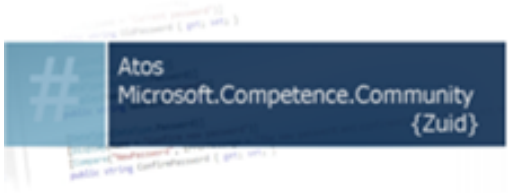


Atos

Workshop JavaScript

jQuery, jQueryUI, KnockoutJS

Marco Franssen, Jan Saris, Sander van de Velde
23-9-2011

Inhoudsopgave

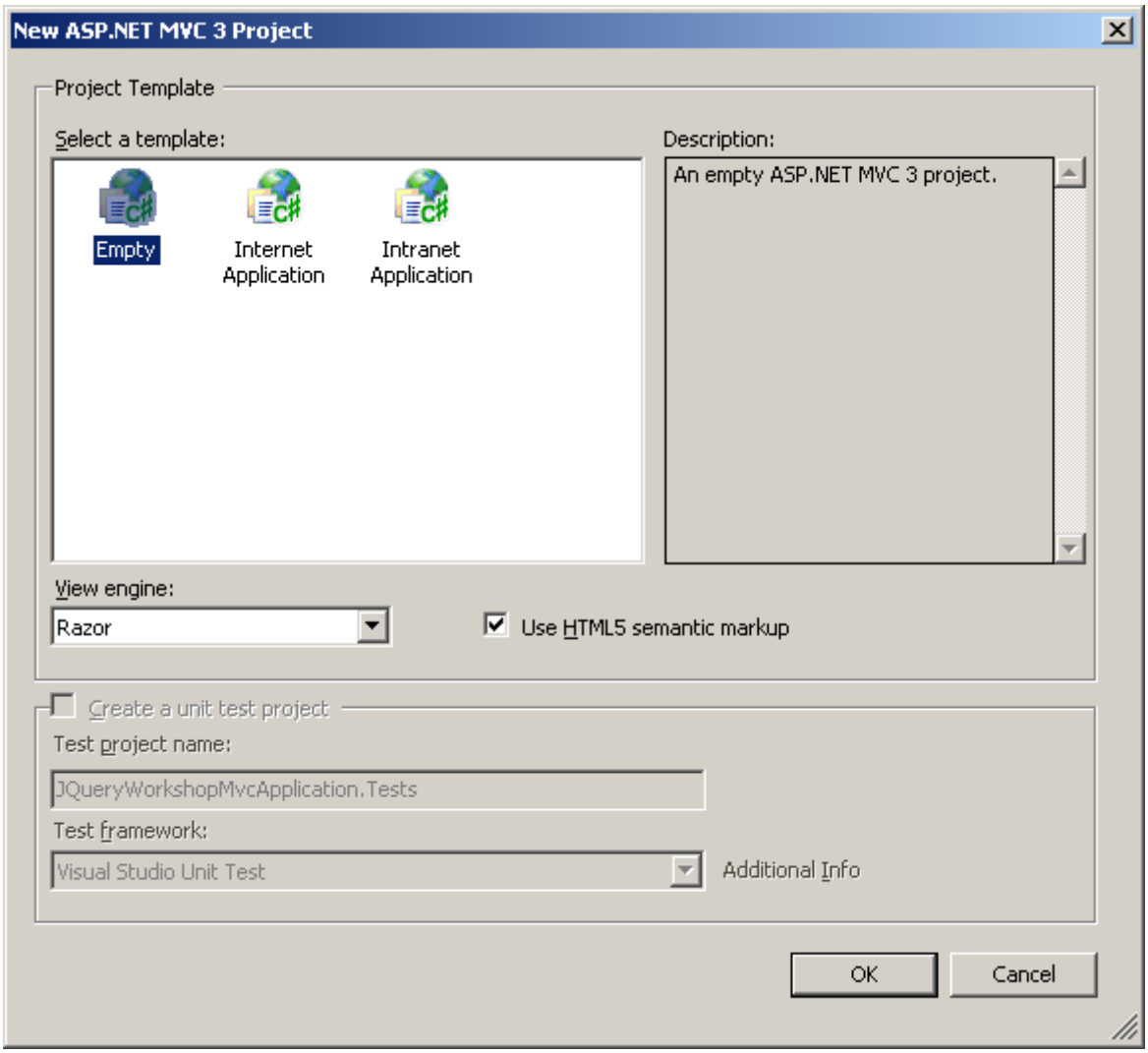
1. Basic jQuery

Welkom bij deze introductie op jQuery. Dit hoofdstuk zal in het teken staan van browser DOM manipulatie. Voor diegene die niet weet wat de browser DOM is: dit is een afkorting voor Document Object Model; oftewel, het is een boomstructuur van alle HTML elementen op het scherm. Deze elementen kunnen genest zijn: Een DIV kan andere elementen bevatten.

Voordat we met de cursus beginnen, bouwen we eerst een project structuur op in VS2010. We gaan een Aps.Net MVC3 project gebruiken.

1.1 Creatie van een MVC3 project

Open een Kaal Aps.Net MVC3 project genaamd JqueryWorkShopMvcApplication aan in VS2010 (kies voor de mogelijkheid met Razor[1] en HTML5 ondersteuning)



We kunnen natuurlijk jQuery met de hand downloaden van de website (http://docs.jquery.com/Downloading_jQuery) maar we kunnen ook Nuget gebruiken om jQuery als

[1] Razor is een notatie voor het schrijven van code die de HTML help genereren als de opmaak met de data in het scherm verweven moet worden.

package toe te voegen aan ons project. Maar standaard is er al een (oudere) versie van de jQuery Nuget package aanwezig in een MVC3 project. Die passen we toe.

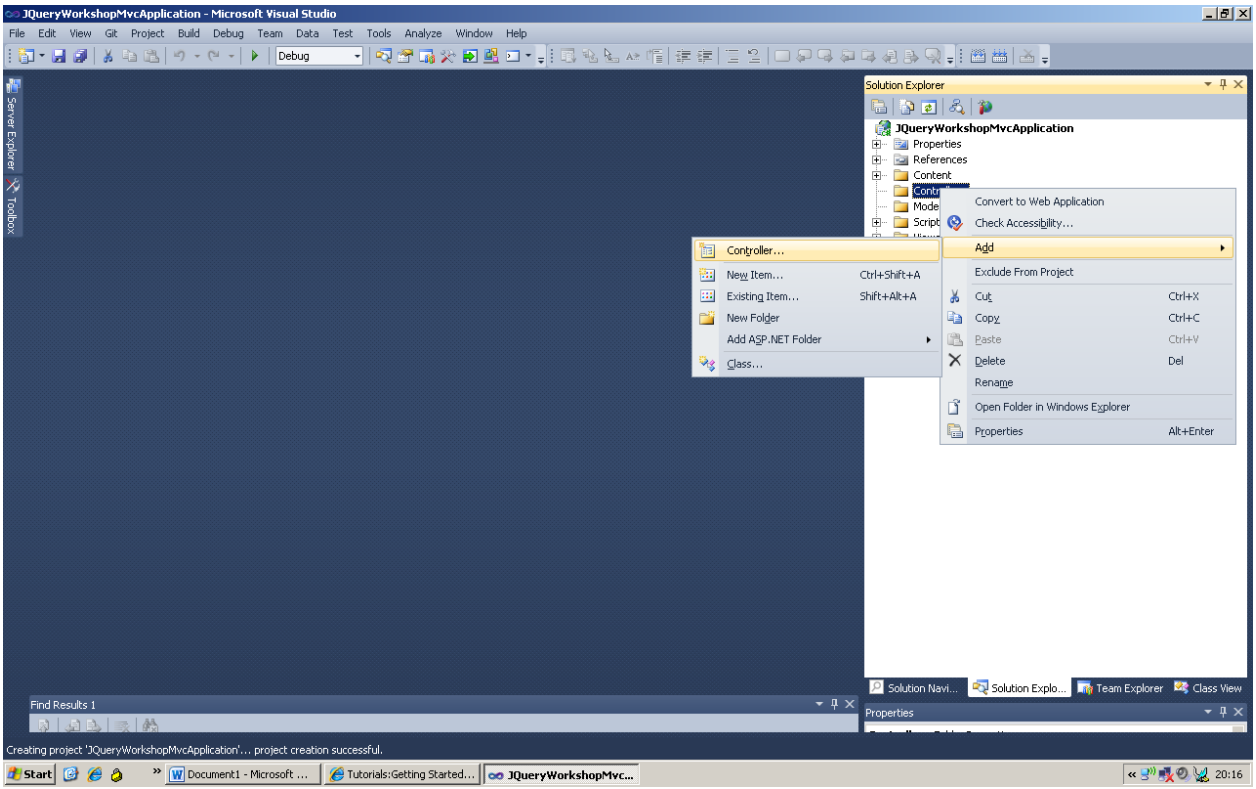
Controleer dus dat de _Layout.cshtml (de masterpage equivalent in Razor) (aanwezig in \Views\Shared) met de juiste referenties gevuld is.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript"></script>
</head>

<body>
    @RenderBody()
</body>
</html>
```

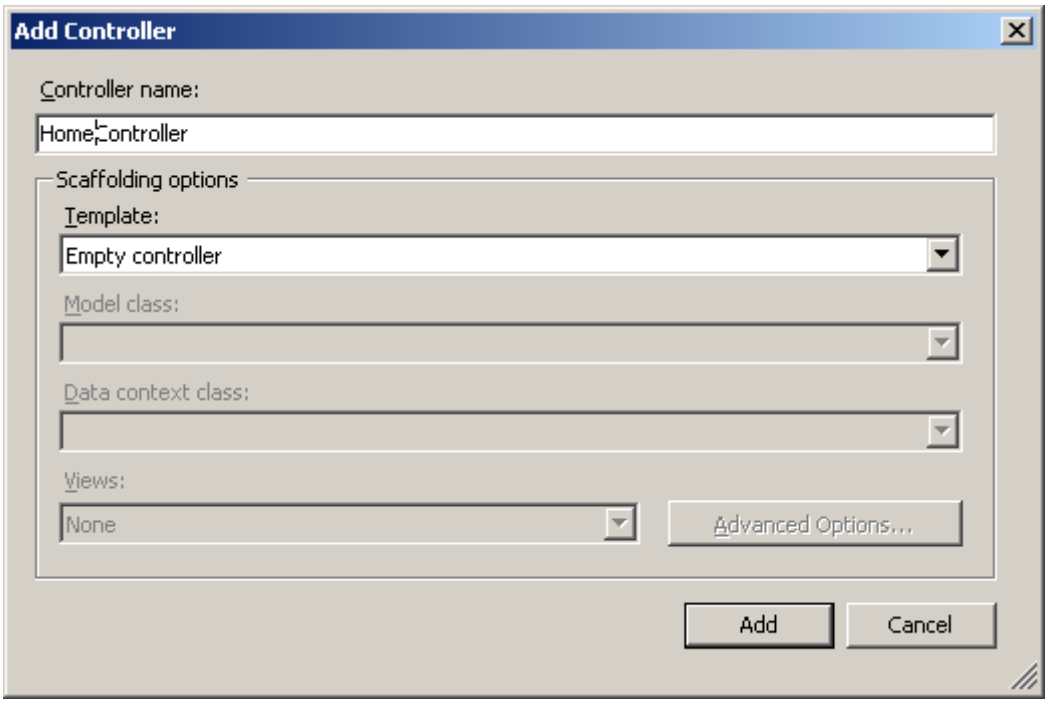
Tip1: de standaard aanwezige modernzr JavaScript bibliotheek is weggelaten (buiten scope voor onze workshop)

Tip2: Eventueel kan /Scripts/jquery-1.5.1.js gerefereerd worden. Dit bestand met exact dezelfde logica is niet geminified. Dwz. de minified versie is geoptimaliseerd voor versturen over het internet door deze zo klein mogelijk te maken (commentaar is verwijderd, functienamen zijn verkort, etc.) De niet-minified versie is prettiger bij JavaScript debug-werkzaamheden.

Voeg een HomeController toe (rechtermuisklik op de de controllers map):

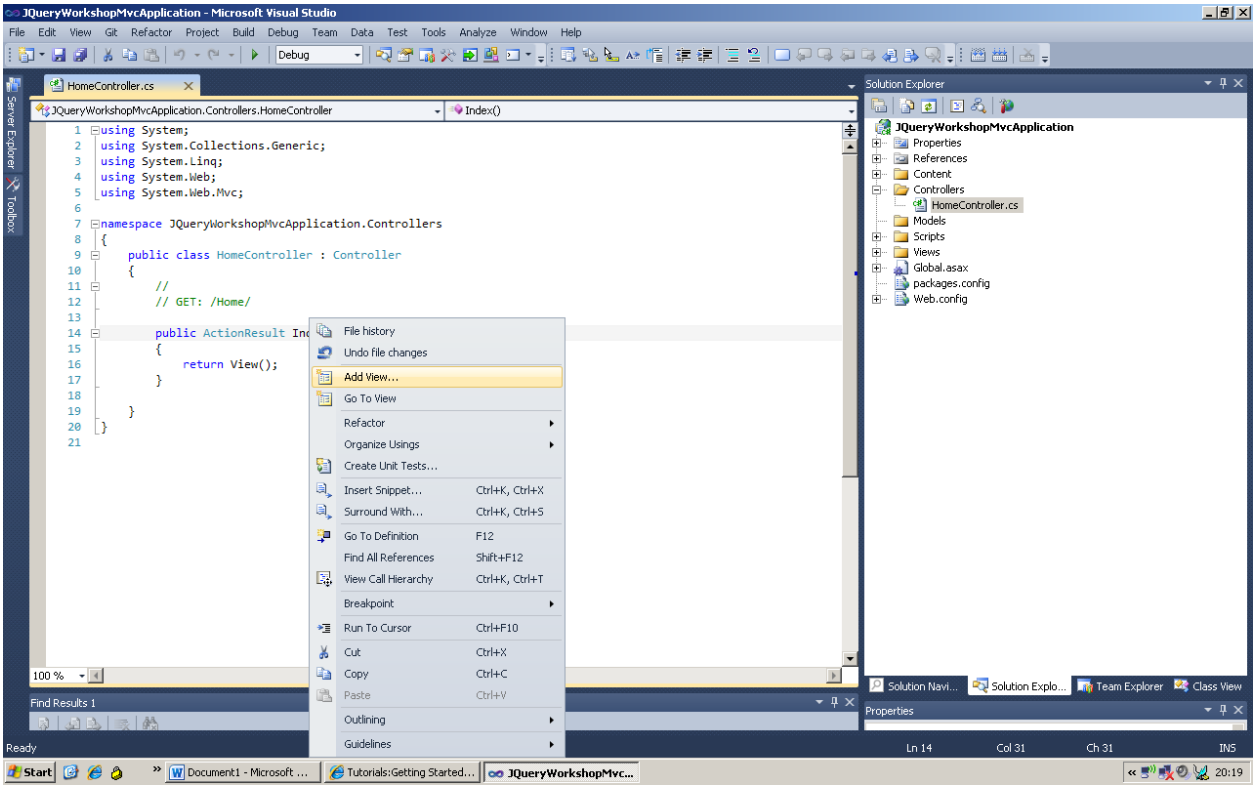


Noem deze HomeController:

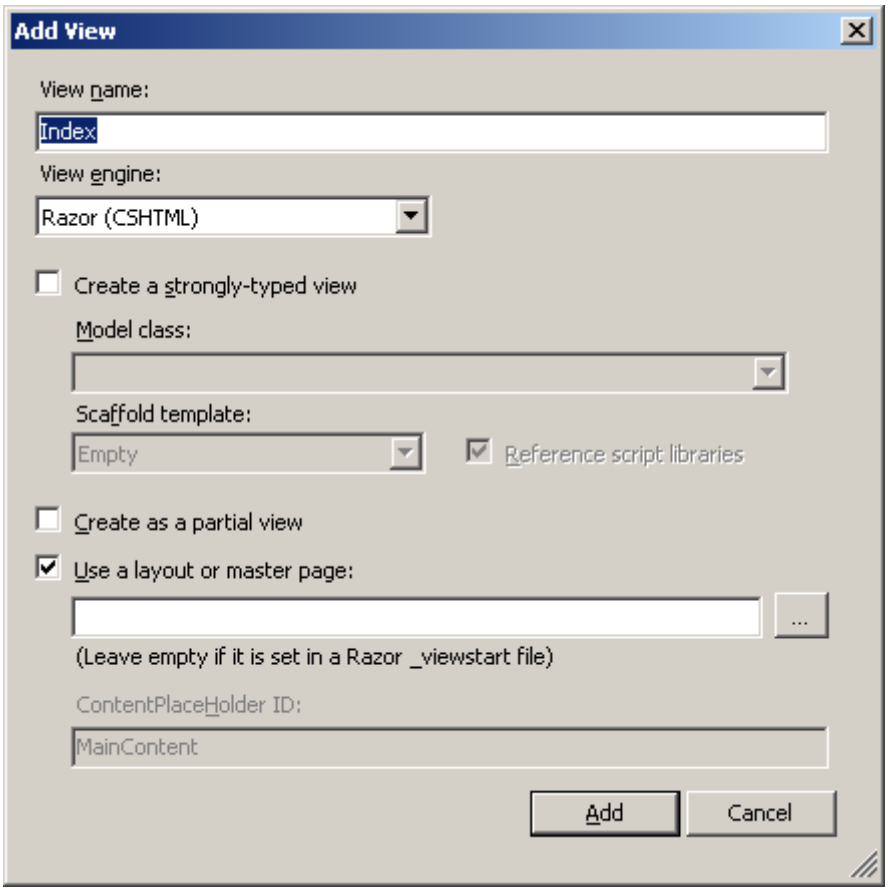


Zie dat er na de generatie een Index methode aanwezig is. Dit is een MVC action, een methode die via als een http aanroep te refereren is en normaal gezien een HTML pagina representeert.

Maak een View aan voor deze action (rechtermuisklik op de methode)



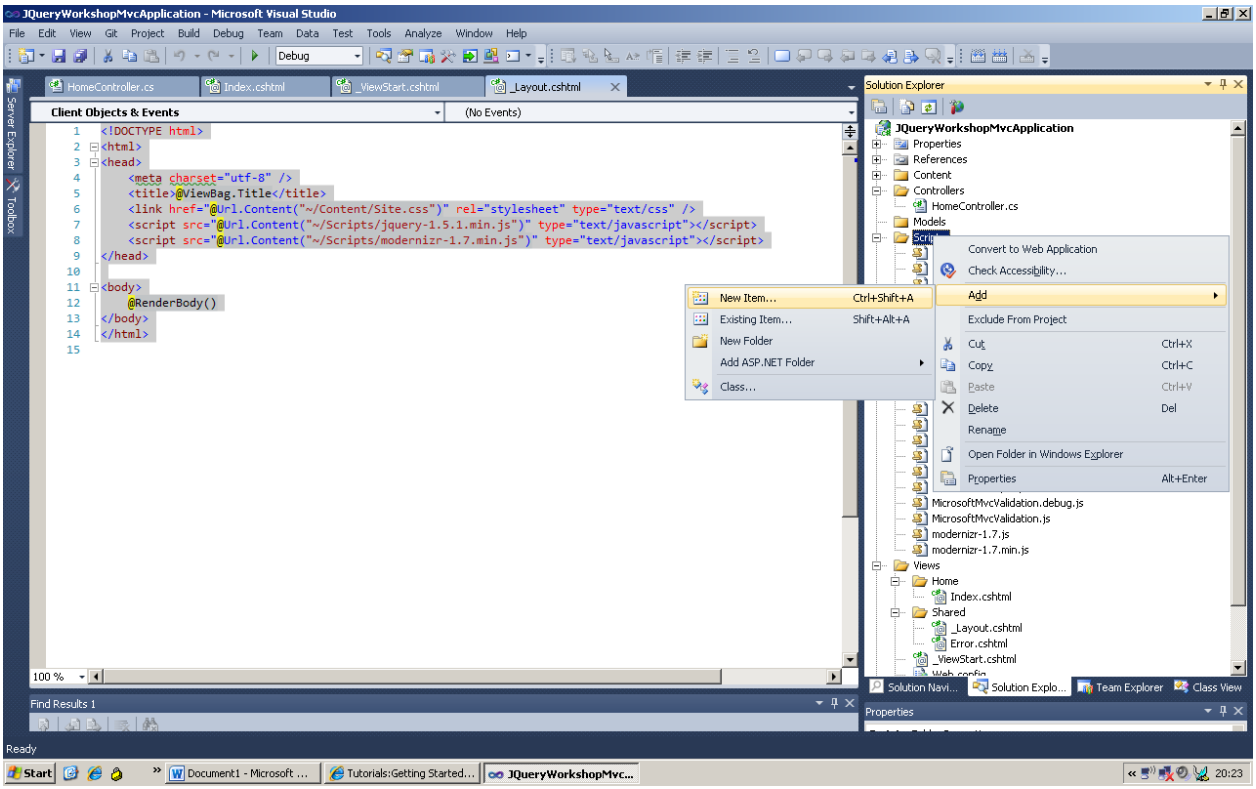
Gebruik de standaard instellingen:



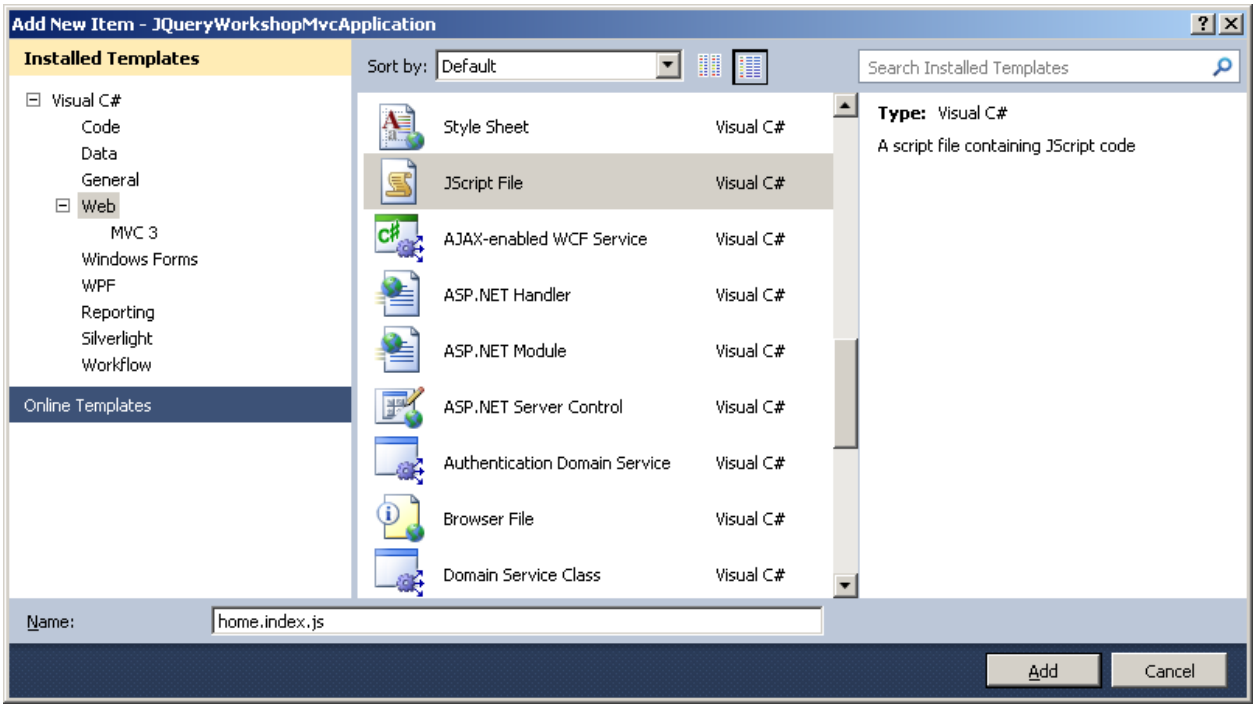
De View met de naam Index is aangemaakt:

