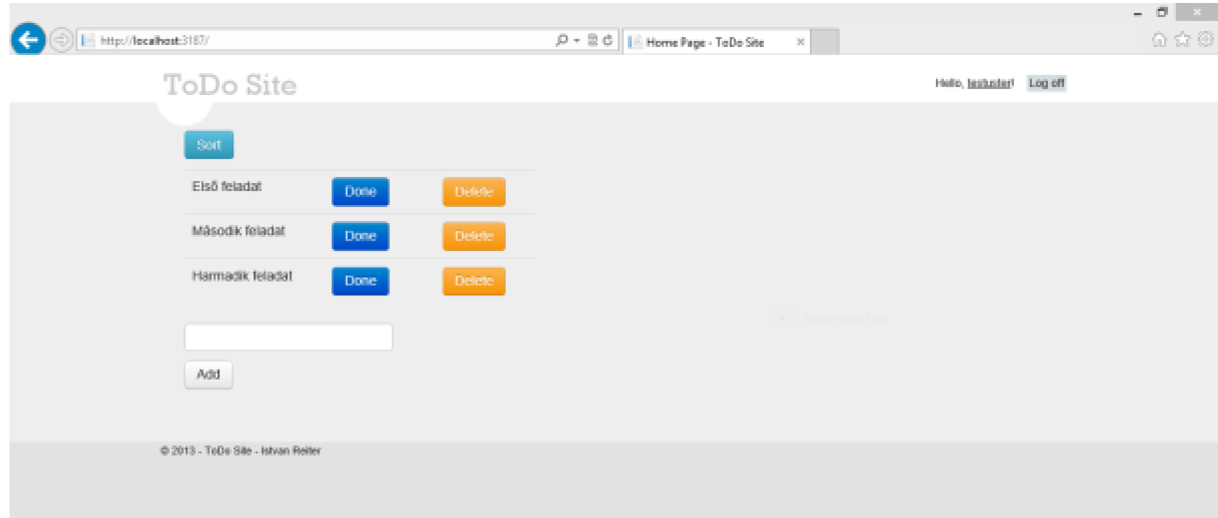


Ezt a helyzetet könnyen megoldhatjuk, mindössze a 401-es kódot kell kezelni a “normális” státusz mellett és átdobni vész esetén a felhasználót a beléptető oldalra, valahogy így:

```
self.get = function () {
    self.persons.removeAll();

    $.ajax({
        url: "api/Values/",
        datatype: "json",
        cache: false,
        statusCode:
            {
                200: function (data) {
                    $.each(data, function (key, value) {
                        self.persons.push(
                            new PersonViewModel(value.firstName, value.lastName, key)
                            );
                    });
                },
                401: function () {
                    window.location = "/Account/Login/";
                }
            }
    });
};
```

ToDoSite

Itt az ideje, hogy eddigi tudásunkat felhasználva elkészítsünk egy rendes alkalmazást is, ez nem más mint a ToDoSite. A nevéből gyorsan ki lehet találni, hogy egy “feladatlistáról” van szó, semmi feleslegesen bonyolult. Tulajdonképpen a program külseje meglehetősen puritán, de azért lesz munka vele bőven.
A kész oldal így néz majd ki:



Néhány szót a felhasznált technológiákról:

Az adatelérésért az Entity Framework Code First felelős. A szerver-kliens kommunikáció a Web API-n keresztül történik, jQuery AJAX hívásokkal. Ezeket az adatokat a Knockout Javascript könyvtárral kezeljük majd. A megjelenítésért ez, illetve a Twitter Bootstrap szintén Javascript könyvtár lesz felelős. Végül, de nem utolsósorban az egész fejlesztés egy ASP.NET MVC 4 projekt keretein belül történik.

A fejlesztéshez Visual Studio 2012 Ultimate-et használok, de a Visual Studio 2012 Express for Web is tökéletesen megteszi. Érdemes (kötelező) telepíteni hozzá az ASP.NET and Web Tools 2012.2 csomagot, ami többek között Intellisense támogatást ad a Knockout adatkötéseihez.

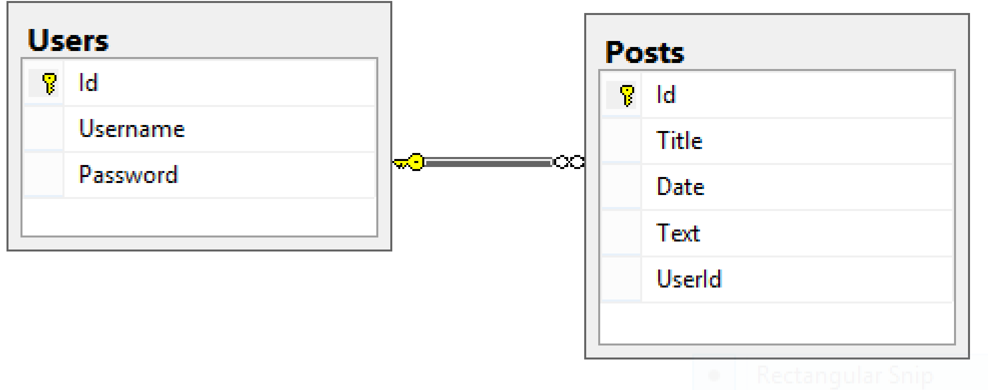
A Visual Studio és a Web Tools telepíthető a Web Platform Installer-ből, a könyvtárakat pedig vagy kapjuk a projekt sablonban, vagy Visual Studio-ból letölthetőek a NuGet segítségével.
A kész alkalmazás forráskódja megtekinthető itt:

https://github.com/reiteristvan/ToDoSite/

Entity Framework Code First bevezető

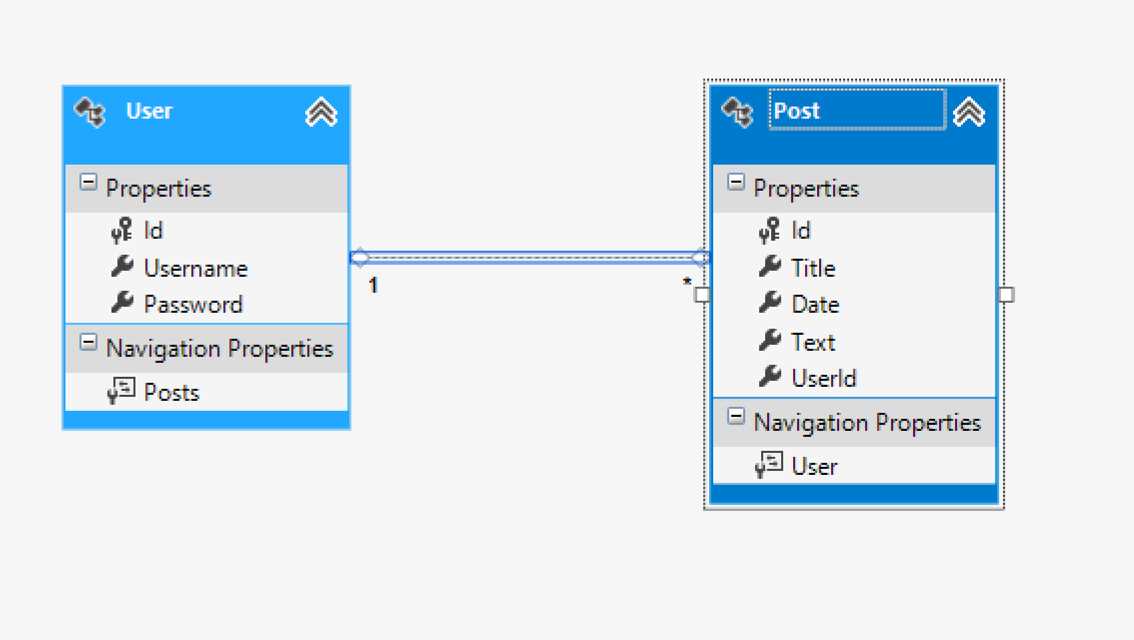
Alulról felfelé építkezünk, tehát először az adatkezelés kérdését oldjuk meg. Amire szükség van a bejegyzés megértéséhez az egy Visual Studio 2012, és egy SQL Server (Express) ami fut és ismerjük az azonosítóját (ez alapértelmezés szerint vagy SQLSERVER vagy SQLEXPRESS).
Ezeken kívül kell még a legfrissebb Entity Framework csomag, ehhez Tools menü, majd Library Package Manager és Manage NuGet Packages for Solution, itt keressük meg az EntityFramework nevű csomagot és telepítsük.

Az Entity Framework egy ún. ORM (Object-Relational Mapping) eszköz, amely lehetővé teszi, hogy egy adatbázis tábláit C# osztályokká képezzünk le. Ez tök jól hangzik, de mi a fenét jelent? Nézzünk egy példát, legyen ez egy blogmotorféleség adatbázisa. A következő néhány fejezetben ezt az adatbázist használjuk kísérleti nyúlként. Van két táblánk, felhasználók és bejegyzések, minden bejegyzéshez tartozik egy darab felhasználó:



A C# egy objektumorientált nyelv, akár használunk ORM eszközt, akár nem, kívánatos, hogy ezeket az adatokat egyszer csak egy osztályban tároljuk. Elkészítjük az osztályt, megírjuk a lekérdezéseket, a visszakapott adatokat feldolgozzuk, jó kis favágó munka. Mi lenne, ha ezt az egészet – és akár többet is – megoldhatnánk 4 kattintással? Itt jön képbe az EF, ami nemhogy elvégzi a fent leírtakat helyettünk, de még van hozzá varázslós felület is. Egyelőre nézzük a “nem Code First” megközelítést.