```
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>
```

Maak nu een ander bestand genaamd home.index.js aan in de scripts map via add New item. Hier gaan we de jQuery statements in schrijven, deze code is namelijk te debuggen via breakpoints in de ‘gutter’.



Het is dus een JavaScript bestand:



Het JavaScript bestand is nu geopend in de Editor.
Sleep nu de jquery-1.5.1-vsdoc.js in de text editor . Er komt een regel 'commentaar' beschikbaar:

```
/// <reference path="jquery-1.5.1-vsdoc.js" />
```

Dit maakt het voor VS2010 mogelijk om CodeInside aan te bieden voor jQuery. Erg handig.

Voeg een lege functie toe:

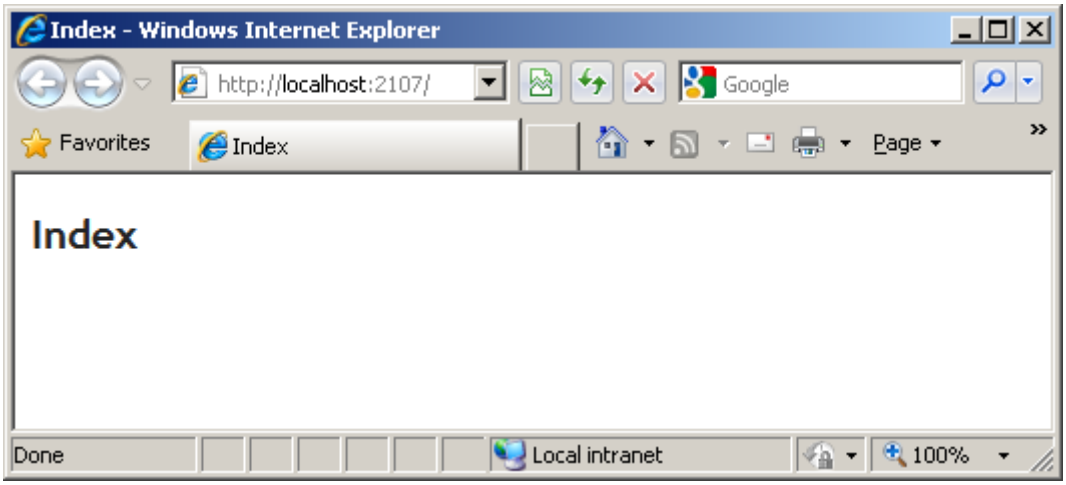
```
/// <reference path="jquery-1.5.1-vsdoc.js" />

function ExecuteOnStartup() {
  // Add your script here
}
```

Refereer nu vanuit de index.cshtml naar het JavaScript bestand:

```
@{
    ViewBag.Title = "Index";

    <script src="@Url.Content("~/Scripts/home.index.js")" type="text/javascript"><
/script>
}

<h2>Index</h2>
```

Alles staat nu klaar om met de workshop te beginnen.
Compileer even om te controleren of alles werkt ;-)

Dit moet de volgende pagina geven:



De gegenereerde HTML is eenvoudige en duidelijke HTML5 opmaak:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Index</title>
    <link href="/Content/Site.css" rel="stylesheet" type="text/css" />
    <script src="/Scripts/jquery-1.5.1.js" type="text/javascript"></script>
</head>
<body>
       <script src="/Scripts/home.index.js" type="text/javascript"></script>

<h2>Index</h2>
</body>
</html>
```

Tip: Deze is gegenereerd door de combinatie van _Layout.cshtml, de index.cshtml, data uit de controller en de Razor ViewEngine .

Voeg nu een aantal HTML objecten en een stukje JavaScript toe aan de index.cshtml:

```
@{
    ViewBag.Title = "Index";

    <script src="@Url.Content("~/Scripts/home.index.js")" type="text/javascript">
</script>
}

<h2>Index</h2>

<a id="link1" href="">Link1</a>
<br />
<a id="link2" href="">Link2</a>

<script type="text/javascript">
    $(document).ready(function () {
        ExecuteOnStartup();
    });
</script>
```

We zien hier twee linkjes zonder echte logica (er zal niks gebeuren als je er op klikt).

Het stukje JavaScript onderaan zal uitgevoerd worden als deze pagina in de browser laadt. De opdracht daarin is niks anders dan een stukje jQuery code wat zegt:

“Zodra de pagina geheel in de browser geladen is en de gebruiker bijna de eerste controls kan gaan aanklikken, dan moet de ExecuteOnStartup methode uitgevoerd worden”.

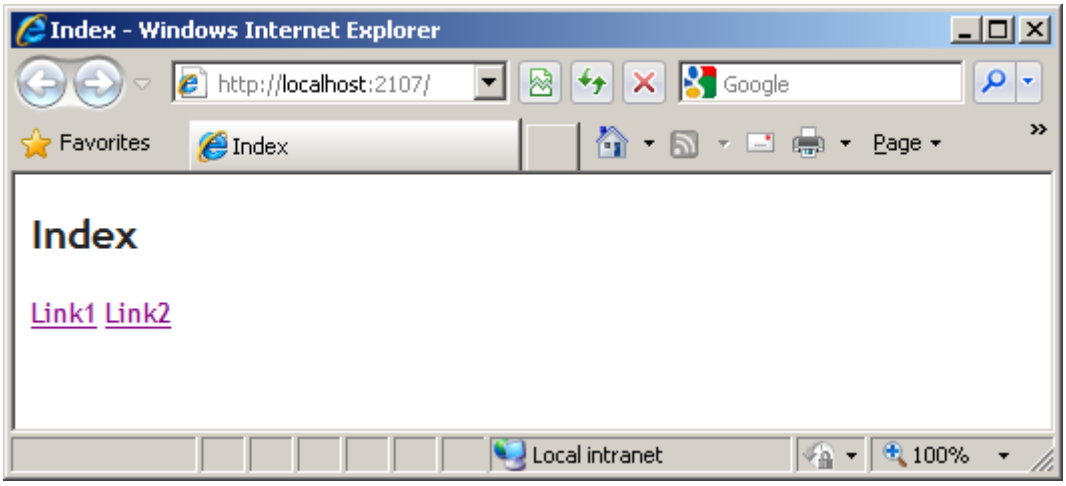
Dit is essentieel! De .ready(); gaat pas af op dat moment af en dit kan per merk browser verschillen! jQuery lost specifieke browser-afhankelijkheden voor ons op.

We passen de ExecuteOnStartup() iets aan zodat er iets zal gebeuren:

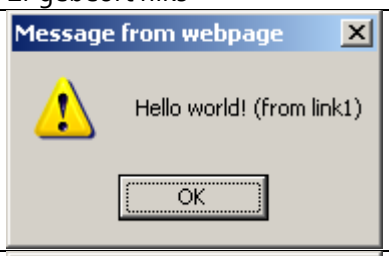
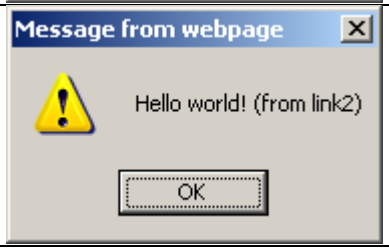
```
function ExecuteOnStartup() {

    // add to all links (the a elements) a click function
    $("a").click(function () {
        alert("Hello jQuery workshop (from " + this.id + ")");
    });
}
```

Hier zien we het eerste voorbeeld van een jQuery selector. Voor alle elementen die voldoen aan de eis dat ze simpelweg een link zijn (een 'a'), wordt aan hun 'click' event een functie toegevoegd (en de code wordt pas uitgevoerd als er op geklikt wordt). Hierin wordt een dialoogje getoond met daarin een tekst en de waarde van het 'id' attribuut van het element waarop *geclicked* is.

Start nu de applicatie en bekijk het resultaat:



We klikken op ieder element:

De H2 (de header met de tekst Index)	Er gebeurt niks
De eerste link	
De tweede link	

Iedere link voert dus die ene functie uit. Een selector kan dus 0, 1 of zelfs meerdere elementen opleveren waar iets mee gedaan moet worden.

jQuery bevat vele varianten op selectors. Er staat een uitputtende lijst op
http://api.jquery.com/category/selectors/

We kunnen ook functies aan andere DOM element events hangen. Zie ook
http://api.jquery.com/category/events/

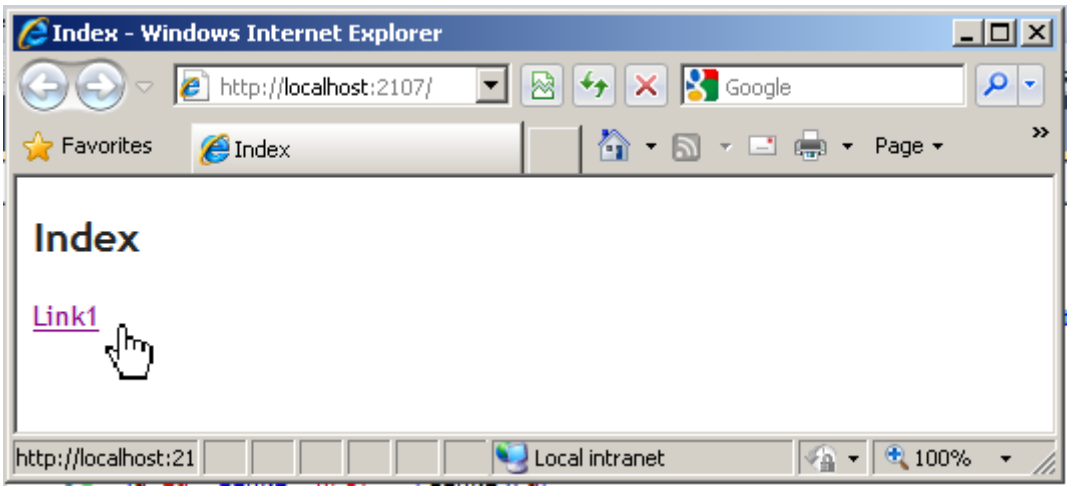
Hier gaan we dieper op het reageren op events in.
We passen dus de ExecuteOnStartup weer iets aan:

```
function ExecuteOnStartup() {

    // add to all links (the a elements) a click function
    $("#link1").mouseenter(function () {
        $("#link2").hide(500);
    });

    $("#link1").mouseleave(function () {
        $("#link2").show(1000);
    });
}
```

Wat we nu zien is data an het specifieke element link1 (en alleen deze; dat komt door het # wat jQuery vertelt alleen naar elementen met deze UNIEKE 'id' te zoeken) maar liefst twee functies te koppelen.

Als de muis naar link1 gaat, dan wordt specifiek link2 onzichtbaar gemaakt. Dit wordt uitgevoerd in 500 milliseconden en dit geeft dus een animatie. Als de muis link1 weer verlaat, dan moet link2 weer zichtbaar gemaakt worden in 1000 milliseconden. Ook dit geeft dus een animatie.

Probeer het maar eens:

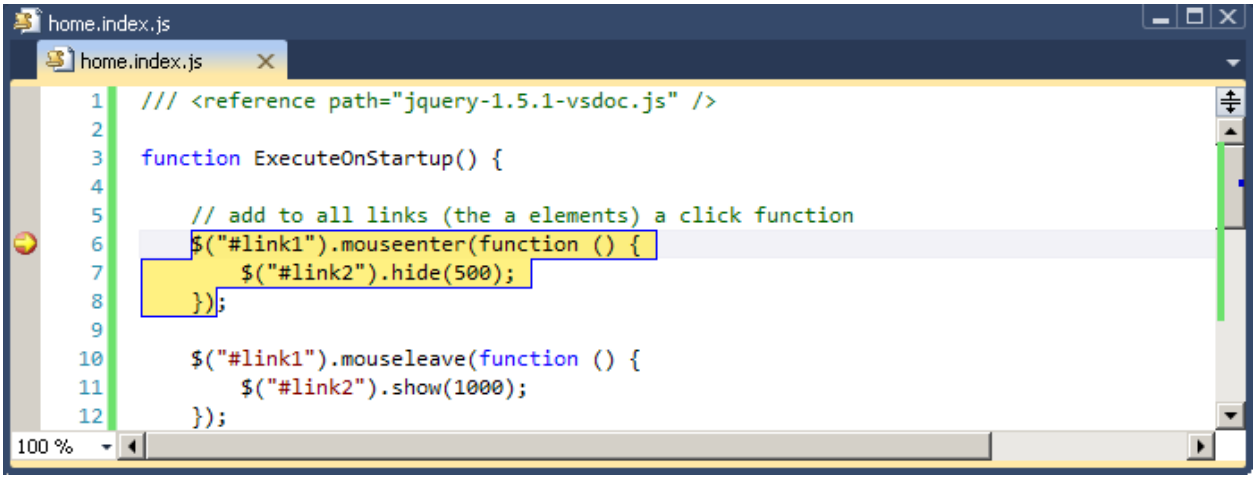


Zie dat de tweede link langzamer terugkomt dan verdwijnt.

We zien hier twee belangrijke aanwinsten met jQuery. De selectors maken het veel eenvoudiger om specifieke elementen te vinden, zelfs als deze genest zijn zoals rijen in een tabel. En jQuery beidt simpele animaties wat een prettige beleving van de pagina voor de gebruiker is.

Er zijn flink wat effecten mogelijk, deze gaan we in hoofdstuk 3.5 bespreken.

Overigens, wie een breakpoint in de home.index.js heeft opgenomen, ziet dat VS2010 netjes stopt op deze breakpoint (dit werkt alleen als je in Internet Explorer test):

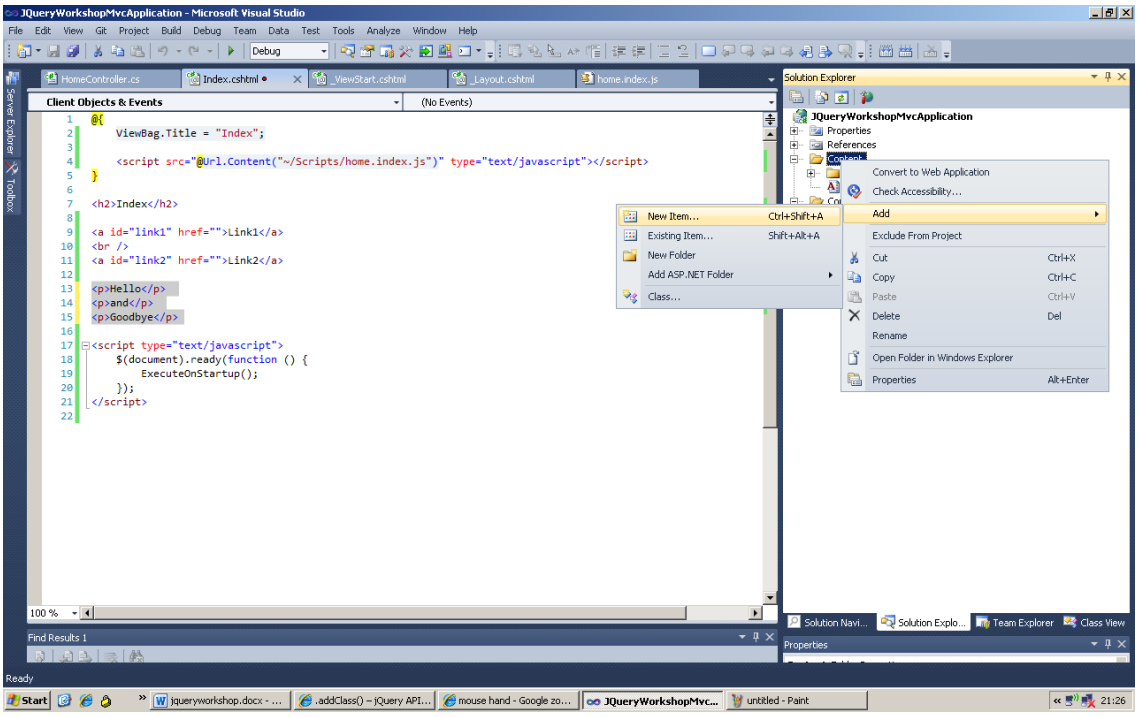


Met bv. het VS2010 Immediate scherm of een mouse-over is de inhoud van de JavaScript elementen te achterhalen.

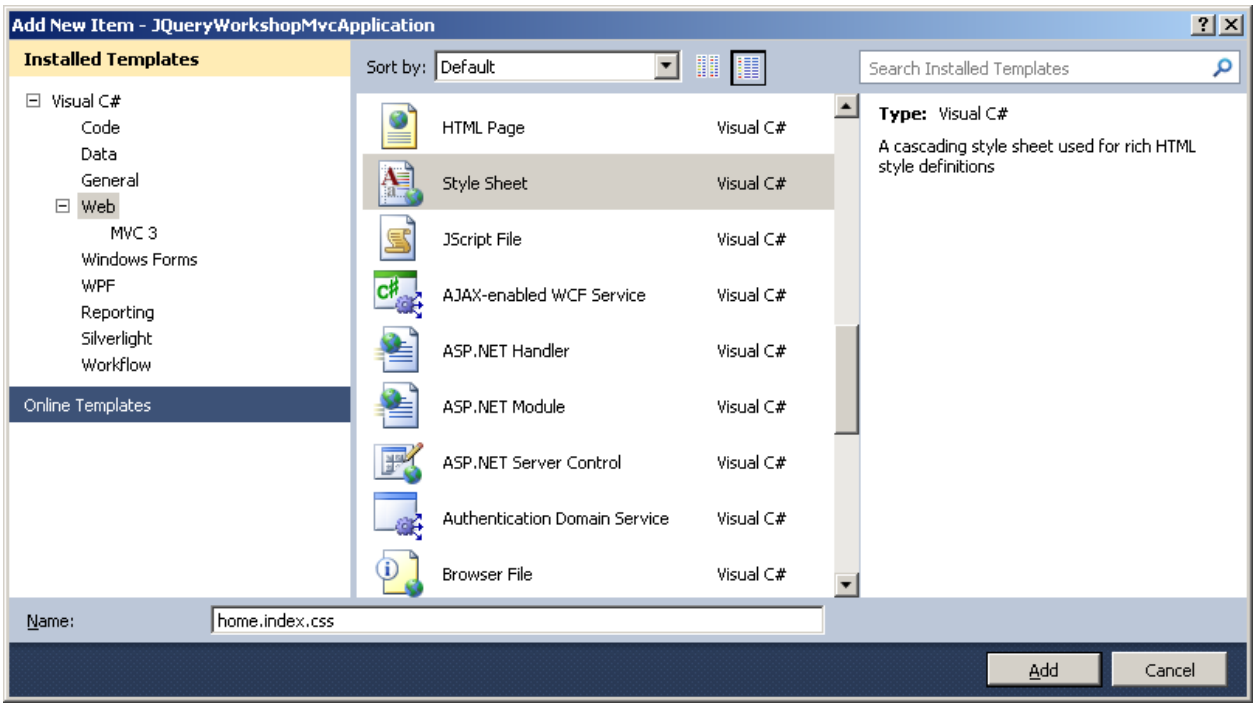
Laten we eens wat meer proberen met de selectors.
Voeg eerst een aantal paragraph elementen toe na de a elementen:

```
...
<p>Hello</p>
<p>and</p>
<p>Goodbye</p>
...
```

En we maken een nieuw stylesheet bestand aan in de Content map:



Noem deze: home.index.css:



En voeg deze toe aan de index.cshtml:

```
@{
    ViewBag.Title = "Index";

    <script src="@Url.Content("~/Scripts/home.index.js")" type="text/javascript"></script>
    <link href="@Url.Content("~/Content/home.index.css")" rel="stylesheet" type="text/css" />
}

<h2>Index</h2>

<a id="link1" href="">Link1</a>
<br />
<a id="link2" href="">Link2</a>

<p>Hello</p>
<p>and</p>
<p>Goodbye</p>

<script type="text/javascript">
    $(document).ready(function () {
        ExecuteOnStartup();
    });
</script>
```

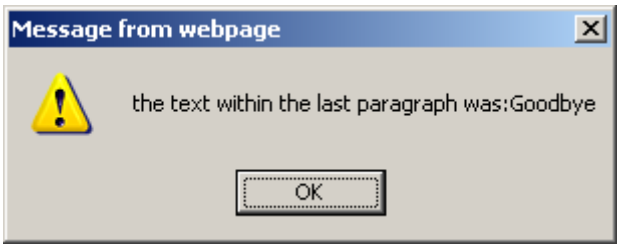
In de CSS moeten de volgende 'styles' toegevoegd worden:

```
  p { margin: 8px; font-size:16px; }

  .selected { color:blue; }
```

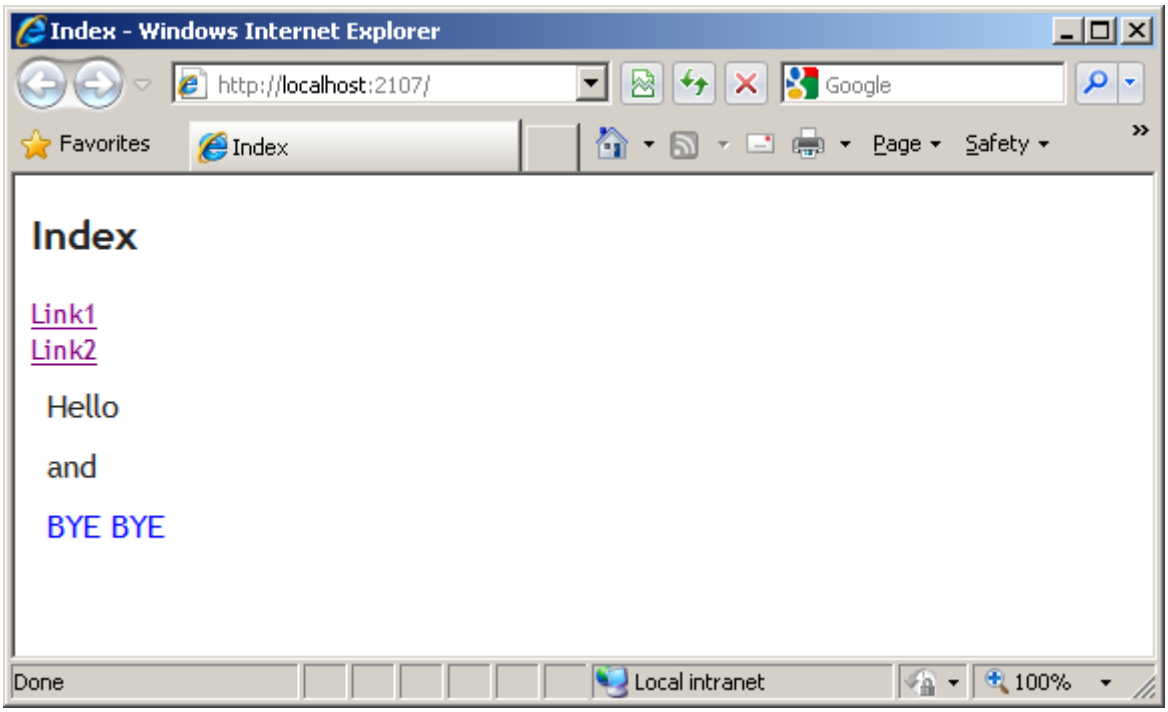
En aan de ExecuteOnStartup voegen we achteraan de volgende regel toe:

```
…
var lastParagraph = $("p:last");
alert("the text within the last paragraph was:" + lastParagraph.text());
lastParagraph.addClass("selected").text("BYE BYE");
…
```

Voer dit eens uit. Eerst volgt er een alert:



Dan volgt de volgende weergave in de browser:



Wat is er gebeurd? Deze volgende situatie is ontstaan:

- Alle drie de paragrafen hebben al een andere marge en lettergrootte gekregen (dit is standaard CSS gedrag)
- Alle laatste paragrafen (de ene dus) worden door een variabele gerepresenteerd. (de selector hoeft maar één keer de DOM af te lopen op zoek naar elementen)
- We geven een alert af met de inhoud van de laatste paragraaf
- We hebben gecodeerd dat de LAATSTE paragraaf de extra selected stijl heeft gekregen
- En we hebben de tekst in de paragraaf overschreven

Tip: Zag je dat we in de laatste regel dezelfde paragraaf twee maal manipuleerden in één zin? Alle functies (zoals de .addClass("...") en de .text("...")) manipuleren de aangeboden lijst van objecten en geven die daarna weer terug. De elementen die dus door de .addClass() zijn gemanipuleerd, worden dus ook door de .text() gemanipuleerd. Dit noemen we een *Fluent* notatie en dat zijn we ook terug bij bv. LinQ en EF 4.1 Code First.

De .ready() is ook al een soort van selector, wat grappig.

We kunnen dus met jQuery ook de attributen van (meerdere of afzonderlijke) elementen uitlezen en aanpassen. En we kunnen de CSS stijlen uitlezen en aanpassen.

Het is zelfs mogelijk om elementen te selecteren op het wel of niet hebben van een bepaalde CSS stijl. Dit is erg krachting. Zo kunnen heel snel afzonderlijke elementen (met dezelfde CSS stijl) een nieuwe CSS look krijgen.