Projekten jobb klikk -> Add New Item -> ADO.NET Entity Data Model. Nincs más dolgunk mint végigkattintgatni, megadni az adatbázis elérését, kijelölni a kívánt táblákat és:



Csodálatosan rusnya, bár a VS 2010 libafos színű undormányánál egy fokkal jobb. Nem is ez a lényeg, amit itt látunk azok tulajdonképpen az osztályok, a táblák oszlopaiból property-k lettek, a kapcsolatokat is így érjük el. Ahhoz, hogy tudjunk ezekkel az osztályokkal dolgozni szükségünk egy közvetítő objektumra is, amely az adatbázis műveletekért lesz felelős, illetve nyomon követi az objektumok változásait. Nézzük meg, hogyan tudjuk lekérdezni, mondjuk az összes felhasználót:

```
using (DatabaseEntities context = new DatabaseEntities())
{
    var result = context.Users;
}
```

Egyszerű, mint a faék. A DatabaseEntities osztály a közvetítő, jó hülye nevet adtam neki, de hát késő bánat. Mi van, ha szeretném elkérni az adott felhasználóhoz tartozó bejegyzéseket? Először le kell kérdeznem őt felhasználónév alapján, majd a navigációs property eljuttat az eredményhez:

```
using (DatabaseEntities context = new DatabaseEntities())
{
    var user = context.Users.SingleOrDefault((item) => item.Username == "ireiter");
    var posts = user.Posts;
}
```

Semmi SQL, semmi tárolt eljárás, tisztán csodaszép C# forráskód. A SingleOrDefault egy, a paramétereként megadott feltételnek megfelelő elemet ad vissza, illetve ha ilyen nincs akkor null értéket. Később még jobban is belemegyek a témába, prezentációs célra megfelelt.

Na most, ha valaki ért angolul akkor feltűnhet, hogy a címben “Code First” van, vagyis először forráskód. Ez meg mi a haltüdőt jelenthet? Amit most csináltunk, azt egy létező adatbázisból generált modellen tettük. Ehhez viszont már el kellett készíteni az adatbázist, ami valljuk be megintcsak fárasztó meló, főleg, hogy utána nem is használjuk a hagyományos értelemben. Szerencsére az Entity Framework ennek a fordítottját is megengedi: osztályokból generál adatbázist.
Megjegyzem, hogy az Entity Framework 4.x és 5 között jelentős különbségek vannak, az egyik ilyen, hogy az adatbázisból generált kód egy az egyben megegyezik azzal amit a Code First esetén írnánk. Illetve vissza lehet térni a régi (alapértelmezett) stratégiára is (Modellen jobb klikk -> Properties -> Code Generation Strategy legyen Default), ha valakinek jobban tetszik az ObjectContext-es katyvasz.
Lényeg a lényegben, jobb klikk mondjuk valamelyik DatabaseEntities (vagy aminek épp elneveztük) definíción és Go to Declaration, ez előhozza a generált kódját a köztes osztálynak, nálam így néz ki:

```
//------------------------------------------------------------------------------
// <auto-generated>
//    This code was generated from a template.
//
//    Manual changes to this file may cause unexpected behavior in your application.
//    Manual changes to this file will be overwritten if the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace CodeFirst
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;

    public partial class DatabaseEntities : DbContext
    {
        public DatabaseEntities()
            : base("name=DatabaseEntities")
        {
        }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            throw new UnintentionalCodeFirstException();
        }

        public DbSet<Post> Posts { get; set; }
        public DbSet<User> Users { get; set; }
    }
}
```

Kettő dolog érdekel minket, az egyik a DbContext. Ez az osztály az eredeti ObjectContext-ből származik, minden Code First közvetítő osztály ebből fog örökölni. A másik, amit látni kell, azok a DbSet deklarációk, az adatbázis minden egyes táblája egy ilyen DbSet deklarációt kap,

tulajdonképpen lehet egyfajta lista adatszerkezetnek tekinteni. Látogassunk el most a Post osztály deklarációjához:

```
//------------------------------------------------------------------------------
// <auto-generated>
//    This code was generated from a template.
//
//    Manual changes to this file may cause unexpected behavior in your application.
//    Manual changes to this file will be overwritten if the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace CodeFirst
{
    using System;
    using System.Collections.Generic;

    public partial class Post
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public DateTime Date { get; set; }
        public string Text { get; set; }
        public int UserId { get; set; }

        public virtual User User { get; set; }
    }
}
```

Látható, hogy az adott tábla minden oszlopának egy-egy property felel meg, leképezve őket a szükséges típusokkal. Az ntext/varchar/nchar oszlopok mindig string típusúak lesznek. Ugyanígy a DateTime és DateTime2 is a .NET féle DateTime típussá alakul. Tipp: ha létező adatbázisból készítünk modellt, a dátum típusú oszlopok mindig DateTime2 típust kapjanak, az EF azt szereti jobban. Végül alul ott a navigációs property, amelyet minden esetben virtuálisnak kell jelölni (ld.: http://msdn.microsoft.com/en-us/library/dd468057).

Ugorjunk még át a User osztálydefinícióhoz is ahol nagyjából ugyanaz a kép fogad, de mivel egy felhasználóhoz több bejegyzés tartozik, ezért itt a navigációs property egy listaszerű valami kell legyen, vagyis valósítsa meg az ICollection interfészt. Ebben az esetben ez egy HashSet lesz ami kiválóan alkalmas olyan elemhalmazok tárolására amelyek egyediek illetve nem fontos a sorrend. Lehet másféle adatszerkezetet is használni, a HashSet teljesítmény szempontjából jó választás.

```
//------------------------------------------------------------------------------
// <auto-generated>
//    This code was generated from a template.
//
//    Manual changes to this file may cause unexpected behavior in your application.
//    Manual changes to this file will be overwritten if the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------
```

```
namespace CodeFirst
{
    using System;
    using System.Collections.Generic;

    public partial class User
    {
        public User()
        {
            this.Posts = new HashSet<Post>();
        }

        public int Id { get; set; }
        public string Username { get; set; }
        public string Password { get; set; }

        public virtual ICollection<Post> Posts { get; set; }
    }
}
```

Ha fognánk ezt a három osztályt, és elhelyezzük őket a projektben, azonnal készen is lennénk, de akkor ugye kevésbé szórakoznánk jól. Írjuk meg őket még egyszer, kiegészítve egy kicsit. A “tábla-osztályokat” és a közvetítő osztályt tegyük nyugodtan egyetlen forrásfájlba, ne foglalják a helyet:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CodeFirst.Model
{
    public partial class BlogContext : DbContext
    {
        public BlogContext ()
            : base( "MyConnection" ) {
            //Configuration.LazyLoadingEnabled = true;
        }

        public DbSet<Post> Posts { get; set; }
        public DbSet<User> Users { get; set; }
    }

    public partial class Post
    {
        [Key]
        [DatabaseGenerated( DatabaseGeneratedOption.Identity )]
        public int Id { get; set; }

        public string Title { get; set; }
        public DateTime Date { get; set; }
```

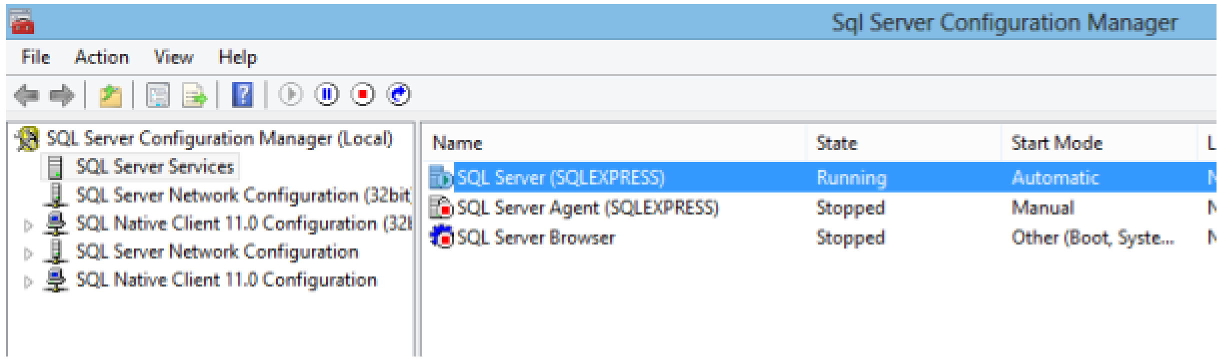
```
        public string Text { get; set; }

        public virtual User User { get; set; }
    }

    public partial class User
    {
        public User () {
            this.Posts = new HashSet<Post>();
            Roles = new HashSet<Role>();
        }