Laten we dus nog iets meer met selectors spelen.
Voeg aan de index.cshtml een tabel toe met een kolomnaam en vier tabelregels:

```
<table border="1">
```

```
<thead>
    <tr><td>Header</td></tr>
</thead>
<tbody>
    <tr><td>Row with Index #0</td></tr>
    <tr><td>Row with Index #1</td></tr>
    <tr><td>Row with Index #2</td></tr>
    <tr><td>Row with Index #3</td></tr>
</tbody>
</table>
```

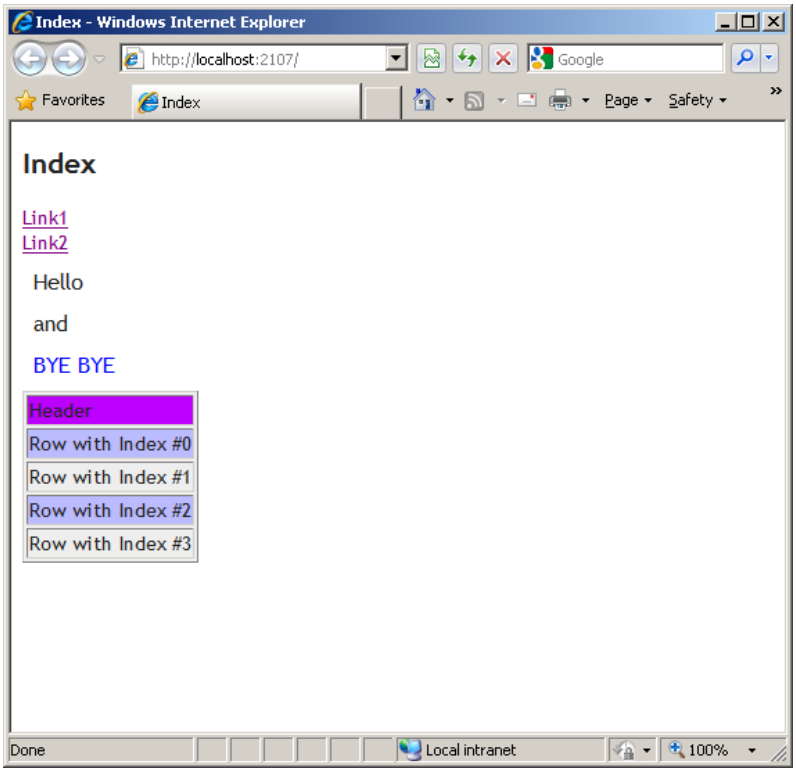
Voeg wederom achteraan de ExecuteOnStartup wat JavaScript toe:

```
…
$("table thead tr:even").css("background-color", "#bb00ff");

$("table tbody tr:even").css("background-color", "#bbbbff");
…
```

En we hebben nog wat extra styling nodig voor de table in de home.index.css :

```
…
table {
  background:#eeeeee;
}
…
```

Dit geeft de volgende tabel invulling:



Wat is hier gebeurd?

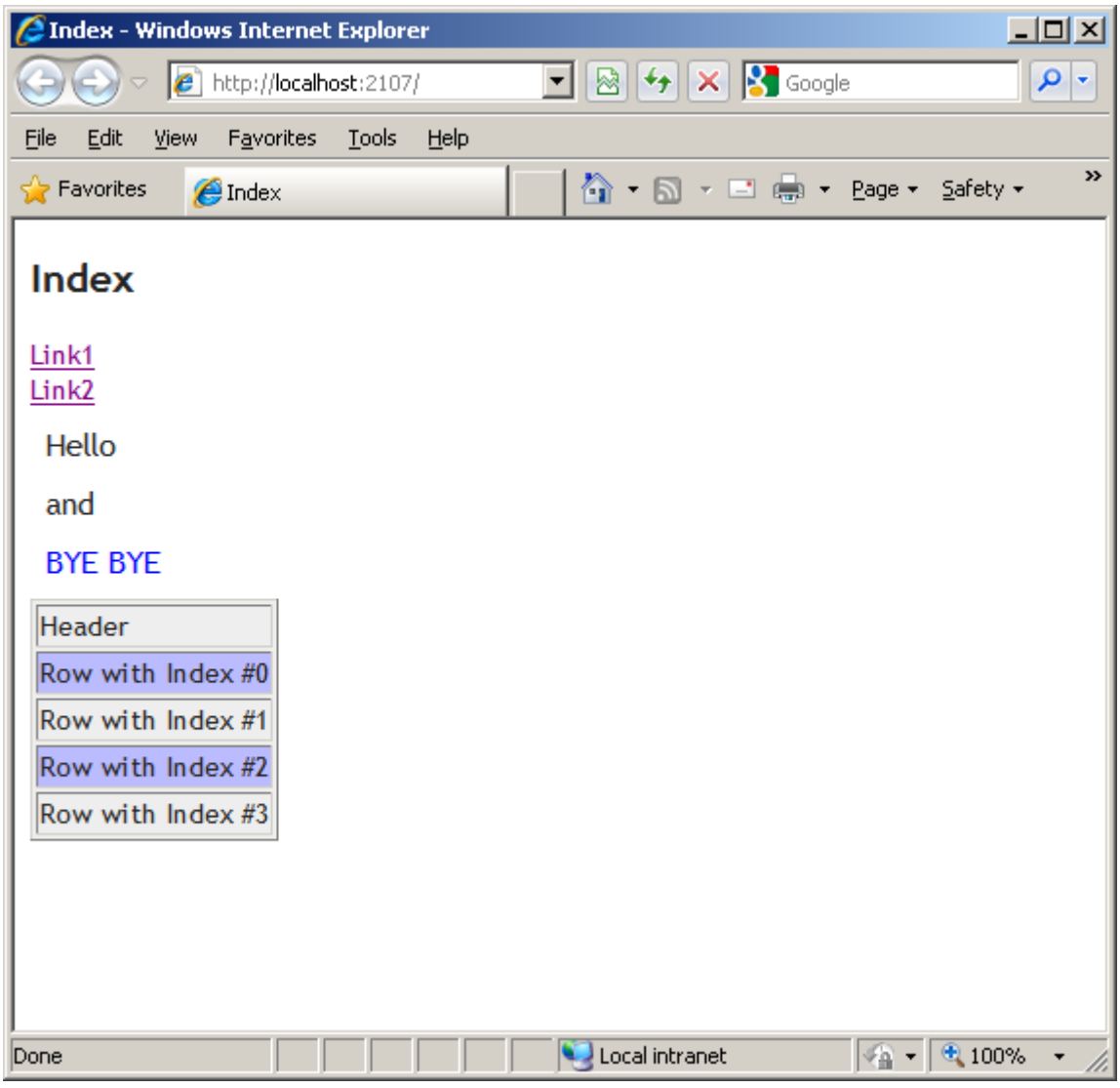
- De totale tabel heeft een achter achtergrond kleurtje gekregen (standaard CSS gedrag)
- De even regels in de tabel head krijgen een ander kleurtje
- De even regels in de tabel body krijgen een ander kleurtje

We kunnen dus bij de selector ook specifiek genest gaan zoeken. Dit heeft de voorkeur bij grote documenten met veel DOM elementen waar slechts enkele objecten moeten worden aangepast. Het zoeken naar elementen is gewoon het aflopen van eigenlijk alle elementen en dit kost tijd en energie (het verbruikt tenslotte stroom van je PDA en het is toch jammer als je merkbaar minder kunt bellen na het bezoek van een website).

Overigens zal de header row NIET gemanipuleerd worden als via de selectie van oneven regels in de header:

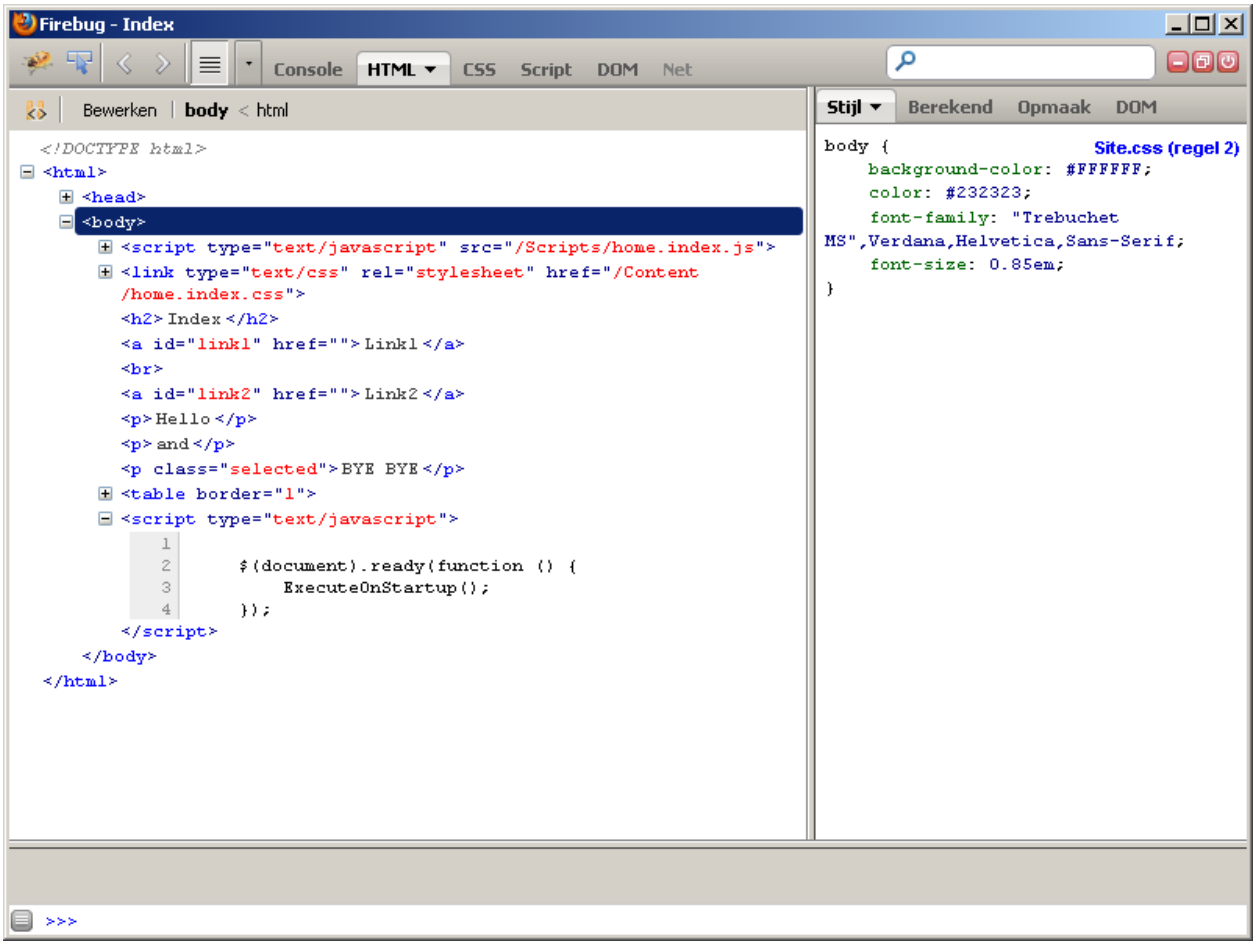
```
$("thead tr:odd").css("background-color", "#bb00ff");
```

Dit geeft namelijk geen verschil op de header:



De regel met daarin "Header" is de nulde regel, dus een even regel, niet een oneven regel… De selector retourneerde dus NUL (= Geen) elementen.

Overigens is de DOM, dus de boomstructuur aan elementen zoals de browser die ziet, ook in te zien in moderne browsers of via plugins zoals Firebug:



Als laatste gaan we nog items in een ordered list manipuleren.

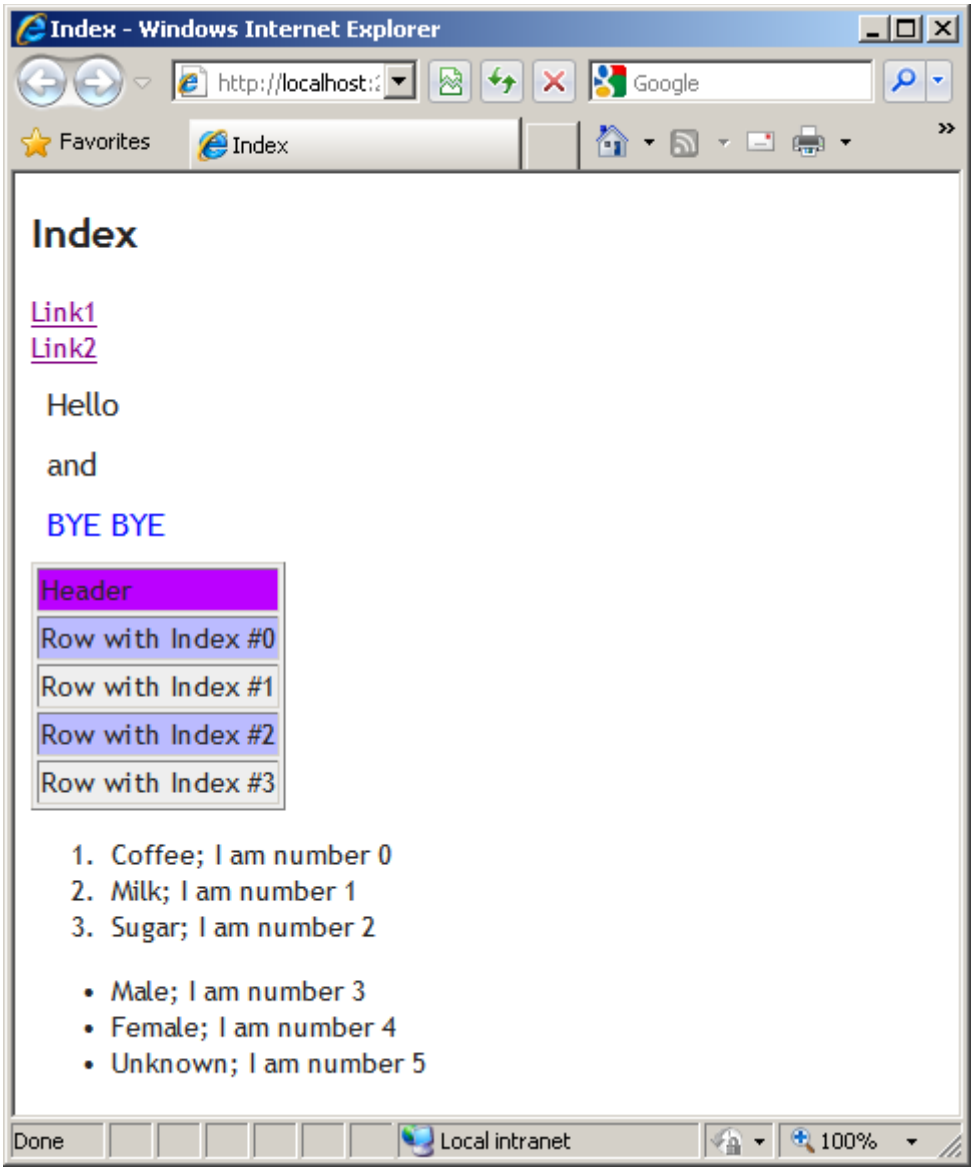
We voegen eerst twee 'ordered' lists toe onderaan de index.cshtml. In HTML worden dan straks volgnummers of bullits voor de items gezet, afhankelijk van het type:

```
<ol class="orderedlist">
    <li>Coffee</li>
    <li>Milk</li>
    <li>Sugar</li>
</ol>

<ul class="orderedlist">
    <li>Male</li>
    <li>Female</li>
    <li>Unknown</li>
</ul>
```

Daarna voegen we de volgende regels JavaScript toe aan de ExecuteOnStartup:

```
…

$(".orderedlist").find("li").each(function (i) {
    $(this).append("; I am number " + i);
});
…
```

Als we dit uitvoeren dan zien we dat achter iedere *list item* extra tekst is toegevoegd:



De 'ordered' lists zijn overigens gekozen via hun ondersteuning van een CSS class. En het interessante is dat de telling doorloopt. Hoe komt dat? Waarom is er niet opnieuw geteld (dus waarom is het 0,1,2,3,4,5 ipv. 0,1,2,0,1,2)?

Dit komt omdat de `$(".orderedlist")` simpelweg een stukje DOM representeert, hier de twee lijstjes. Daarin gaat de `.find("li")` alle list items achterhalen. Dus er worden zes list items aangetroffen en niet twee keer drie items. Let hier dus goed op.

Conclusie

Het is heel eenvoudig geworden om met slechts enkele regels jQuery code een aantrekkelijke (..) pagina te creëren. Had jij dit ook met standaard JavaScript kunnen schrijven en had die code ook in alle browsers gewerkt?