        [Key]
        [DatabaseGenerated( DatabaseGeneratedOption.Identity )]
        public int Id { get; set; }

        public string Username { get; set; }
        public string Password { get; set; }

        public virtual ICollection<Post> Posts { get; set; }
        public virtual ICollection<Role> Roles { get; set; }
    }

    public class Role
    {
        public Role () {
            Users = new HashSet<User>();
        }

        [Key]
        public String RoleName { get; set; }

        public virtual ICollection<User> Users { get; set; }
    }
}
```

A táblák elsődleges kulcsain elhelyeztünk két attribútumot. Ezekkel kijelöljük az elsődleges kulcsot, illet megadjuk, hogy az elsődleges kulcs értékét automatikusan növelje.
A BlogContext osztály a saját ősének (DbContext) átadja az adatbázis connectionstring-jét. Mivel most még nincs létező adatbázis, ezért ezt nekünk kell megadni az App.Config vagy Web.Config fájlban. Ha nincs ilyen a projektben akkor adjunk hozzá egy Application Configuration File-t. A configuration szekción belül írjuk a következőt:

```
<connectionStrings>
  <add name="MyConnection"
        connectionString="server=.\sqlexpress;database=BlogDatabase;Integrated
Security=yes;"
        providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Ebben az esetben egy SQL Express adatbázismotort használunk, a server értékének a futó példány nevét kell megadni, erre utaltam a bejegyzés elején. Ha nem tudjátok a nevét, akkor az

SQL Server Configuration Manager-ben (Start menüben megtalálható) meg lehet nézni, ez mindenképp települ az összes SQL Server változattal:

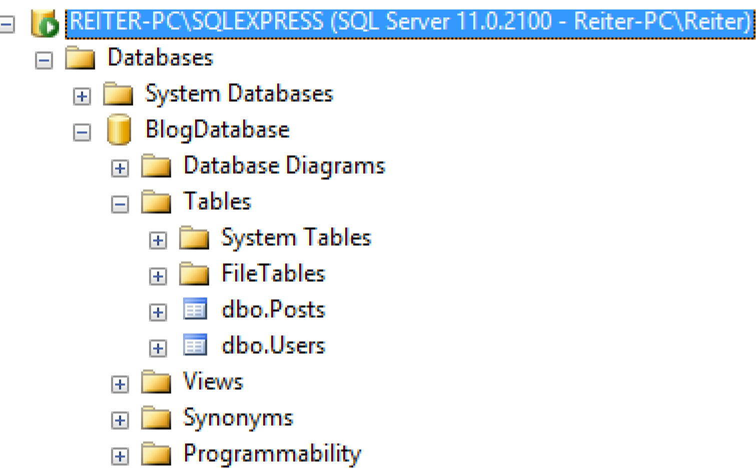


Vissza a connectionstring-hez: a database értékének az adatbázis nevét kell megadni, tetszőleges.

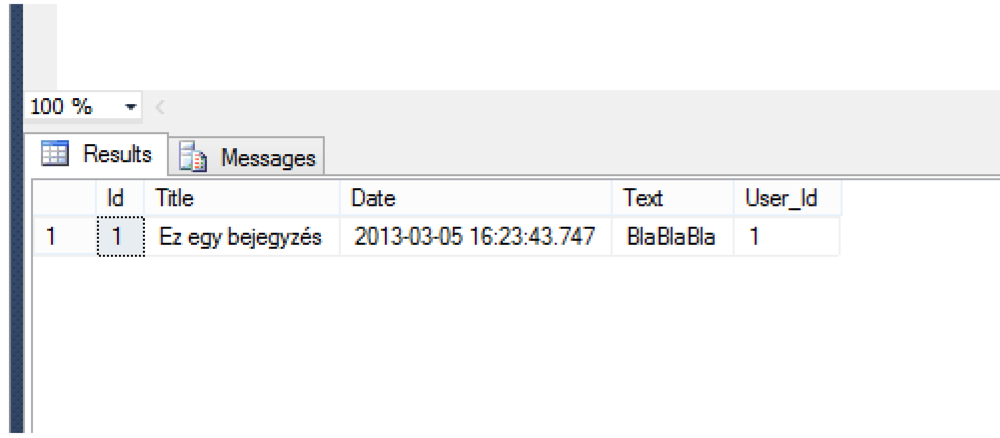
Most már készen is vagyunk, próbáljuk ki a művünket! Az alábbi kódrészlettel beszúrunk egy felhasználót és egy bejegyzést, majd futtatunk egy lekérdezést:

```
using (BlogContext context = new BlogContext())
{
    User user = new User()
    {
        Username = "ireiter",
        Password = "password",
    };

    Post post = new Post()
    {
        Title = "Ez egy bejegyzés",
        Date = DateTime.Now,
        Text = "BlaBlaBla",
    };

    context.Users.Add(user);
    user.Posts.Add(post);

    context.SaveChanges();

    var result = context.Posts.SingleOrDefault(
        (item) => item.User.Username == "ireiter");

    Console.WriteLine(result.Text);
}
```

A közvetítő osztályon meghívott SaveChanges lesz ami minden változást elment az adatbázisba. Nyissunk is gyorsan egy Management Studio-t, lássuk tényleg ott vannak-e az adatok:



Adatbázis megvan, kérjük le mondjuk a Posts tábla adatait:



Abszolúte tökéletes, ügyesek voltunk. Látható, hogy a Code First egy meglehetősen erőteljes eszköz, rengeteg favágó munkát meg tudunk vele spórolni.

A Repository minta

Amikor adatbázissal dolgozunk gyakorlatilag minden amit teszünk visszavezethető négy alapműveletre. Ezek az ún. CRUD műveletek: Create/Beszúrás, Retrieve/Kiválasztás, Update/Módosítás, Delete/Törlés. A Code First modell ugyan hozzáférhetővé teszi ezeket nekünk, de megspórolhatunk némi favágó munkát azzal, ha építünk fölé még egy réteget.A Repository tervezési minta pont erre szolgál, az alapműveleteket kivezethetjük rajta, a részleteket pedig elrejtjük. Ez ugye jól hangzik, de ha belegondolunk az Entity Framework maga is ezt teszi, tehát technikailag egy újabb réteget építünk egy már létező rétegre. Mindjárt foglalkozunk még ezzel a témakörrel, most inkább nézzünk meg egy tényleges Repository implementációt.

Az újrafelhasználás érdekében érdemes absztrakt osztályt, de főleg interfészt használni. Az alap repository interfész így néz ki nálunk:

```
public interface IRepository<T>
{
    IEnumerable<T> SelectAll();
    IEnumerable<T> Select(Expression<Func<T, bool>> filter);
    void Insert(T entity);
    void Update(T entity);
    void Delete(object id);
}
```

A négy alapművelet mellé bevettük a csapatba a szűrést is. A generikus paraméter típusa a repository által kezelt típus lesz, tehát az általunk használt példában ez a Post vagy a User osztályok egyike.

A Repository tipikusan az a minta, amit a végtelenségig lehet cifrázni, beépíthetünk rendezést, lapozás-támogatást, akármit amit a kis szívünk megkíván. Mi most megmaradunk az alapoknál, a UserRepository osztály teljes valójában, magyarázat a forráskód után:

```
public class UserRepository : IRepository<User>
{
    public IEnumerable<User> SelectAll()
    {
        using (BlogContext context = new BlogContext())
        {
            return context.Users.ToList();
        }
    } // End SelectAll

    public IEnumerable<User> Select(Expression<Func<User, bool>> filter)
    {
        using (BlogContext context = new BlogContext())
        {
            return context.Users.Where(filter).ToList();
        }
```

```
    } // End Select

    public void Insert(User entity)
    {
        using (BlogContext context = new BlogContext())
        {
            context.Users.Add(entity);
            context.SaveChanges();
        }
    } // End Insert

    public void Update(User entity)
    {
        using (BlogContext context = new BlogContext())
        {
            var task = context.Users.SingleOrDefault((item) => item.Id == entity.Id);

            if (task == null)
            {
                throw new ArgumentException();
            }

            context.Entry(task).CurrentValues.SetValues(entity);

            context.SaveChanges();
        }
    } // End Update

    public void Delete(object id)
    {
        using (BlogContext context = new BlogContext())
        {
            var task = context.Users.SingleOrDefault((item) => item.Id == (int)id);

            if (task == null)
            {
                throw new ArgumentException();
            }

            context.Users.Remove(task);
            context.SaveChanges();
        }
    } // End Delete
}
```

A két Select elég egyértelmű, viszont lehetne még őket csiszolni. IEnumerable-t adunk vissza, mert az IRepository interfészben ilyet követeltünk meg. Viszont így nem teljesen jó a megvalósítás, hiszen az Entity Framework egy másik interfészt jobban szeret: IQueryable. Na most, van egy kis csavar! Az EF lekérdezések akkor jutnak el az adatbázishoz, amikor szükség van rá, vagy ha kikényszerítjük. A Where például úgy működik, hogy lekérhetjük ugyan vele a szűrésnek megfelelő listát, de csak akkor fog lefutni ténylegesen a lekérdezés, ha a lista elemeit használjuk is:

```
var result = akármi.Where(feltétel); // még nem ment el a lekérdezés

foreach ( item in result ) { // MOST indul

}
```

Kényszeríteni persze lehet, ebben az esetben a ToList egy ilyen kényszerítő erő, hiszen az adatbázisból lekérdezett elemeket egy listába tesszük. Ehhez természetesen hozzá kell nyúlni az adatbázisszerverhez. Itt jön a képbe az IQueryable, ami lehetővé teszi, hogy ezt a számunkra rendkívül kívánatos viselkedést (ld. deferred execution ~ elhalasztott végrehajtás) megőrizzük. Írjuk át egy kicsit a Select-eket:

```
public IQueryable<User> SelectAll()
{
    using (BlogContext context = new BlogContext())
    {
        return context.Users;
    }
} // End SelectAll

public IQueryable<User> Select(Expression<Func<User, bool>> filter)
{
    return SelectAll().Where(filter);
} // End Select
```

A rossz hír: az IRepository definíciónkban le kell cserélni az IEnumerable-t IQueryable visszatérési típusra. A SelectAll-ból így csak lekapjuk a ToList-et és készen is vagyunk. A szűrős Select már érdekesebb, ott a SelectAll-t hívtuk és ahhoz csaptuk hozzá a Where kitételt. Ezt megtehetjük, mert a SelectAll nem nyúl az adatbázishoz amíg nem használjuk ténylegesen az eredménylistát, így a Where majd szépen beépül a végső lekérdezésbe és egyetlen körben szed le mindent. Persze van itt egy kis gond. Ha ebben a formában akarjuk használni, akkor a szűrésre kivételt kapunk:

The operation cannot be completed because the DbContext has been disposed

Mivel a SelectAll-ban using blokkot használtunk, ezért a DbContext osztályunk a return után feldobja a talpát. Nekünk viszont egy aktív osztályra van szükségünk, különben a Where nem tud "beépülni". Több lehetőségünk is van: a.: nem használjuk a SelectAll-t, de így több kódot kell írni, ráadásul ez odaver a kódújrafelhasználásnak is, b.: az IRepository leszármazottunk használjon egyetlen DbContext-et minden lekérdezéshez.
Utóbbit fogjuk használni, és ha már itt tartunk egy újabb dilemmával nézünk szembe: adjuk-e meg az IRepository interfészben, hogy tartalmazzon az implementáció egy DbContext példányt? Ha igen, akkor az IRepository már nem lesz univerzális, hiszen hozzákötöm a Code First/EF pároshoz. Megtehetem persze, hogy kiszervezem ezt az egyetlen property-t egy külön interfészbe, de cask emiatt nem tűnik okos megoldásnak.
Azért még nincs vége: feltesszük, hogy több repository megvalósításunk is lesz, amelyek mindegyikének tartalmaznia kell egy DbContext leszármazottat, ami a mi esetünkben a BlogContext lesz. Ez azt jelenti, hogy lesz két-három-négy repository implementációnk,

amelyeknek van egy közös pontja. Készítsünk talán egy absztrakt osztályt? De akkor hol jön képbe az IRepository? Hiszen annak meg kell adni a tárolt típust, ez az információ az ősosztályban nyilván nem áll rendelkezésre. Látható, hogy csodálatosan bele lehet ám bonyolódni az egész objektum-orientált marhaságba. Sokféle megoldás létezik, mindegyikhez fel tudjuk sorolni az előnyöket és hátrányokat gondolkodás nélkül.
A legszebb pedig, hogy az egész azzal kezdődött, hogy meg akartunk spórolni két sort a using blokkok elhagyásával. Tucatnyi lehetőség van, a repository implementáció használhat absztrakt osztályt a DbContext-hez plusz mellé behozhatom az eredeti IRepository-t, felparaméterezhetem az absztrakt osztályt, használhatok reflection-t, és így tovább…
Lényeg a lényegben, a kód-újrafelhasználás elmehet a ráktüdőbe, azt az egy sort majd szépen odaírjuk magunknak. Vegyünk hát fel egy adattagot a repository megvalósításunkba:

```
private BlogContext context = new BlogContext();
```

Még egy kis finomításra szükség lesz: a BlogContext példányunkat illik tisztességesen kinyírni, ha már nincs szükség rá, valósítsa meg az osztályunk az IDisposable interfészt is. Tehát, a végső megvalósításunk:

```
public class UserRepository : IDisposable, IRepository<User>
{
    public IQueryable<User> SelectAll () {
        using ( BlogContext context = new BlogContext() ) {
            return context.Users;
        }
    } // End SelectAll

    public IQueryable<User> Select ( Expression<Func<User , bool>> filter ) {
        return SelectAll().Where( filter );
    } // End Select

    public void Insert(User entity)
    {
        context.Users.Add(entity);
        context.SaveChanges();
    } // End Insert

    public void Update(User entity)
    {
        var task = context.Users.SingleOrDefault((item) => item.Id == entity.Id);

        if (task == null)
        {
            throw new ArgumentException();
        }

        context.Entry(task).CurrentValues.SetValues(entity);

        context.SaveChanges();
    } // End Update

    public void Delete(object id)
    {
```

```
        var task = context.Users.SingleOrDefault((item) => item.Id == (int)id);

        if (task == null)
        {
            throw new ArgumentException();
        }

        context.Users.Remove(task);
        context.SaveChanges();
    } // End Delete

    // Properties

    private BlogContext context = new BlogContext();

    // IDisposable interface

    public void Dispose()
    {
        context.Dispose();
    }
}
```

Kérdés, hogy mire jutottunk ezzel? Megéri-e ennyit szenvedni csak azért, hogy olyan funkciókat használjunk, amik amúgy is készen vannak. Hiszen nem csinálunk mást, mint ráépítünk az EF-re egy réteget, hogy az adatok felhasználásakor öt sor helyett csak egyet kelljen írni. Tulajdonképpen nem. A probléma az, hogy alapműveletekről van szó, semmi extra üzleti logika, semmi olyan körülmény, ami indokolná, hogy bevezessünk egy plusz osztályt.

Alapvetően a Repository mintával nincsen semmi baj, sőt kiválóan alkalmas arra, hogy az alkalmazásunk adatelérési rétegét jól elkülönítse az üzleti logikától. Viszont nem szabad megfeledkeznünk a KISS elvről sem: Keep It Simple Stupid, vagyis törekednünk kell az egyszerüségre is. Ha a szoftver nem elég komplex, nem biztos, hogy jó ötlet ennyit trükközni, egyszerű megoldásokkal hamarabb célt érünk. Mérlegelni kell, hogy érdemes-e plusz időt áldozni olyan programrészre, aminek nem vesszük különösebb hasznát. Ha a programunk semmi mást nem csinál, csak néhány alapműveletet végez az adatbázison akkor valószínűleg a Repository bevezetése felesleges. Ha ennél többről van szó, akkor mindenképpen jó ötlet használni, sőt olyankor már érdemes közelebbről megismerkedni a UnitOfWork mintával is.

Összességében a Repository hasznos minta, de az általunk készített alkalmazás nem veszi hasznát. Bármilyen vicces, de maga a Repository megvalósítás több programsort tartalmazna, mint a program többi része.

Entity Framework műveletek

Ebben a fejezetben nagyon egyszerű dolgom lesz. Mivel a LINQ családot úgy alkották meg, hogy a lekérdezések gyakorlatilag módosítás nélkül fussanak minden környezetben, ezért aki járatos a LINQ To Objects-ben az tud adatbázisokra is lekérdezéseket írni. Persze sok olyan dolog is van, ami máshogy működik, de az alapok ugyanazok. Ajánlom tehát a saját könyvemet: C# programozás lépésről lépésre. Az alábbi linkről ingyenesen letölthető, illetve papírformában a legtöb könyvesboltban megvásárolható.

http://bit.ly/17itoSS

Nézzük hát meg a speciális eseteket, ez most a törlés és módosítás témakörét jelenti. A “régi” Entity Framework óta sok minden változott, a DbContext nagyjából mindent elrejt előlünk. Az egyik ilyen dolog az entitások változásainak követése. Mivel a Code First osztályokat mi magunk írjuk, ezért az állapotot csak a DbContext segítségével kérdezhetjük le:

```
BlogContext context = new BlogContext();

var user = context.Users.Single((item) => item.Username == "ireiter");
var state = context.Entry(user).State;

Console.WriteLine(state); // unchanged
```

A State property az EntityState enumeráció értékeit veheti fel:

- **Unchanged** : az entitás originál kiadású, senki nem nyúlt hozzá VAGY már hívtunk az utolsó módosítás óta SaveChanges-t
- **Added** : az entitást hozzáadtuk a listához, de még nem hívtunk SaveChanges-t
- **Deleted** : az entitást eltávolítottuk a listából, de még nem hívtunk SaveChanges-t
- **Modified** : az entitás értékei megváltoztak, de még nem hívtunk SaveChanges-t
- **Detached** : az entitást törültük ÉS hívtunk SaveChanges-t vagy kézzel leválasztottuk a listáról, a Detached állapotú entitások változásait NEM követi az EF

Ami fontos, hogy ezek az állapotok a SaveChanges előtt érvényesek, abban a pillanatban, ahogy meghívtuk az entitások vagy Unchanged vagy Detached állapotot vesznek föl. Az alábbi forráskód egy kis demonstráció, hogy hogyan működik ez az egész élesben:

```
BlogContext context = new BlogContext();

var user = context.Users.Single((item) => item.Username == "ireiter");
var state = context.Entry(user).State;

Console.WriteLine(state); // unchanged

var user2 = new User()
{
```

```
    Username = "ireiter2",
    Password = "almaspite",
};

context.Users.Add(user2);

Console.WriteLine(context.Entry(user2).State); // added

context.SaveChanges();

Console.WriteLine(context.Entry(user2).State); // unchanged

context.Users.Remove(user2);

Console.WriteLine(context.Entry(user2).State); // deleted

context.SaveChanges();

Console.WriteLine(context.Entry(user2).State); // detached
```

Ugye mondanom sem kell, hogy a SaveChanges után nincs visszaút, vagyis majdnem, ugyanis tranzakciókat támogat az EF. Ehhez vagy a TransactionScope osztályt, vagy a DbContext Database property-jén található Connection-ön belüli BeginTransaction metódust kell használni. Ebbe a témába most nem megyek bele, a lényeg, hogy ha egy intergalaktikus moly megrágja a műveleteket, akkor van lehetőség visszavonni azokat.