Bonus opdracht 1:

Hoe zou jij de vraag naar telling 0,1,2 0,1,2 in de laatste opdracht oplossen?

Bonus opdracht 2:

Voeg een checkbox toe aan de pagina en als ik die van waarde verander, toon of verberg dan de tabel. (hints: input, type , checkbox, .change() .toggle())

3. Ajax binnen jQuery bereik

In het vorige hoofdstuk hebben we gezien hoe we de DOM kunnen onderzoeken en manipuleren via jQuery. In dit hoofdstuk willen we laten zien hoe gemakkelijk het is geworden om Ajax calls uit te voeren binnen een website[2].

3.1 Terminologie

We kennen allemaal wel de mogelijkheid om een Post uit te voeren in de browser, bewust of onbewust. Veel schermen met invoer, zelfs inlogschermen, werken met enkele invoervelden en een 'submit' button. In het algemeen wordt dan alle invoer naar de server gestuurd, deze wordt dan afgehandeld door de achterliggende serverlogica en er wordt een nieuw scherm gegenereerd. Dit mechanisme heet een **post-back**.

In dit hoofdstuk gaan we een **call-back** uitvoeren. De gebruiker (of zoiets als een timer; in moderne browsers kan dit ook een gebeurtenis op het Host OS zijn (bv. een lokatiewijziging) initieert de uitvoer van een stukje JavaScript. Er is bv. wat data ingevoerd in invoervelden. Vervolgens wordt deze informatie naar de server doorgegeven, maar alleen deze data. Die data moet op de server worden verwerkt en de call-back geeft weer informatie terug. Dat is in het algemeen weer alleen wat data, geen stukken schermopmaak.

Die callback poging kan wellicht mislukken of lang op zich laten wachten. Daarom gebeurt dit **asynchroon**. De browser vuurt de aanroep af en blijft niet wachten op een antwoord. Nee, er wordt alleen een voorschot genomen in de vorm van: als er ooit een antwoord komt, dan moet bij terugkomst nog wat extra code uitgevoerd worden, namelijk een bepaalde JavaScript functie. Dit zijn dan de callback functies. En de attributen van die functies zijn de antwoorden vanuit de server.

Ok, simpel. Maar hoe ziet die communicatie tussen browser en server er uit? Dit is in het algemeen opgelost met een query string (een 'fat' URL) voor de aanroep en het antwoord is verpakt in **Json**. En Json is niks anders dan een alternatieve notatie, net zoals Xml dat is. Xml is gewoon minder toepasselijk omdat dit tot grotere berichten kan leiden (de belangrijkste reden is de dubbele omarmingen binnen Xml). Json is lichtgewicht en het heeft nog een ander voordeel. Json data is heel simpel naar JavaScript variabelen om te zetten, dit kost amper processor tijd.

[2] Communicatie tussen verschillende websites wordt als een beveiligingsrisico gezien. Er is wel een mogelijk in de vorm van JsonP. Deze geeft de mogelijkheid om de aanvragen een callback functie op te geven. De opgevraagde site zal zijn antwoord dan via die callback functie doorgeven. Dit valt buiten scope voor deze workshop.

3.2 Implementatie

Laten we beginnen met het aanbieden van data voor Ajax calls.
We maken een extra action (methode) aan op de server controller:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult DoSomething(string name, string phone, int? zip, string email)
{
            var myJsonResult = new MyJsonResult();
            try
            {
                // do some filtering
                 Thread.Sleep(2000);

                List<Person> persons = new List<Person>
                {
                    new Person{ID=1, Name="Donald Duck", Age=77},
                    new Person{ID=2, Name="Kwik", Age=74},
                    new Person{ID=3, Name="Kwek", Age=74},
                    new Person{ID=4, Name="Kwak", Age=74}
                };

                myJsonResult.Persons = persons.ToArray();

                myJsonResult.Succeeded = true;
            }
            catch (Exception)
            {
                myJsonResult.Succeeded = false;
            }

            return new JsonResult { Data = myJsonResult };
        }
```

Hier wordt dus een methode aangeboden om name, phone, zip en email door te geven. Omdat dit met een Post is geïnstrumenteerd, zou dit kunnen duiden op het feit dat we hier data (op de server) gaan wijzigen. Dit is dan niet met een spider uit te voeren.

Tip: de nullable int is als nullable ingesteld om te voorkomen dat we een 500 (internal server errror) foutmelding krijgen als de aangeboden data 'undefined' is in JavaScript. Het veld zou in de TextBox leeg gelaten kunnen blijven. Met de 'nullable' int kan de invoer wel op de parameters gemapped worden.

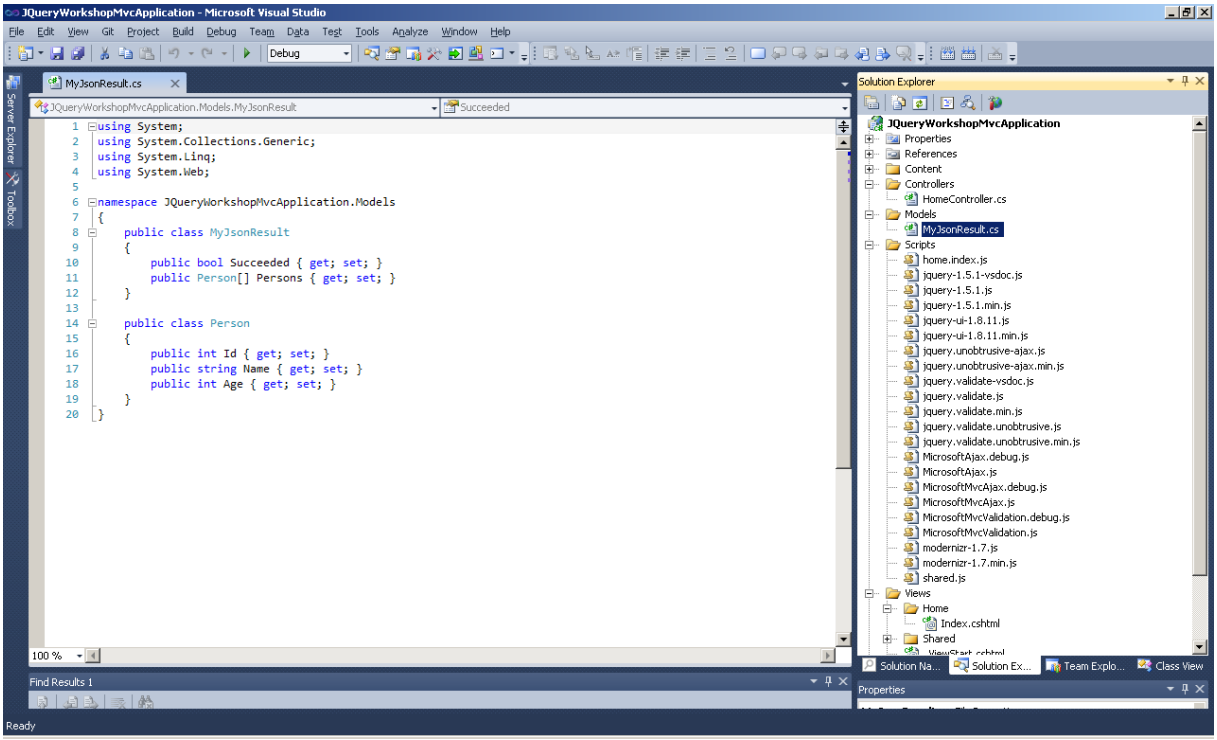
Tip: We moeten natuurlijk nog steeds testen of alles correct doorgegeven wordt maar dat is het altijd met Ajax: vertrouw nooit wat je van buiten aangeleverd krijgt!

Voor onze workshop is de functionaliteit iets simpeler: we wachten twee seconden en geven een lijstje met niet al te jeugdigen terug.

Het belangrijkste van deze action logica op de server is geschreven op de laatste regel van de action. Hier geven we een speciale overerving van ActionResult terug, namelijk onze objecten verpakt als JsonResult.De C# objecten die we teruggeven zijn namelijk zo gedefinieerd op de server.

```
    public class MyJsonResult {
        public bool Succeeded { get; set; }
        public Person[] Persons { get; set; }
    }

    public class Person {
        public int ID { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }
    }
```

Maak dus deze klassen aan in de Models map van onze MVC applicatie:

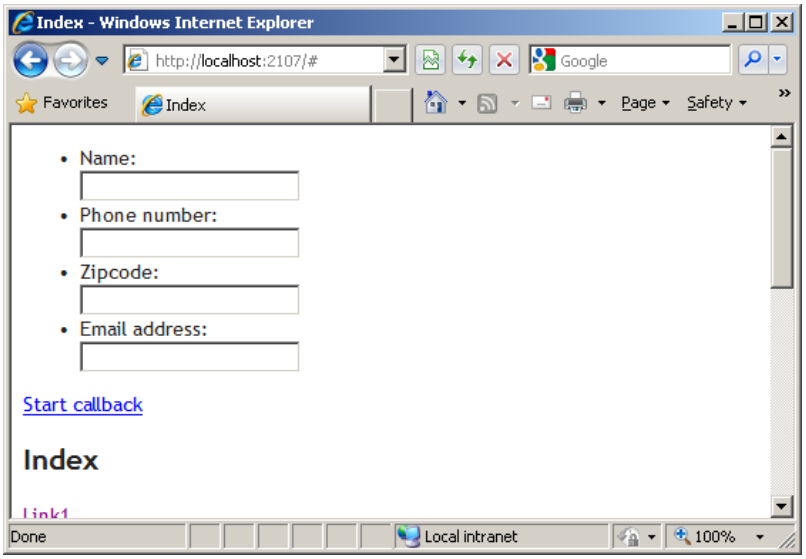


In de index.cshtml moeten we de aanroep gaan voorbereiden. Eerst zorgen we dat er een aantal velden beschikbaar komen waarin we de gebruiker eea. laten invoeren. Voeg dit bij voorkeur BOVENAAN in de index.cshtml toe. Dit geeft later een beter effect…

```
…
<hr/>

<ul>
    <li>
        Name:
        <br />
        <input type="text" name="name" placeholder="name" id="name" />
    </li>
    <li>
```

```
        Phone number:
        <br />
        <input type="tel" name="phone" placeholder="Phone" id="phone" />
    </li>
    <li>
        Zipcode:
        <br />
        <input type="number" name="zip" placeholder="Numbers" id="zip" />
    </li>
    <li>
        Email address:
        <br />
        <input type="email" name="Email" placeholder="Please enter your email address" id="email" />
    </li>
</ul>
<a href="#" id="btnStartCallback">Start callback</a>

<div id="result"></div>

<div id="errorInfo"></div>
…
```

Dit zal er dan zo uit gaan zien.



Hier kan de gebruiker eea. invullen.

Tip: We gebruiken hier een link om een klik op uit te laten voeren. Dat had beter een echte submit button kunnen zijn. Dit voorkomt dat spiders per ongeluk de Ajax call aftrappen en zo onze database ongewild vullen of legen.

Nu moet de ingevulde informatie (de zipcode is overigens een integer) naar een action "DoSomething" op de Home controller verstuurd worden. We laten op de server het adres samenstellen door de viewengine voorafgaande aan het tonen van de pagina aan de gebruiker. Daarvoor laten we de URL van de Ajax call samenstellen op de index.cshtml en het adres wordt dan op de client beschikbaar als JavaScript parameter van de ExecuteOnStartup. Pas dus de parameter aan:

```
<script type="text/javascript">
    $(document).ready(function () {
        ExecuteOnStartup('@Url.Action("DoSomething", "Home")');
    });
</script>
```

Tip: Deze manier van URL samenstellen zorgt ervoor dat de Ajax call ook blijft werken als de code uiteindelijk op de server (lees: andere machine) wordt uitgerold.

De parameter moet dus ontvangen gaan worden op de functie:

```
function ExecuteOnStartup(callbackUrl) {
...
```

Wat rest is dan nog de call zelf. We gebruiken hiervoor de.ajax().

Deze call is iets wat uitgebreid maar geeft een goede vergelijking met wat er op de server gebeurd.

Voeg de volgende code toe aan de ExecuteOnStartup:

```
$('#errorInfo').hide();
$('#name').val("Sinterklaas");
$('#phone').val("91 588 10 00");
$('#zip').val("28001");
$('#email').val("sint@madrid.es");

$('#btnStartCallback').bind('click', function (e) {
    var name = $('#name').val();
    var phone = $('#phone').val();
    var zip = $('#zip').val();
    var email = $('#email').val();

    var dataToSend = {
        'name': name,
        'phone': phone,
        'zip': zip,
        'email': email
    };

    $.ajax({
        url: callbackUrl,
        data: dataToSend,
        dataType: 'json',
        type: 'POST',

        beforeSend: function () {
            // before callback
            $('#errorInfo').show().text("Busy...");
```

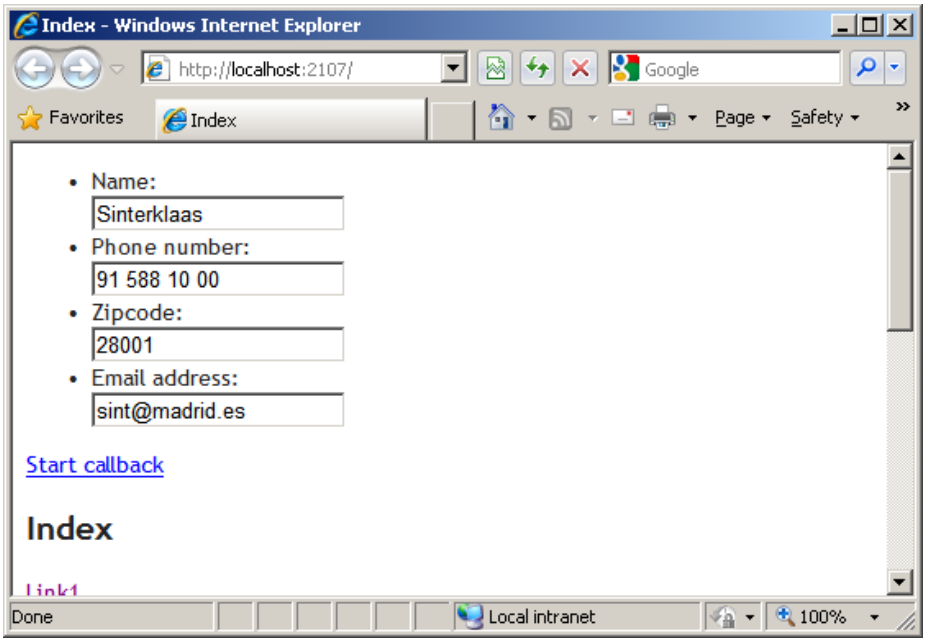
```
            $('#result').empty();
        },
         success: function (result) {
            if (result.Succeeded) {
                // after a successfull callback
                $.each(result.Persons,function () {
                    $('#result').append(this.Name + ";");
                });