Kicsit elkalandoztam, térjünk vissza az eredeti vágányra! A törlés elég egyértelmű ezek után, a módosításnál viszont nem árt gondolkodni. Ha megnézzük a repository megvalósításunkban az Update műveletet akkor látni fogjuk, hogy paraméterként egy User entitást kap. Ne felejtsük el, hogy ez a sorozat egy kliens/szerver felépítésű program készítéséről szól, vagyis lekérjük az entitásokat, elküldjük őket a kliensnek, ahonnan valamilyen állapotban visszakapjuk őket. Ilyenkor persze ezek az entitások lecsatolódnak a DbContext-ről, nem követi – nem is tudná – a változásokat. Ezért a munkamenet a következő lesz: lekérjük azonosító alapján az entitást ami az adatbázisból jön (tehát gyakorlatilag a bejövő entitás az ennek a klónja, csak más adatokkal) és felülírjuk az értékeit. Ez jól hangzik, viszont ha belegondolunk akkor simán előfordulhat, hogy 8-10 propertyt kellene átírni. Nyilván ezt csak egyszer kellene megrajzolni, de a programozó lusta, ezért utánanéz, hogy van-e egyszerűbb megoldás. Van, így néz ki:

```
var entity = new User() { /* ... */};
var user = context.Users.Single((item) => item.Username == "ireiter");

context.Entry(user).CurrentValues.SetValues(entity);
```

Első sorban az elképzelt megváltoztatott entitás, a másodikban lekérjük az adatbázisból a mi változatunkat, végül a harmadik sorban felülírjuk. Az Entry metódus kapja a sajátunkat, a SetValues-nek pedig az újat kell átadni. Persze ez a módszer akkor hasznos, ha az entitásokat lecsatoljuk és/vagy fizikailag máshol változtatjuk meg, egyébként követi őket az EF.

ToDoSite – Model

Eljött végre az ideje, hogy érdemi munkát is végezzünk. A New Project menüponttal kezdünk, és egy ASP.NET MVC 4 Web Application-t kell készítenünk. A sablon kiválasztása utáni ablakban az Internet Aplication ikonra csapjunk le. Hiába Web API alapú a projekt, a Web API sablonnal dolgozni kell, hogy hozzáadjuk a felhasználókezelést. Ezzel szemben az Internet Application sablon már fel van készítve a Web API-ra, elég csak az App_Start mappába belenézni. Erre a témára még visszatérünk, most más dolgunk lesz.
Elsősorban állítsuk be az adatbázis elérést. Egy adatbázist már használ a projekt, méghozzá az authentikációs szolgáltatás, ami egyébként Code First alapon működik. Dönthetünk úgy, hogy ehhez rakjuk hozzá a saját dolgainkat, de az átláthatóság kedvéért most használjunk egy különálló adatbázist. A Web.config fájl tetején a connectionStrings szekcióban meg is találjuk a létező kapcsolatot DefaultConnection néven, amely az App_Data mappában lévő adatbázisfájlra mutat. Mivel Code First-ről van szó, ezért a projekt első indításáig, illetve az első adatbázis felé irányuló lekérdezésig ez a fájl nem fog létezni, utána aspnet-Projektneve-Dátum.mdf néven megjelenik. A Visual Studio-ban nem látszik, a Solution Explorerben a Show All Files gombra kattintva előbukkan.
A saját adatbázisunkhoz nyugodtan lemásolhatjuk ezt, átírva a fájlnevet, illetve a kapcsolat nevét, én azonban egy Sql Compact adatbázisfájlt fogok használni, az ehhez tartozó connectionString így fog kinézni (ToDoConnection):

```
  <connectionStrings>
    <add name="DefaultConnection" connectionString="Data Source=(LocalDb)\v11.0;Initial 
Catalog=aspnet-ToDoSite-20121031140333;Integrated 
Security=SSPI;AttachDBFilename=|DataDirectory|\aspnet-ToDoSite-20121031140333.mdf" 
providerName="System.Data.SqlClient" />

    <add name="ToDoConnection" connectionString="Data 
Source=|DataDirectory|\ToDoDB.sdf;Persist Security Info=False;" 
providerName="System.Data.SqlServerCe.4.0" />
  </connectionStrings>
```

A |DataDirectory| mutat az App_Data mappára, ahová hagyományosan az ASP.NET (MVC) adatbázisok kerülnek.

Az előző fejezetekben vagy 1500 szót áldoztam az Entity Framework-re, az igazság azonban az, hogy a programunk adatbázis tekintetében fakocka egyszerűségű lesz, mindössze egyetlen táblát fog tartalmazni. Sok szó nem esett még az alkalmazás tényleges funkciójáról, pótoljuk hát be. Tulajdonképpen arról van szó, hogy a felhasználók felvihetnek egy adott tevékenységet, amelyet lehet törölni, illetve elvégzettnek jelölni. Az adatbázisban így az elsődleges kulcson (numerikus, automatikusan növekszik) kívül a következőkre van szükség: tevékenység megnevezése, a hozzáadás dátuma, állapot, illetve a felhasználó neve. Ez utóbbinál egyszerűsítettem kicsit, hiszen megtehettem volna, hogy a saját tábláimat hozzákapcsolom a kész infrastruktúrához és “rendes” relációval kötöm össze a felhasználók tábláit a a tevékenységekével. Viszont, azt tudom, hogy minden felhasználónak egyedi

felhasználóneve van, így a fent vázolt megoldásra nincs szükség, egyértelműen tudom azonosítani a felhasználókat a választott nevük alapján.

A táblám tehát így fog kinézni (Models\ToDoTask.cs):

```
[Table("Task")]
public class ToDoTask
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    [JsonProperty("id")]
    public int Id { get; set; }

    [Required]
    [JsonProperty("description")]
    public string Description { get; set; }

    [JsonProperty("date")]
    [JsonConverter(typeof(JsDateConverter))]
    public DateTime Date { get; set; }

    [JsonProperty("isCompleted")]
    public bool IsCompleted { get; set; }

    [JsonProperty("userName")]
    public string UserName { get; set; }
}
```

A Code First-ös cuccokon kívül csak a JsonProperty attribútum van jelen, ugye kliens oldalon könnyebb az élet ha betartjuk a Javascript elnevezési konvencióit. Ezeken kívűl még egy JsonConverter attribútumot tettem a Date property fölé. Ezzel majd egy későbbi fejezetben foglalkozunk, egyelőre nyugodtan ki lehet kommentezni, hiszen még nem írtuk meg a hivatkozott osztályt.
Fogom a táblámat, bekötöm egy DbContext leszármazottba (Models\ToDoContext.cs):

```
public class ToDoContext : DbContext
{
    public ToDoContext()
        : base("ToDoConnection")
    {

    }

    public DbSet<ToDoTask> Tasks { get; set; }
}
```

A konstruktorban a Web.config-ban elhelyezett connectionstringre hivatkoztam. Van a Models mappában még egy fájl (ToDoModelInitializer.cs) ami ismeretlen funkciót tölt be. Lehetőségünk van inicializálni az adatbázisunk és ez a fájl pont erre való. Amennyiben az osztályunk megvalósítja az IDatabaseInitializer interfészt rendben is vagyunk. Vannak ám előre gyártott elemek, mi itt a DropCreateDatabaseAlways osztályból származtatunk, ami megvalósítja az

előbb említett interfészt, emellett minden indításkor törli majd újrakreálja az adatbázist. Generikus paramétereként a szükséges DbContext származékot kell megadnunk. A Seed metódus felülírásával akcióba is léphetünk:

```
public class ToDoModelInitializer : DropCreateDatabaseAlways<ToDoContext>
{
    protected override void Seed(ToDoContext context)
    {
        ToDoTask task1 = new ToDoTask()
        {
            Description = "Első feladat",
            Date = DateTime.Now,
            IsCompleted = false,
            UserName = "testuster"
        };

        ToDoTask task2 = new ToDoTask()
        {
            Description = "Második feladat",
            Date = DateTime.Now,
            IsCompleted = false,
            UserName = "testuster"
        };

        ToDoTask task3 = new ToDoTask()
        {
            Description = "Harmadik feladat",
            Date = DateTime.Now,
            IsCompleted = false,
            UserName = "testuster"
        };

        context.Tasks.Add(task1);
        context.Tasks.Add(task2);
        context.Tasks.Add(task3);

        context.SaveChanges();
    }
}
```

A “testuster” felesleges ‘t’-je teljesen véletlen, így maradt. Ebben az esetben persze csak tesztelésre használjuk, hiszen nem tudhatjuk, hogy milyen felhasználónevet választ majd a felhasználó. Volt is egy vicces kis kavarodás az alkalmazás készítésekor, ugye én meg voltam róla győződve, hogy “testuser” név szerepel az adatbázisban és borzalmasan csodálkoztam, hogy miért nem jelennek meg az előre felvitt adatok. Még egy teendő, a Global.asax fájlban regisztrálni is kell ezt az osztályt, mindössze egy sorra van szükségünk az Application_Start metódusban:

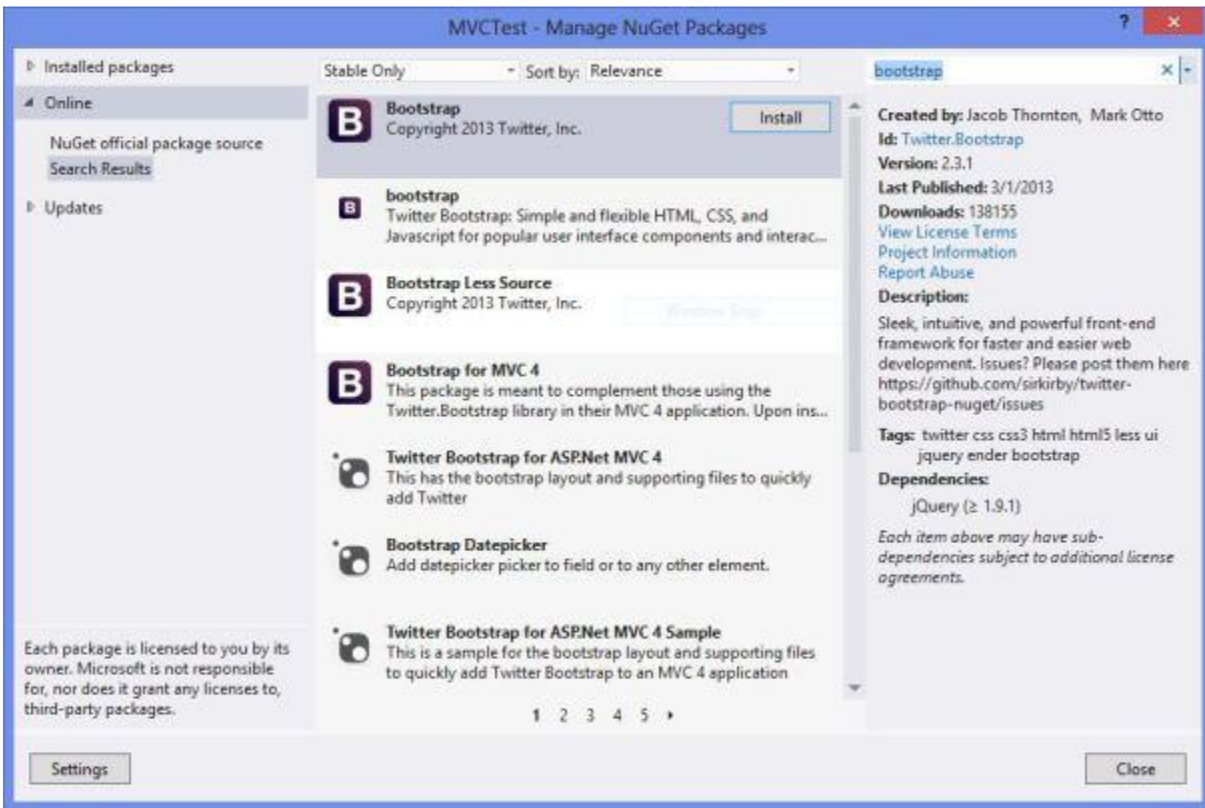
```
Database.SetInitializer<ToDoContext>(new ToDoModelInitializer());
```

Twitter Bootstrap

Újabb közjáték, ezúttal a Twitter Bootstrap könyvtár rejtelmeiben mélyedünk el. A Bootstrap egy Javascript/CSS könyvtár, amely elsősorban design szempontjából segít rajtunk (reszponzív grid layout, hű de jól hangzik), illetve tartalmaz jó pár izgalmas komponenst is.
A nevéből ki lehet találni, hogy a Twitteres lányok és fiúk rakták össze és meglepő módon maga a Twitter is ezt használja. Meg a NASA, StumbleUpon, Squedoo, etc… A csomag a hivatalos GitHub oldalról elérhető:

http://twitter.github.com/bootstrap/

Használatához a sima vagy minifikált .js és .css fájlokat kell linkelnünk, de persze a jó öreg Visual Studio ebben is tud nekünk segíteni. Az MVC sablonban alapértelmezés szerint nincs benne, de ezen könnyen segíthetünk, mivel NuGet csomag formájában fél perc alatt lekaphatjuk. Jobb klikk a projekten, Manage NuGet Packages, majd a keresőbe írjuk be, hogy “bootstrap” első helyen megérkezik:



Install-ra kattintva frissíti a jQuery könyvtárat is, mivel legalább 1.9-es verzió szükséges hozzá (a legújabb verzióhoz, korábbiak is elérhetőek, ha valamiért nem opció a jQuery upgrade). Ha végzett szépen beépül a Script illetve Content mappákba. A következő lépésben tudatnunk kell az oldallal, hogy szeretnénk használni ezt a könyvtárat. Természetesen lehetőség van

a hagyományos módszerre is (ld. fent), de egyszerűbb ha az új bundling-es megoldást alkalmazzuk. Ehhez nyissuk meg az App_Start mappában lévő BundleConfig.cs nevű fájlt, a tartalmát átnézve elég könnyű kitalálni mit kell írni. A ScriptBundle-t használó sorok alá/fölé/mellé szúrjuk be az alábbi sort:

```
bundles.Add(new ScriptBundle("~/bundles/bootstrap").Include(
            "~/Scripts/bootstrap.*"));
```

A StyleBundle-k közé pedig ezt:

```
bundles.Add(new StyleBundle("~/Content/bootstrap").Include(
            "~/Content/bootstrap.css",
            "~/Content/bootstrap-responsive.css"
    ));
```

Értelemszerűen ha reszponzív designt szeretnénk, akkor a bootstrap.responsive.css fájlra lesz szükségünk. Oké, egy dolgunk maradt, meg kell mondanunk az oldalnak, hogy használja is a Bootstrap-et. Látogassunk el a Views\Shared_Layout.cshtml nevű fájlhoz, majd helyezzük el a stílusra hivatkozó bundle-t a létező Style.Render alá:

```
@Styles.Render("~/Content/bootstrap")
```

Most jön a Javascript, ez trükkösebb lesz. A jQuery bundle az oldal kódjának alján van, a Bootstrap kódot mindenképpen ez után kell elhelyezni. Vagy tegyünk mindent egy helyre az oldal tetejére, vagy ott helyben másoljuk be, ízlés kérdése, de ha nem jó a sorrend az oldal hibát fog dobni. A szükséges sor:

```
@Scripts.Render("~/bundles/bootstrap")
```

Kész vagyunk. Az a szerencsés helyzet állt elő, hogy a Bootstrap oldalon nagyon részletes és nagyon szájbarágós dokumentációt lelhetünk, ezért csak azt fogom megmutatni amivel ténylegesen dolgozunk is, minden másért irány a doksi.

ToDoSite – View

A Bootstrap által üzemeltetett grid rendszer 940 pixel széles és 12 oszlopra osztható fel. Lehetőség van reszponzív módra is, ekkor ha a megjelenítő felület szélessége 767 pixel alá csökken az oszlopok függőlegesen helyezkednek majd el.

Az egyes sorokat a “row” osztály segítéségével jelöljük ki, az oszlopokat pedig a “span*” használatával, ahol “*” a felhasznált oszlopok számát jelenti. Az egy sorban lévő “span*”-ok 12 oszlopot kell felhasználjanak, vagy ha az őket tartalmazó konténer maga is egy oszlop, akkor az ő szélességének megfelelő értéknek kell kijönnie. Nézzünk egy gyakorlati példát, adott egy sor, amelyet egy 4 illetve 8 oszlopos szekcióra bontunk fel (egyenesen a Bootstrap doksiból):

```
<div class="row">
  <div class="span4">...</div>
  <div class="span8">...</div>
</div>
```

Amikor pedig beágyazunk oszlopokat egy oszlopba:

```
<div class="row">
  <div class="span3">...</div>
  <div class="span9">
    Level 1
    <div class="row">
      <div class="span6">Level 2</div>
      <div class="span3">Level 2</div>
    </div>
  </div>
</div>
```

A második oszlop 9 egységnyi, ezért a benne lévő “aloszlopoknak” is ennyit kell elfoglalniuk. Meglehetősen egyszerű dologról van szó, térjünk is rá az alkalmazásunkra. Kettő darab szekciónk lesz, az egyik megjeleníti a feladatok listáját, illetve a hozzájuk tartozó törlés és “kész” gombokat. A másik segítségével tudunk új feladatot hozzáadni a listához, itt egy szövegdobozra és egy gombra lesz szükségünk.