                $('#errorInfo').text("Ready!").hide(2000);
            }
            else {
                // after an unsuccessful callback
                $('#errorInfo').text("Invalid input");
            }
        },

        error: function (xhr, err) {
            // after an error
            $('#errorInfo').text("Status: " + xhr.status);
        }
    });
});
```

Wat zien we hier staan? Eerst kijken we naar 'errorInfo'. Dit is een div-je waarin errors en andere ondersteunende boodschappen getoond worden. Die verstoppen we tijdens de ready.

Vervolgens vullen we de invoer controls met verschillende waarden (dit gaat via de .val('some value') functie) welke de 'Value' attributen vult.

Dat wordt er aan de start URL een binding gelegd tussen het 'click' event en de een functie. Dit is een alternatieve notatie voor de .click() (dit had ook een 'tap' op een iPhone kunnen zijn ☺) De .bind() is dus een generieke manier van functies aan events *binden*.

Uiteindelijk wordt nog vóór het uitvoeren van de callback het scherm zo getoond:

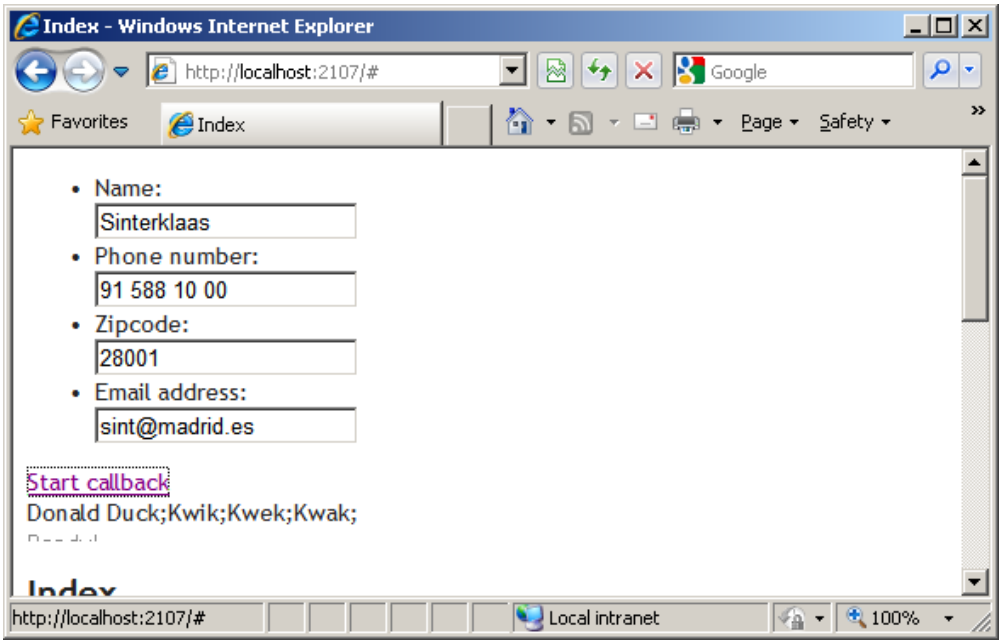


Als er dus een 'click' event optreedt, dan wordt de volgende code uitgevoerd:

1. Eerst lezen de values van de verschillende invoer controls uit via .val(). Al deze waarden worden dan tot één JavaScript variable samengevoegd.
2. Daarna volgt de Ajaxcall waarbij we de volgende zaken instellen:

url	De aan te roepen URL voor de callback. Hier is dat /Home/DoSomething.
data	De door te geven data.
dataType	Het formaat waarin gecommuniceerd gaat worden. Dit is hier 'json'.
type	De type HTTP aanroep, in dit geval een 'POST' (gelijk aan het attribuut op de action op de controller).
`beforeSend`	De functie die uitgevoerd moet worden voordat de aanroep plaats vind. Ik gebruik het om de text "Busy" te tonen. Dit is gedaan met een Fluent notatie. En ik leeg het te vullen resultaten div-je met .empty(). Dit verwijdert alle tekst en HTML elementen binnen die div.
`Success`	De functie die uitgevoerd moet worden als de Ajaxcall succesvol wordt uitgevoerd. Het aangeleverde resultaat (de myJsonResult) wordt hier als 'result' doorgegeven. Hiervan kunnen we direct de result.Succeeded uitlezen en daarna lopen we door de lijst van result.Persons via de `$.each()`. Hier zal voor ieder item een functie uitgevoerd worden waarbij de 'this' het betreffende 'person' voorstelt.
`error`	Deze functie wordt uitgevoerd als er onverhoopt een probleem optreedt tijdens de communicatie. Dit komt nog wel eens voor als de verkeerde actie aangeroepen wordt of als de verkeerde parameters meegegeven worden. Dit zijn dus technische problemen!

Voer nu deze code uit. Zie dat de velden gevuld zijn. Pas eventueel de inhoud aan en druk op de Start Callback URL. We komen nu op de server waar de '*sleep*' even op zich laat wachten. Na terugkomst worden de namen getoond en zal de tekst "Ready" getoond worden (die daarna langzaam verdwijnt).

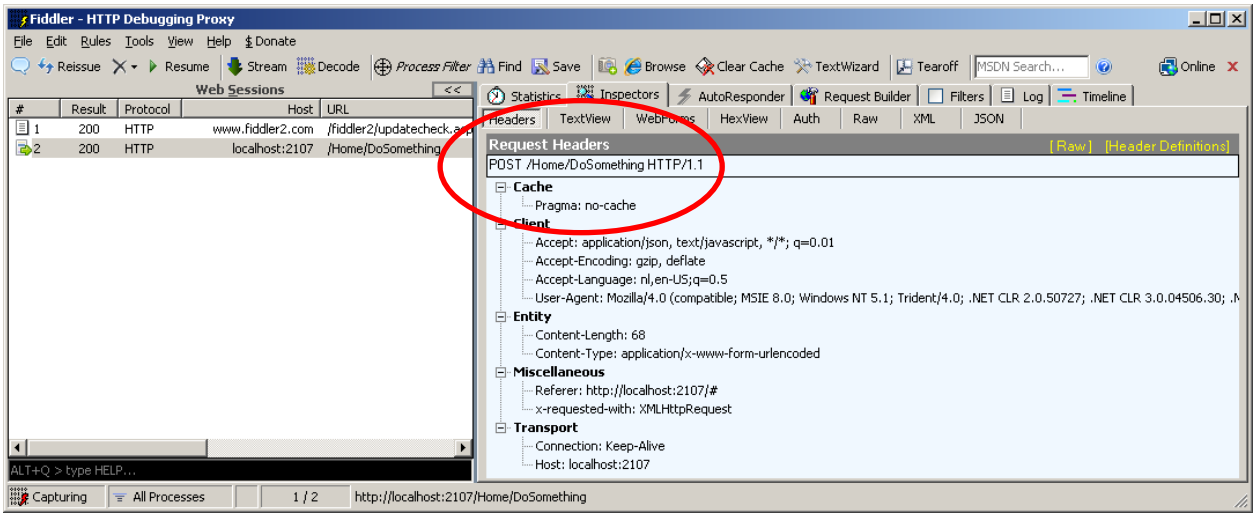


3.3 Testen

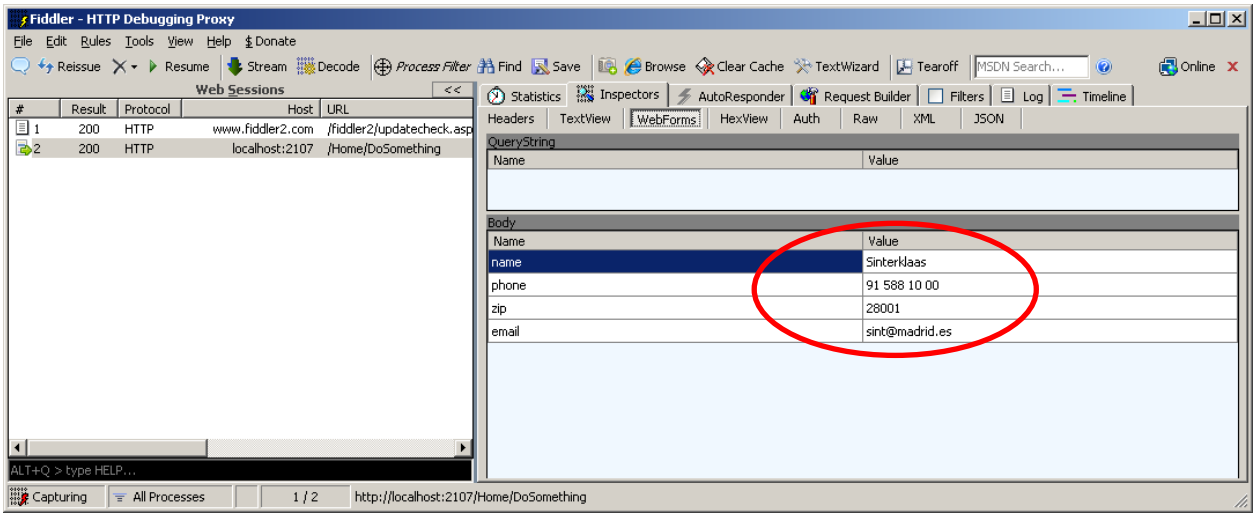
Het is een goede gewoonte om Ajax logica te bouwen EN te testen samen met een tool als Fiddler (moderne browsers hebben dezelfde functionaliteit ook ingebouwd; Probeer eens F12 in jouw Browser ☺). Fiddler werkt als een extra lokale proxy waar al het webverkeer van de browser doorheen gevoerd wordt. Het is dus perfect om hiermee te 'luisteren' naar de vragen van de browser en de antwoorden van de server. Kijk maar eens mee wat Fiddler weet te vertellen over deze callback.

3.3.1 Request

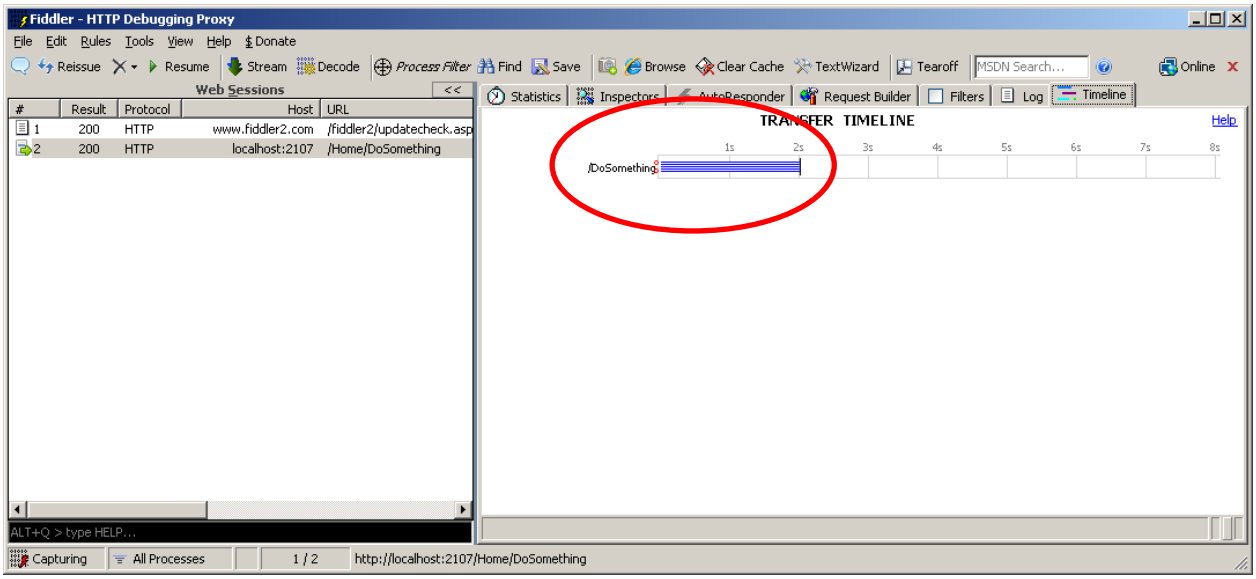
De call was een POST naar /Home/DoSomething:



De meegegeven waarden zijn de volgende:

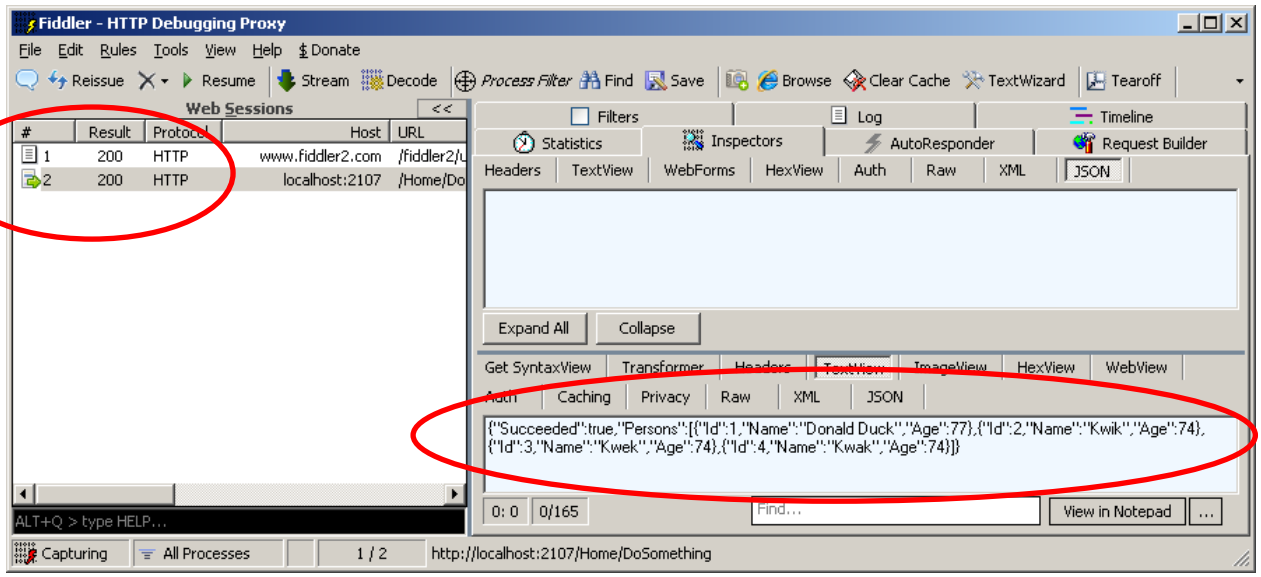


De call duurde zo'n 2 seconden (vanwege de '*sleep*' op de server):

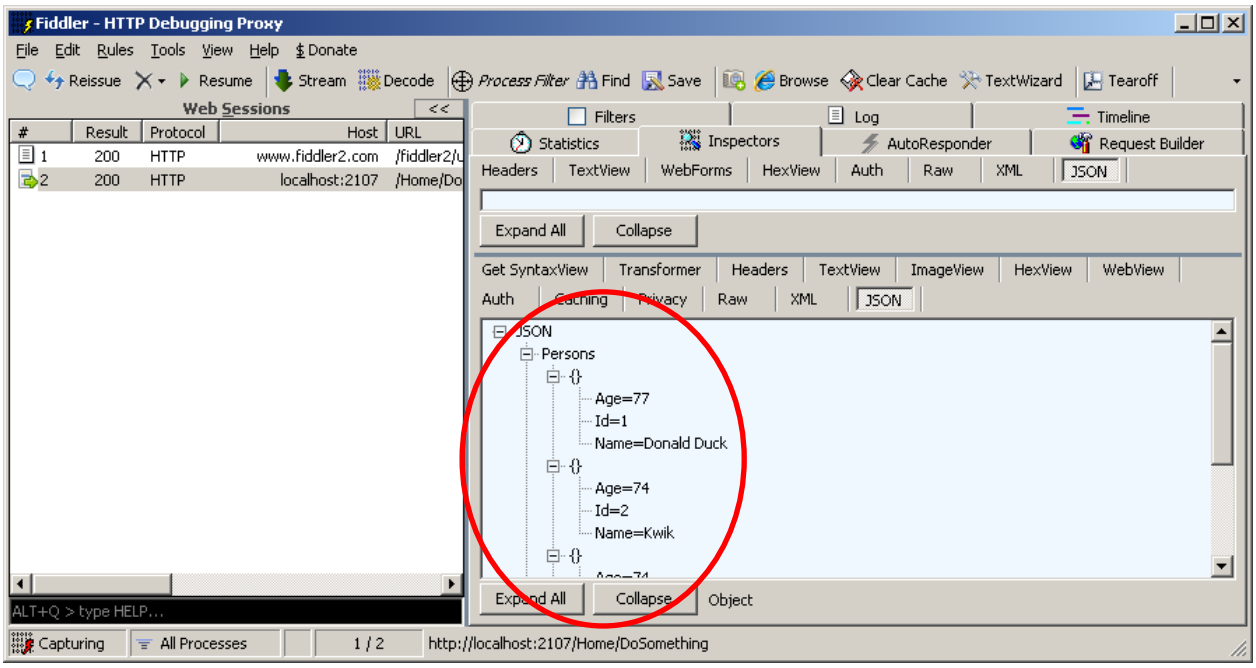


3.3.2 Response

Dat de call met een 200 terug kwam was al zichtbaar. Hier zien de we ruwe Json terugkomen:



Voor complexere constructies is ook een andere notatie mogelijk:



Conclusie

Het uitvoeren van een asynchrone callback is zeer eenvoudig geworden met de combinatie Asp.Net MVC3 en jQuery. Test wel wat er over de lijn gaat en test goed op alle omstandigheden.

Zie voor meer informatie over de .ajax():

http://api.jquery.com/category/ajax/

Tip: Voor de gevorderden onder ons, zijn er oo de $.Get() en $.Post(). Dit zijn vereenvoudigde notaties voor dezelfde call maar hebben als voordeel dat je minder typefouten zou kunnen maken.

Extra opdracht 1:

Probeer ook eens met de code te spelen zodat ook de error functie en de andere situatie in de Success functie optreden

Extra opdracht 2:

doe ook eens een GET

Extra opdracht 3:

Maak van de link eens een button om op te klikken

4. jQuery UI

We weten inmiddels hoe de basis van jQuery werkt. We kunnen DOM elementen en attributen aanpassen en asynchroon gegevens ophalen van een server. Nu wordt het tijd voor het echte werk.

We gaan uitgebreide UI controls gebruiken die in de browser draaien. We willen ten slotte het gevoel krijgen alsof we met een echte Windows applicatie aan het werken zijn.

We beginnen ook hier weer met de basis. En wat is een dedicated Windows applicatie zonder Error box??? ☺

4.1 Scripts

jQuery heeft al een heleboel werk voor je gedaan en biedt daarom al verschillende UI controls aan. Hiervoor moeten we natuurlijk wel eerst een jQuery JavaScript bestand en een jQuery CSS bestand inlezen.

Net als het eerste hoofdstuk hebben we de volgende zaken nodig:

1. Zorg dat jQuery UI aan je project toegevoegd is. Dit is standaard al het geval bij een MVC3 project want die heeft de betreffende Nuget package al geïnstalleerd.
2. Maak een controller aan genaamd UIController. Deze bevat standaard een Index Action
3. Maak een view aan genaamd Index.cshtml in de map /Views/UI
4. Maak in de /scripts map een jQueryUI.js aan
5. Verwijs vanuit de index.cshtml naar de advanced.js
6. Corrigeer de route die standaard gebruikt wordt door de Global.asax.cs te openen en daar de volgende regel code:

```
new { controller = "Home", action = "Index", id = UrlParameter.Optional } // Parameter defaults
```

Te vervangen door:

```
new { controller = "UIController", action = "Index", id = UrlParameter.Optional }
```

Tip: Je kunt zelf ook een 'lichtgewicht' jQuery UI uitvoering samenstellen op hun site. Dit is relevant als je heel specifiek maar enkele zaken wilt gebruiken. Dit voorkomt dat er onnodig veel bestanden over het internet gestuurd worden. Het is dan wel een extra afhankelijkheid voor onderhoud.

Laad de JavaScript en CSS in door de volgende regels op te nemen bovenaan in de _Layout.cshtml:

```
…
<link href="@Url.Content("~/Content/themes/base/jquery.ui.all.css")"rel="stylesheet" type="text/css" />
<script src="@Url.Content("~/Scripts/jquery-ui-1.8.16.min.js")" type="text/javascript"></script>
…
```

We hebben nu de jQuery UI JavaScript library en de bijbehorende CSS ingelezen. Nu kunnen we echt aan de slag!

Voeg de volgende HTML toe aan de index.cshtml:

```
<h2>JQuery UI</h2>

<div class="clear"></div>
  <button id="openDialogButton" type="button">Open dialog</button>

  <div id="myDialog">
      <input type="text" id="myDate" />
  </div>

  <div id="dialogState">
</div>

<script src="@Url.Content("~/Scripts/jQueryUI.js")"
        type="text/javascript"></script>
```

Zie dat hier de verwijzing naar jQueryUI.js is opgenomen.

Voeg de volgende code toe aan de jQueryUI.js:

```
//Zodra je op internet gaat zoeken kom je al vaker de volgende nieuwe korte notatie tegen
//Deze notatie wordt op dit moment het meeste gebruikt.

$(function () {
    ExecuteOnStartup();
});

//extra: Onder het motto: Kort-korter-korst kun je ook nog het volgende gebruiken:
//$(ExecuteOnStartup);

function ExecuteOnStartup() {
 // hier komt jouw JavaScript code
}
```

Hier gaan wij zo meteen in werken. Bekijk eens de eerste paar regels, waar we het starten van het script iets verkort opschrijven.

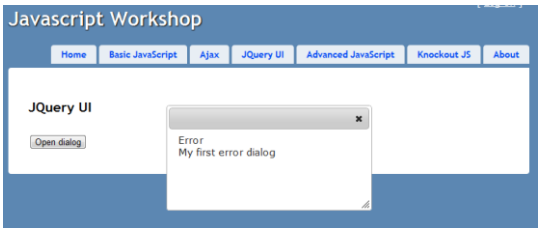
4.2 Dialog

Ok, laten we nu eens onze pagina interactief en mooi gaan maken!
We gaan een element omtoveren tot een dialog-box die je op je scherm gaat zien.

De HTML is al voorbereid zoals je had gezien. Ga nu naar het JavaScript bestand (jQueryUI.js) en voeg aan de functie: ExecuteOnStartup de volgende regel toe:

```
$('#myDialog').dialog();
```

Start de applicatie en zie al direct het resultaat:



Dankzij de UI library van jQuery hebben we nu een standaard *dialog* plugin tot onze beschikking, die van een opgegeven DOM element op je pagina een dialog maakt. Dit alles door alleen de .dialog() functie aan te roepen.

Maar je zag al dat het dialog nog erg kaal is. Zo had het dialog geen titel en misten we de "Ok" knop om ons foutmelding dialoogje te sluiten. En het dialog ging automatisch open, dit willen we natuurlijk zelf regelen.

Met andere woorden, we willen onze dialog configureren met instellingen.

Maak vooraan in ExecuteOnStartup een config object aan en geef deze mee aan de initialisatie van het dialog. Dit doe je door de volgende code boven de initialisatie van het dialog te zetten:

```
var config = {
        autoOpen: false,
        resizable: false,
        draggable: false,
        modal: true,
        height: 500,
        title: 'My dialog title',
        buttons: { Ok: function () { $(this).dialog("close"); } }
    };
```

We hebben nu een config object aangemaakt met een aantal properties die allemaal redelijk voor zichzelf spreken. De property "buttons" bevat weer een nieuw object (met properties, in dit geval *Ok*). Als je gaat debuggen zul je dit ook zien. In het geval van de "ok" button hebben we er geen 'primitieve waarde' maar een hele functie aan toegekend. Deze wordt uitgevoerd zodra op de "ok" button wordt geklikt.

We gaan nu eerst de config meegeven aan de initialisatie van het dialog. En we koppelen het openen van het dialog aan de button-click van de button die op het scherm staat:

```
…
$('#myDialog').dialog(config);
$('#openDialogButton').click(function () { $('#myDialog').dialog('open'); });
…
```

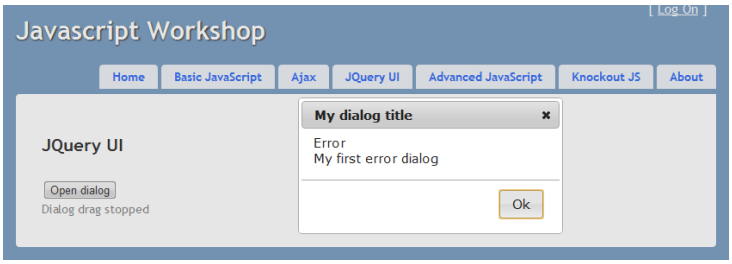
Voer dit eens uit. Zie dat het dialoog pas getoond wordt als de klik uitgevoerd wordt.

Een dialog genereert ook verschillende events. Hier kun je je op abonneren en daar kun je voordeel uit halen. We gaan in dit geval de status van het dialog laten zien op de pagina net onder de button.

Voeg nu de volgende regels code toe om de events af te vangen:

```
$('#myDialog').bind('dialogopen', function (){ $('#dialogState').html('Dialog opened'); });
$('#myDialog').bind('dialogdragstart',function() { $('#dialogState').html('Dialog drag started'); });
$('#myDialog').bind('dialogdragstop', function() { $('#dialogState').html('Dialog drag stopped'); });
$('#myDialog').bind('dialogclose', function () { $('#dialogState').html('Dialog closed'); });
```

Start de applicatie en bekijk het resultaat:



***Tip* 1:** Nu abonneren we ons na de initialisatie pas op de events, maar zoals je bij de ok button al had gezien kun je het ook gewoon in de config opnemen. De configuratie zou er dan als volgt uit komen te zien:

```
var config = {
        autoOpen: false,
        resizable: false,
        draggable: true,
        modal: true,
        title: 'My dialog title',
        buttons: { Ok: function () { $(this).dialog("close"); } },
        open: function () { $('#dialogState').html('Dialog opened'); },
        close: function () { $('#dialogState').html('Dialog closed'); },
        dragStart: function () { $('#dialogState').html('Dialog drag started'); },
        dragStop: function () { $('#dialogState').html('Dialog drag stopped'); }
    };
```

***Tip* 2:** Voor een mooi overzicht van alle functionaliteit over de dialog zie:
http://jqueryui.com/demos/dialog/

4.3 Datepicker

Nu we hebben gezien hoe eenvoudig het is om een dialog te tonen gaan we kijken wat jQuery UI nog meer voor ons in petto heeft!

We gaan nu aan ons dialog een Datepicker toevoegen. Hiervoor moeten we een stukje HTML toevoegen en een extra regel JavaScript.

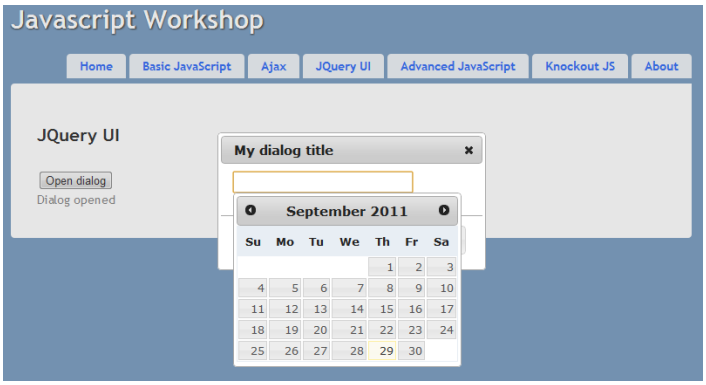
Maak de inhoud van de met id:"myDialog" leeg.
Zet daar een input element van het type tekst in met het id: "myDate"

```
<input type="text" id="myDate" />
```

Voeg onderaan in de ExecuteOnStartup de initialisatie van de Datepicker toe:

```
…
$('#myDate').datepicker();
…
```

Start de applicatie, open de dialog en zie het resultaat:



Dus door letterlijk één regel code toe te voegen kunnen we van een tekst vak een Datepicker maken. Ook hier geldt hetzelfde als voor het dialog: we kunnen de Datepicker helemaal configureren door options mee te geven en/of ons te abonneren op een event.

Als je een datum selecteert zie je dat hij niet de standaard Nederlandse notatie gebruikt, ook niet als dit in de browser is ingesteld. Verder zagen we dat de kalender standaard op zondag begint. Dit gaan we her-configureren.

We gaan dus nu de firstDay en de datum-notatie wijzigen. Om onnodige regels code te gebruiken gaan we nu het config object *inline* aanmaken en vullen. We kunnen dan de initialisatie die we zojuist hadden toegevoegd aan het JavaScript bestand vervangen met de volgende code:

```
$('#myDate').datepicker({ firstDay: 1, dateFormat: 'dd-mm-yy' });
```

Start de applicatie en bekijk het resultaat:

Ook voor de Datepicker geldt dat het veel meer mogelijkheden heeft. Daarvoor kun je alle demo's bekijken en de technische documentatie raadplegen op: http://jqueryui.com/demos/datepicker/

4.4 Buttons

Tegenwoordig gebruiken we MVC3 wat al een modern uiterlijk heeft, we hebben daar mooie dialogen en date pickers aan toegevoegd, maar nu zien onze buttons er nog steeds een beetje eenvoudig uit. Ook hier heeft jQuery UI wat aan gedaan. En we op een héél erg makkelijke manier.

4.4.1 Standaard button met icon

We beginnen met het stylen van de button die al op de pagina staat. We gaan daar een mooiere button van maken met een icon.

Hiervoor gaan we weer naar het "jQueryUI.js" bestand en voegen daar de volgende regels code aan toe:

```
$('#openDialogButton').button({
        icons: {
            primary: "ui-icon-info",
            secondary: "ui-icon-notice"
        }
    });
```

Laad de pagina en zie dan het resultaat:

Je ziet nu dat die ene button er mooi gestyled uitziet met twee verschillende icons. Je kunt dus een icon voor de tekst EN achter de tekst zetten. Kies maar wat je mooi vindt. We hebben nu dus de button gestyled door alleen de functie "button()" aan te roepen.

Omdat we aan deze functie een configuratie object hebben meegegeven met één property "icons" toont jQuery automatisch de BEIDE icons die meegegeven zijn.

De plaatjes zijn al aanwezig doordat we jQueryUI binnen ons project hebben geplaatst. Deze wordt standaard uitgebreid met allerlei CSS en images. Dit is dus aan te passen.

Onderaan op de themeroller site kun je alle icons zien die beschikbaar zijn: http://jqueryui.com/themeroller/#themeGallery . Zodra je met je muis over een van de items hovert zie je de naam verschijnen die je aan het configuratie object kunt meegeven.

Tip: Het stylen van een button werkt zowel op het <input type="button"> HTML element als op <button> HTML5 element. Maar het **toevoegen van icons** werkt **alleen** maar op **<button> elementen**! **De volgende regel code styled in één keer alle button elementen in je hele pagina:**

```
$("button").button();
```

Hier staat dat voor alle (op dat moment bestaande) buttons binnen de hele DOM, de functie .button() uitgevoerd moet worden. En dit zorgt voor een nette styling uit de jQueryUI CSS voor al die buttons. Die referentie naar de icons hierboven was dus een extraatje.

Dit zorgt ervoor je één uniforme look krijgt op je pagina's.

4.4.2 Radio buttons

Een andere veel gebruikte button is de radiobutton. Ook deze willen we van een mooi jasje voorzien. We beginnen eerst met een aantal radio buttons toe te voegen aan de web pagina.

Voeg de volgende regels toe aan de "Index.cshtml" tussen de <h2> en <button> elementen:

```
…
<div id="radio" style="float: left">
   <input type="radio" id="radio1" name="radio" checked="checked" /><label for="radio1">Fade out</label>
   <input type="radio" id="radio2" name="radio" /><label for="radio2">Bounce</label>
   <input type="radio" id="radio3" name="radio" /><label for="radio3">Explode</label>
</div>
…
```

Start nu de pagina en zie het volgende resultaat:

Dit is niet onaardig maar het past niet in de rest van het design. We willen namelijk drie knoppen die elkaars keuze uitsluiten (de selectie van de ene deselecteert de vorige andere keuze). Om dit op te lossen gaan we ook hier een styling op toepassen.

Zoals je kunt zien in de HTML die we zojuist hebben toegevoegd hebben we de radiobuttons gegroepeerd in één div met het id "radio".

Dit hebben we met voorbedachten rade gedaan, want die hebben we nodig voor jQuery.

Voeg nu de volgende regel code onderaan in de "ExecuteOnStart" functie:

```
$( "#radio" ).buttonset();
```

Herlaad de pagina en zie meteen dat het er mooi en gelikt uitziet:

4.5 Animations / effects

Yeah, let the party begin!

We gaan onze applicatie nog mooier maken door charmante animaties toe te voegen. Let wel op dat je er geen kitscherige boel van gaat maken ;-) Maar voor deze demo gaan we dat uiteraard wel doen! Het leven is tenslotte een zuurstok!

We gaan eerst een extra knop en een extra HTML element toevoegen waar we de animaties op los gaan laten.

Voeg eerst DIRECT achter de radio buttons een animation button toe:

```
…
<button id="animationButton" type="button" style="float:left; margin-left: 10px;">Animate</button>
…
```

Helemaal onderaan de pagina voegen we nog een div toe die we gaan gebruiken voor animaties.