A Views\Shared_Layout.cshtml fájlban kell elhelyeznünk a külső könyvtárak hivatkozásait, és a saját kódunkra mutató linket is, így néz ki:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>@ViewBag.Title - ToDo Site</title>
        <link href="~/favicon.ico" rel="shortcut icon" type="image/x-icon" />
        <meta name="viewport" content="width=device-width" />
        @Styles.Render("~/Content/css")
        @Styles.Render("~/Content/bootstrap")
```

```
        @Scripts.Render("~/bundles/modernizr")
        @Scripts.Render("~/bundles/jquery")
        @Scripts.Render("~/bundles/bootstrap")
        @Scripts.Render("~/bundles/knockout")
        @RenderSection("scripts", required: false)

        <script type="text/javascript" src="~/ClientScripts/ToDoViewModel.js"></script>

        <script type="text/javascript">
            var viewmodel = new ToDoViewModel();

            $(document).ready(function () {
                ko.applyBindings(viewmodel);
                viewmodel.getAll();
            });
        </script>

    </head>
    <body>
        <header>
            <div class="content-wrapper">
                <div class="float-left">
                    <p class="site-title">
                        @Html.ActionLink("ToDo Site", "Index", "Home")
                    </p>
                </div>
                <div class="float-right">
                    <section id="login">
                        @Html.Partial("_LoginPartial")
                    </section>
                </div>
            </div>
        </header>
        <div id="body">
            @RenderSection("featured", required: false)
            <section class="content-wrapper main-content clear-fix">
                @RenderBody()
            </section>
        </div>
        <footer>
            <div class="content-wrapper">
                <div class="float-left">
                    <p>&copy; @DateTime.Now.Year - ToDo Site - Istvan Reiter</p>
                </div>
            </div>
        </footer>
    </body>
</html>
```

Az @RenderBody lesz az, ami a megfelelő view kódját illeszti be ebbe a keretbe. Nekünk mindössze az Index view sablon kell, ezt a Views\Home\Index.cshtml fájl képviseli. A projekt sablon belerak egy csomó vackot, ettől megszabadulunk. A Knockout nélküli HTML az alábbi lesz:

```
@{
    ViewBag.Title = "Home Page";
}

<div class="container">
    <div class="row">
        <div class="span4">
            <button class="btn btn-info">Sort</button>
            <table id="tasks" class="table">
                <tbody>
                    <tr>
                        <td></td>
                        <td>
                            <div >
                                <button class="btn btn-primary">Done</button>
                            </div>
                        </td>
                        <td>
                            <i class="icon-ok"></i>
                        </td>
                        <td>
                            <button class="btn btn-warning">Delete</button>
                        </td>
                    </tr>
                </tbody>
            </table>
        </div>
        <div class="span8"></div>
    </div>

    <div class="row">
        <div class="span2">
            <form>
                <fieldset>
                    <legend>Add new Task</legend>
                    <input type="text" />
                    <button type="submit" class="btn">Add</button>
                </fieldset>
            </form>
        </div>
        <div class="span10"></div>
    </div>

</div>
```

Két sort készítettünk, az elsőt egy táblázat foglalja el, amiben majd az adatok foglalnak helyet, a másodikba pedig egy form elemet helyeztünk, amely majd a feltöltésről gondoskodik. A “class” attribútumok értékei mind a Bootstrap előre legyártott stílusait húzzák be, kezdve a “szép” táblázattal, bezárólag a jellegzetes kék gombig.

ToDoSite – Controller

Adjunk a Controllers mappához egy új controllert, méghozzá egy “API Controller with empty read/write actions” típusút. Nevezzük el ToDoController-nek. View-t nem kell generálni hozzá, jó parazita módjára megszálljuk majd a HomeController Index view-ját, ezzel nem kell törődni.

Ahogy azt jó pár fejezettel korábban már eldöntöttük, nem használjuk a Repository mintát, ezért a controllernek tartalmaznia kell egy ToDoContext példányt. Viszont, jómunkásember módjára ezt a példányt el is kell takarítanunk majd, ezért jó lenne használni az IDisposable interfészt. A jó hír, hogy maga az ApiController megvalósítja ezt az interfészt, ezért nekünk csak annyi a dolgunk, hogy készítünk a Dispose metódusból egy személyre szabott változatot, amely egyrészt meghívja a ToDoContext-en a saját Dispose-át (hiszen ő maga is IDisposable), másrészt ugyanezt megteszi az ApiController-en definiált Dispose metódussal. Írjuk hát a következőt:

```
// Fields

private ToDoContext context = new ToDoContext();

// Dispose

protected override void Dispose(bool disposing)
{
    context.Dispose();

    base.Dispose(disposing);
} // Dispose
```

Ezzel megvagyunk, nézzük a műveleteket, nem neveztem el őket külön, Get, Post, Put és Delete néven szerepelnek, hiszen mindegyikükből csak egy változat létezik. Get:

```
public IEnumerable<ToDoTask> Get()
{
    return context.Tasks.Where(
        ( task ) => task.UserName == WebSecurity.CurrentUserName );
} // Get
```

Meglehetősen egyszerű, ne feledjük, hogy új authentikációs szolgáltatást használunk már az MVC 4-ben, így a WebSecurity osztályon keresztül kérhetjük le az aktuális felhasználót. Post:

```
public HttpResponseMessage Post(ToDoTask task)
{
    if (task == null)
    {
        throw new HttpException(400, "Task cannot be null");
    }

    task.UserName = WebSecurity.CurrentUserName;
    //task.Date = DateTime.Now;
    context.Tasks.Add(task);
```

```
    context.SaveChanges();

    return Request.CreateResponse<ToDoTask>(HttpStatusCode.OK, task);
} // Post
```

Ez már lényegesen érdekesebb. Elsősorban ellenőrizzük, hogy a kliens oldalról valóban érkezett-e valami, vagy csak szemétkedik valaki velünk. Utóbbi esetben meghajítjuk egy 400-as kóddal (Bad request). Ezután két dolgot kell megtennünk, lekérjük a felhasználónevet, mivel kliens oldalról ezt nem kapjuk meg (nyilván megoldható lenne, én most így csináltam), illetve megadjuk a feltöltés dátumát. Ez ugye kliens oldalon is elérhető, viszont szerencsétlen módon a Javascript Date típusa nem kompatibilis a .NET DateTime típusával. Feltölti ugyan gond nélkül, de mivel a JSON deszérializálás nem tud vele mit kezdeni, ezért a DateTime alapértelmezett értékét kapja meg a szerveroldali objektumunk. Ez még a kisebbik baj, ugyanis SQL Compact-ot használunk, ezért a legkisebb dátum, amit ez kezelni tud az 1753, a DateTime-é viszont 0001, így akkora kivételt kapunk, hogy kilóg a képernyőről. Tegyünk hát egy kis kitérőt!

Ugye kliens oldalról szerver oldalra (és vice versa) JSON formában utaznak az adatok, tehát kézenfekvő a folyamat ezen részét megcélozni. Szerencsére lehetőségünk van tetszőleges JSON konverter megadására, írjunk hát egyet.
A Javascript féle Date.now() függvény a Unix epoch (1970.01.01) óta eltelt ezredmásodpercet adja vissza (nem pedig másodpercet, vigyázat), tehát semmi más dolgunk nincs, mint a beépített DateTime függvényekkel megdolgozni a bejövő adatokat. Készítsünk egy osztályt, amely a DateTimeConverterBase absztrakt osztályból származik, itt a komplett forráskód:

```
public class JsDateConverter : DateTimeConverterBase
{
    public override object ReadJson(JsonReader reader, Type objectType,
        object existingValue, JsonSerializer serializer)
    {
        if (reader.TokenType != JsonToken.Integer)
        {
            throw new Exception("Unexpected token");
        }

        return new DateTime(1970, 1, 1)
            .AddMilliseconds((long)reader.Value);
    }

    public override void WriteJson(JsonWriter writer, object value,
        JsonSerializer serializer)
    {
        if (!(value is DateTime))
        {
            throw new Exception("Unexpected type, shame on you");
        }

        var unix = new DateTime(1970, 1, 1);
        var delta = ((DateTime)value) - unix;

        if (delta.TotalSeconds < 0)
        {
```

```
            throw new Exception("1970.01.01, let me google that for you");
        }

        writer.WriteValue((long)delta.TotalMilliseconds);
    }
}
```

Látható, hogy összeadás, kivonás és kész is vagyunk, nem igényel különösebb magyarázatot. A bejövő adat esetében a típust (Integer) ellenőrizzük, visszafelé pedig az 1970 utáni dátumot. Persze a dátum UTC szerint megy, vagyis hozzánk képest 2 óra mínuszban van, erre nem árt figyelni.
Kész a konverter, már csak ki kell dekorálni a szükséges property-t a JsonConverter attribútummal és kész is vagyunk. Most lehet levenni a comment jelzést a Model fejezetbeli Date property-ről.

Még egy megjegyzés: A végére egy megjegyzés: az IE 8 óta az összes böngészőben elérhető a Date típus toJSON nevű függvénye, ami olyan formára hozza a dátumot, amivel az alapértelmezett konverter is elbír. Ha nem kell tekintettel lenni a régebbi böngészőkre, akkor bátran használjuk ezt.

Vissza a controllerhez, Put:

```
public void Put(ToDoTask task)
{
    var original = context.Tasks
        .SingleOrDefault((item) => item.Id == task.Id);

    if (original == null)
    {
        throw new HttpException(404, "The database doesn't contains this item");
    }

    context.Entry(original).CurrentValues.SetValues(task);
    context.SaveChanges();
} // Put
```

Végül a Delete:

```
public void Delete(int id)
{
    var task = context.Tasks
        .SingleOrDefault((item) => item.Id == (int)id);

    if (task == null)
    {
        throw new HttpException(404, "The database doesn't contains this item");
    }

    context.Tasks.Remove(task);
    context.SaveChanges();
} // Delete
```

A tanulság, hogy bár a műveletek maguk egyszerűek, de az apró részleteken nagyon könnyű úgy megcsúszni, hogy azt csak intenzív fórumolvasásokkal kompenzálhatjuk.
Majdnem végeztünk, marad az authentikáció. Írtam már erről, dekoráljuk ki a ToDoController osztályunkat az Authorize attribútummal:

```
[Authorize]
public class ToDoController : ApiController
{
    public IEnumerable<ToDoTask> Get()
    {
```

Ezután ha jogosulatlanul akarjuk használni a controller függvényeit szép 401-es hiba lesz a jutalmunk. Természetesen nem véletlenül használjuk az Internet Application sablont, be van már lőve a felhasználókezelés, az App_Start mappa AuthConfig.cs fájljában pedig a külső szolgáltatók oAuth szolgáltatását kapcsolhatjuk be, van itt Google, Microsoft, LinkedIn, kisnyúl…

ToDoSite – Kliens oldal

Elérkeztünk az utolsó felvonáshoz. Szerveroldalon kész vagyunk, már csak némi Javascript kód szükséges.

Természetesen a Knockout könyvtárat fogjuk használni, ennek beállításáról a korábbi bejegyzésekben untig lehet olvasni, így ettől most megkímélem magunkat. Ajánlom viszont a Visual Studio 2012 Update 2 csomag telepítését ami kiváló Intellisense támogatást ad a Knockout adatkötéseihez.

A projecten belül létrehoztam egy ClientScripts nevű könyvtárat, ezen belül pedig a ToDoViewModel.js nevű fájlt amiben tulajdonképpen a komplett kliens oldali kód terpeszkedik el.

Elsősorban szükségünk lesz egy viewmodel-re, amely majd a szerver felől érkező adatokat tárolja:

```
function TaskViewModel(task) {
    var self = this;

    self.id = task.id;
    self.description = task.description;
    self.date = task.date;
    self.isCompleted = task.isCompleted;
    self.userName = task.userName;
}
```

A paraméterként megadott task objektumon is a kisbetűs tulajdonságokat tudjuk hívni, hiszen a szerveroldali osztályokat elláttuk a JsonProperty attribútumokkal, így illeszkednek a Javascript elnevezési konvenciójához.
Kell még egy osztály, ez tartalmazza majd a műveleteket, illetve egy listát a felhasználó teendőiről. A műveletek nélkül ez így néz ki:

```
function ToDoViewModel() {
    var self = this;

    self.tasks = ko.observableArray();
    self.taskToAdd = ko.observable("");
}
```

A taskToAdd ebben az esetben egy szimpla string érték lesz, a teljes objektumot majd a feltöltéskor rakjuk össze. Jöjjön a getAll, amivel lekérjük a feladatok listáját:

```
self.getAll = function () {
    self.tasks.removeAll();

    $.ajax({
```

```
        url: "api/ToDo/",
        accepts: "application/json",
        cache: false,
        statusCode:
            {
                200: function (data) {
                    $.each(data, function (key, value) {
                        self.tasks.push(new TaskViewModel(value));
                    });
                },
                401: function () {
                    window.location = "/Account/Login/";
                }
            }
    });
};
```

A statusCode segítségével tudjuk eldönteni, hogy be van-e a felhasználó jelentkezve, vagy sem. Utóbbi esetben 401-es kódot kapunk vissza, ekkor átirányítjuk a felhasználót a login oldalra. Itt természetesen más megoldás is lehetséges, ez volt a legegyszerűbb.

Következik a post függvény. A felhasználónevet itt most üres stringnek hagytam, annak ellenére, hogy az megérkezik a feladatok listájával.

```
self.post = function () {
    var task = JSON.stringify({
        id: -1,
        description: self.taskToAdd(),
        date: Date.now(),
        isCompleted: false,
        userName: ""
    });

    $.ajax({
        url: "api/ToDo/",
        accepts: "application/json",
        cache: false,
        type: "POST",
        contentType: "application/json; charset=utf-8",
        data: task,
        success: self.getAll
    }).fail(function (xhr, textStatus, err) {
        alert(err);
    });

    self.taskToAdd("");
};
```

A módosításhoz segítségünkre van a Knockout, így azonnal hozzájutunk a szükséges objektumhoz. Ebben az esetben a módosítás csak annyit jelent, hogy a feladatot elvégzettnek jelöljük, így a függvény első sorában mindössze át kell állítanunk az isCompleted tulajdonságot. Az url-hez pedig hozzá kell még csapnunk az objektumunk azonosítóját, hogy szerveroldalon megtaláljuk azt az adatbázisban. Figyelni kell viszont arra, hogy a felküldeni kívánt példány nem

“átlagos” Javascript objektum, ezért a Knockout beépített toJSON függvényét kell használni ahhoz, hogy kinyerjük belőle az adatokat.

```
self.update = function (task) {
    task.isCompleted = true;

    $.ajax({
        url: "api/ToDo/" + task.id,
        datatype: "json",
        contentType: "application/json; charset=utf-8",
        type: "PUT",
        cache: false,
        data: ko.toJSON(task),
        success: self.getAll
    }).fail(function (xhr, textStatus, err) {
        alert(err);
    });

    return true;
};
```

Végül a törlés hasonlóképpen igényli az url-hez adott azonosítót, de semmi mást, hiszen nem kell feltöltenünk külön adatokat.

```
self.delete = function (task) {
    $.ajax({
        url: "api/ToDo/" + task.id,
        type: "DELETE",
        cache: false,
        success: self.getAll
    }).fail( function ( xhr, textStatus, err ) {
        alert(err);
    });
};
```

Készítettem még egy rendező függvényt is, amely a feladatok állapota (kész-nem kész) alapján sorba rakja azokat:

Már csak annyi a dolgunk, hogy a View fejezetben elkészített HTML kódhoz hozzáadjuk a Knockout kötéseit és készen is vagyunk. Ha emlékszünk, akkor a Twitter Bootstrap könyvtárral tulajdonképpen két sorba rendeztük az oldalunkat, a felső sorban egy táblázat a feladatokkal, az alsóban pedig az új feladat űrlap.

A felső sor kódja:

```
<div class="row">
    <div class="span4">
        <button class="btn btn-info" data-bind="click: sort">Sort</button>
        <table id="tasks" class="table">
            <tbody data-bind="foreach: tasks">
                <tr>
                    <td data-bind="text: description"></td>
```

```
                    <td>
                        <div data-bind="ifnot: isCompleted">
                            <button class="btn btn-primary"
                                data-bind="click: $root.update">Done</button>
                        </div>
                    </td>
                    <td data-bind="if: isCompleted">
                        <i class="icon-ok"></i>
                    </td>
                    <td>
                        <button class="btn btn-warning"
                            data-bind="click: $root.delete">Delete</button>
                    </td>
                </tr>
            </tbody>
        </table>
    </div>
    <div class="span8"></div>
</div>
```

A rendező gombbal kezdünk, erre csak a sort függvényt kell rákötni. Ezután a táblázaton a már ismert foreach kötést alkalmazzuk, de van egy kis csavar, mégpedig a kész/nem kész állapotok miatt. Ha nem végeztük el az adott feladatot, akkor egy gombot kell megjelenítenünk, ellenkező esetben pedig valami mást. Erre használjuk az if és ifnot kötéseket, amelyek segítségével adott HTML elemet megjeleníthetünk, illetve elrejthetünk. Minden más standard, nincs látnivaló.

A második sor ennyire sem érdekes, minden további nélkül:

```
<div class="row">
    <div class="span2">
        <form data-bind="submit: post">
            <fieldset>
                <legend>Add new Task</legend>
                <input type="text"
                    data-bind="value: taskToAdd, valueUpdate: 'keyUp'" />
                <button type="submit" class="btn">Add</button>
            </fieldset>
        </form>
    </div>
    <div class="span10"></div>
</div>
```

Ezzel el is készültünk. A végső forráskód megtalálható az alábbi webhelyen:

https://github.com/reiteristvan/ToDoSite/