```
…
<div id="animationDiv"

     style="clear:both;
     margin-top: 20px;
     padding: 5px;
     border: 1px solid black;
     width:200px;
     height:200px;
     background-color: lightblue;">
    Op deze div gaan we animaties los laten
</div>
…
```

Tip: De styling doe ik hier direct in de HTML. Wil je het echt netjes doen, dan verplaats je de styling natuurlijk naar een CSS bestand.

Bekijk nu het resultaat:

Hé, de button "Animate" bevat nog geen styling. Pas nu eerst de tip uit de paragraaf 3.4.1 toe.

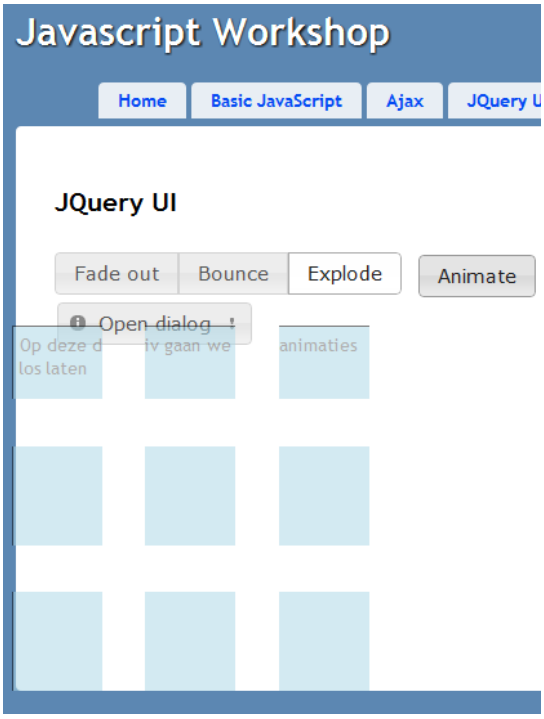
En voeg daarna de volgende code toe aan het einde van de ExecuteOnStart functie toe:

```
…
$('#animationButton').click(animateDiv);
…
```

En voeg onderaan het JavaScript bestand (dus buiten 'ExecuteOnStart') de volgende functie toe:

```
function animateDiv() {
    var selectedEffect = '';
    var checkedButton = $(':checked', '#radio');

    switch(checkedButton[0].id) {
        case 'radio1': selectedEffect = 'fade'; break;
        case 'radio2': selectedEffect = 'bounce'; break;
        case 'radio3': selectedEffect = 'explode'; break;
    }

    $("#animationDiv").effect(selectedEffect, {}, 500 );
}
```

En speel nu eens met de knoppen. Zie dat er verschillende animaties uitgevoerd worden:



De code spreekt eigenlijk wel voor zich. Met behulp van de :checked selector gaat jQuery het element zoeken die een attribuut met de waarde checked bevat. Hij zoekt alleen binnen de DOM elementen die child zijn van #radio.

Omdat ik zeker weet dat er minimaal één element altijd checked is (Het zijn radio buttons en ik selecteer er één bij default). Nu kan ik in het switch statement het eerste object aanspreken en daar het 'id' van opvragen.

Tot slot wordt het effect uitgevoerd, met een lege config en het aantal milliseconden. Ik heb nu bewust gekozen voor effecten die geen configuratie nodig hebben. Voor een lijst van alle effecten en de daarbij behorende configs kun je kijken op: http://jqueryui.com/demos/effect/#option-effect

4.1 Extra (optionele) opdrachten

Opdracht 1:

Geef andere icons weer op de "#openDialogButton" bij het selecteren van één van de radio buttons

Opdracht 2:

Laat het openen van het dialog gebeuren met een effect. (let op: dit kun je doen met behulp van de configuratie)

Opdracht 3:

Probeer bij de effecten eens het "size" effect te gebruiken (let op: hiervoor moet je dus een config gaan meegeven).

5. Advanced Javascript

5.1 Maak je eigen jQuery functie

We hebben nu al meerdere functies gebruikt van jQuery. Denk daarbij aan de $.ajax() of $('').button().
Maar hoe schrijf ik nu zelf een functie die we later vaker kunnen hergebruiken?
In deze paragraaf gaan we hier eens dieper op in.

Voer eerst de volgende stappen uit:

1. Maak een controller aan genaamd AdvancedController. Deze bevat standaard een Index Action
2. Maak een view aan genaamd Index.cshtml in de map /Views/Advanced
3. Maak in de /scripts map een advanced.js aan
4. Vul advanced.js met dezelfde basislogica als in hoofdstuk 4.1.
5. Verwijs vanuit de index.cshtml naar de advanced.js
6. Corrigeer de route die standaard gebruikt wordt door de Global.asax.cs te openen en daar de volgende regel code:

```
new { controller = "Home", action = "Index", id = UrlParameter.Optional } // Parameter defaults
```

Te vervangen door:

```
new { controller = "AdvancedController", action = "Index", id = UrlParameter.Optional }
```


Voeg eerst aan de Index.cshtml de volgende regel toe:

```
<a id="nupuntnl"  href="http://www.nu.nl">Nu</a>
```


En aan de ExecuteOnStart functie in "advanced.js" de volgende code:

```
$('#nuputnl').text('Public link');
```


En bekijk nu het resultaat:

We hadden verwacht dat er niet "Nu" zou staan maar "Public link"

Zie jij wat het probleem is?

Hierboven is de verwijzing naar het 'id' van de link niet correct over genomen.

In dit geval wisten we bij het schrijven dat we één en slechts één control wilden manipuleren. Met een klein beetje meer moeite had dit ook afgedwongen kunnen worden want nu hebben we een foutsituatie. We manipuleren nu GEEN element ipv. één element.

Wat is een oplossing?

Het is heel simpel om eigen jQuery functies te schrijven die .Net ontwikkelaars het beste als ExtensionMethods zullen herkennen. Deze zijn dan fluent uit te voeren op de $(..).

We schrijven nu een functie die piept als nul of meerdere objecten gevonden zijn. Voeg de volgende code toe onderaan het advanced.js (en **buiten** de **ExecuteOnStart** functie).

```
(function ($) {
    $.fn.onlyOne = function (mode) {
        var mode = mode || 'off';
        if (mode != 'off') {
            if (this.length == 1) {
                if (mode != 'on') {
                    alert('Information: one object found.');
                }
            }
            else {
                alert('Warning: not one but '
               + this.length + ' objects expected.');
            }
        }
        // do not break the normal flow;
        return $(this);
    };
})(jQuery);
```

Deze functie heet .onlyOne() en accepteert een string om de werking te manipuleren.

Tip: Normaal schrijf je zo'n functie in een apart JavaScript bestand die gerefereerd zal worden, maar voor nu is het zo ook voldoende.

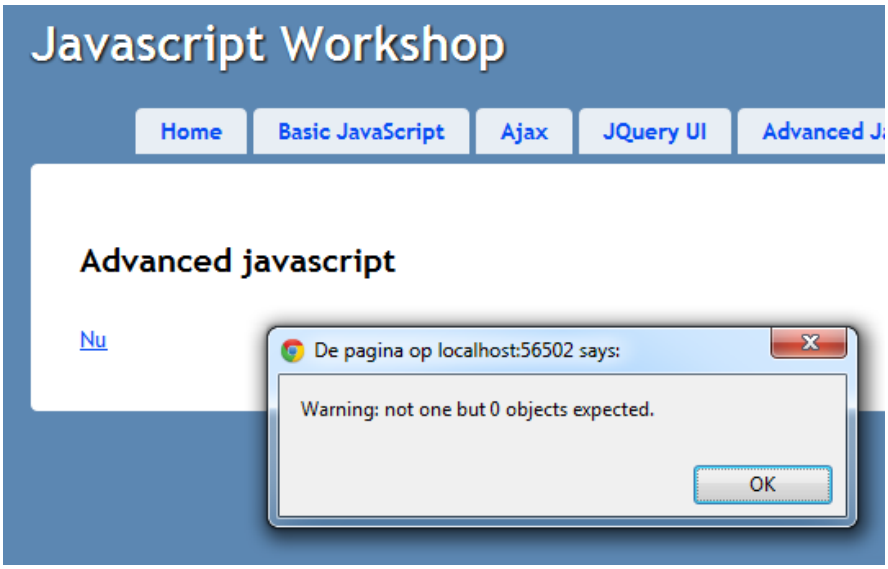
De (function($){ … })(jQuery); is bedoeld om ruzie met andere (jQuery) bibliotheken te voorkomen.

We gaan nu direct deze functie toepassen en breiden onze code uit met tot het volgende:

```
…
$('#nuputnl').onlyOne('on').text('Public link');
…
```

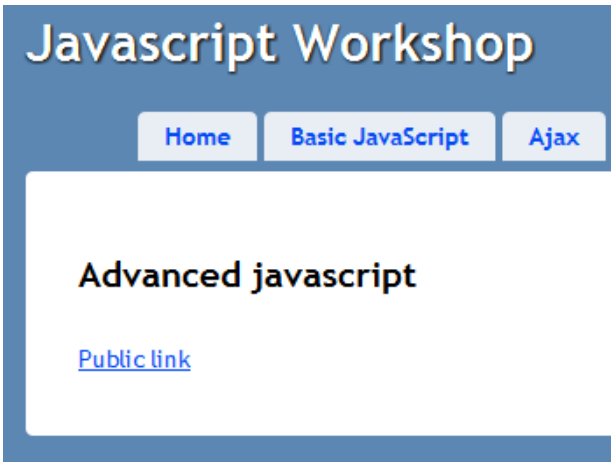
Binnen de onlyOne functie wordt veel gebruik gemaakt van `this`. Hier is `this` het resultaat van de selector $('#nuputnl'). Om de fluent notatie in stand te kunnen blijven houden retourneer je op het laatste altijd `$(this)` (of een bewerkte versie van die elementen daarvan als je een filter maakt).

Start de applicatie en zie nu het resultaat:



Corrigeer en test je code.

Je zou het volgende resultaat moeten zien:



Extra opdracht:

Herschrijf de functie zodat hij alleen een foutmelding geeft als er geen elementen zijn gevonden. Bij meerdere elementen moet hij goed gaan.

5.2 Creëer een JavaScript object

In de vorige hoofdstukken hebben we onze eerste functies geschreven binnen JavaScript. Maar als iedereen zomaar functies gaat schrijven die direct in de 'global' terecht komen zal je al heel snel zien dat het erg onoverzichtelijk wordt en je functies zomaar gaat overschrijven. Dit is niet echt handig!

We willen structuur in onze code aanbrengen maar hoe? JavaScript is geen *Object Oriented* taal? In plaats van **Klasse** gebaseerd programmeren noemen ze constant **Prototype**-gebaseerd programmeren, maar wat is dat?

Wat zegt Wikipedia over Prototype-gebaseerd programmeren:

> **Prototype-gebaseerd** programmeren is **een vorm van object-georiënteerd programmeren** waarin **geen klassen** aanwezig zijn. In plaats van klassen te gebruiken, wordt een **object kopie** gemaakt van bestaande objecten, welke dienen als prototypen. Dit model wordt ook wel klasse-loos, 'prototype-georiënteerd' of 'instantie-gebaseerd' programmeren genoemd. Prototyping wordt mogelijk gemaakt door talen die delegation ondersteunen.

Een vorm van object georiënteerd? Geen klassen? Object, kopie?

We gaan een functie maken waarin we variabelen en functies gaan gebruiken. Van deze functie kunnen we dan straks een object maken zoals we gewend zijn in C#. We krijgen dan toch een beetje het gevoel alsof we netjes met classes aan het programmeren zijn.

Laten we snel aan de slag gaan en kijken hoe het werkt.

We beginnen met het definiëren van een function met de naam "Person" buiten de ExecuteOnStart methode. Deze function gaan wij gebruiken als onze class waarvan we objecten gaan maken.

```
function Person() {

};
```

Nu gaan we aan die functie Person eens eerst twee private variabelen toevoegen genaamd: firstName en lastName.

```
function Person() {
    var firstName = 'Bill';
    var lastName = 'Gates';
};
```

Private variabele en functions beginnen altijd met `var`. Dat betekent dat het object alleen leeft en alleen bekend is **binnen** de function Person.

Maar hoe kunnen we dan public variabele aanmaken? Dat doe je door `this` te gebruiken.
We voegen nu een public property MyName toe.

```
function Person() {
    var firstName = 'Bill';
    var lastName = last;
    this.MyName = firstName + ' ' + lastName;
};
```

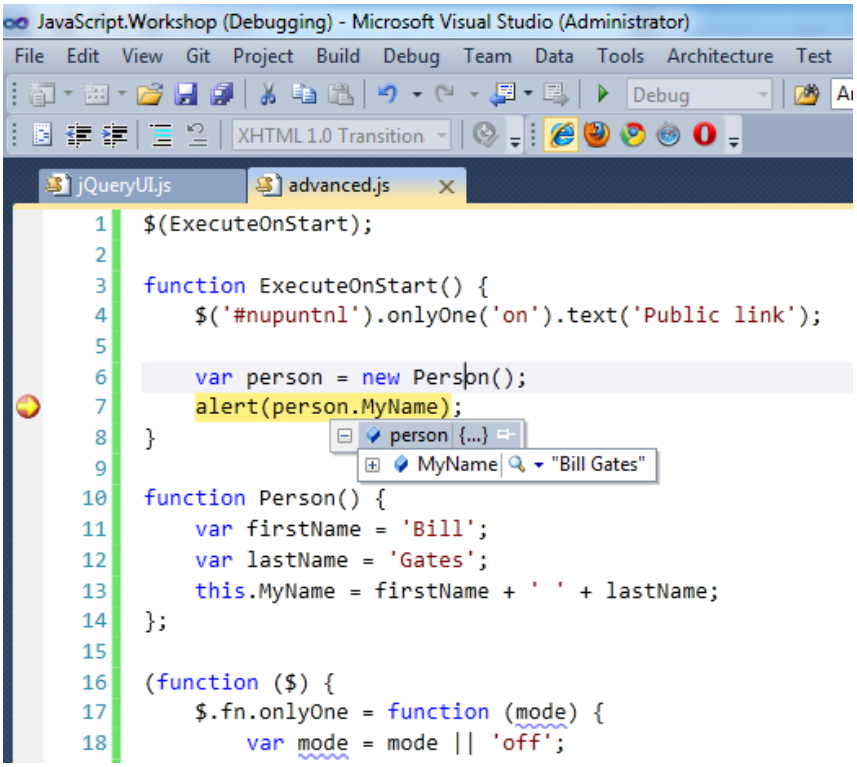
Tot zo ver de aanmaak van de Klasse…

Nu gaan we proberen om er een instantie van aan te maken en deze te gebruiken.
Instantieer een object achteraan in de ExecuteOnStart methode en gebruik daarna alert() om de naam weer te geven.

```
var person = new Person();
alert(person.MyName);
```

Start de applicatie en bekijk het resultaat:

Zet een break-point op de alert regel en bekijk het object eens met de debugger.



Je ziet nu ook echt alleen de public variabele die we aangemaakt hebben!

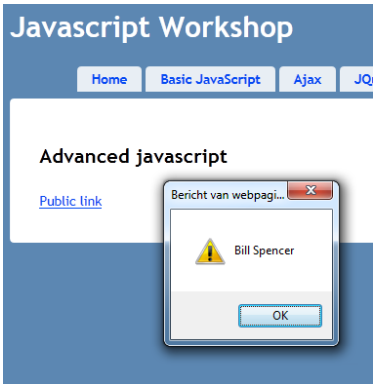
Zo werkt dat ook (bijna) met interne functies.
We voegen nu een aantal functies binnen de function Person() toe:

```
var fullName = function () {
    return firstName + ' ' + lastName;
};

this.ChangeLastName = function (newLastName) {
    lastName = newLastName;
    this.MyName = fullName();
};
```

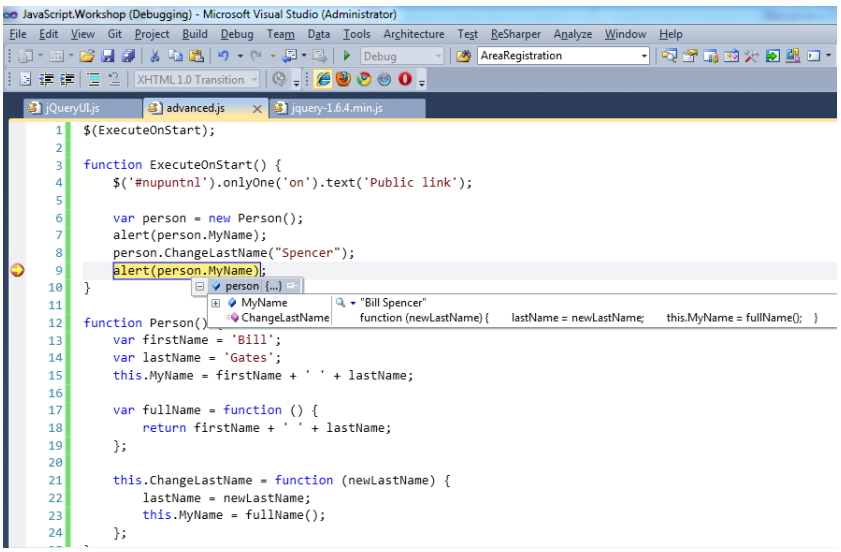
De public function ChangeLastName gaan we dan natuurlijk ook aanroepen.
Dus voeg deze toe aan de ExecuteOnStart na de vorige alert()

```
person.ChangeLastName("Spencer");
alert(person.MyName);
```

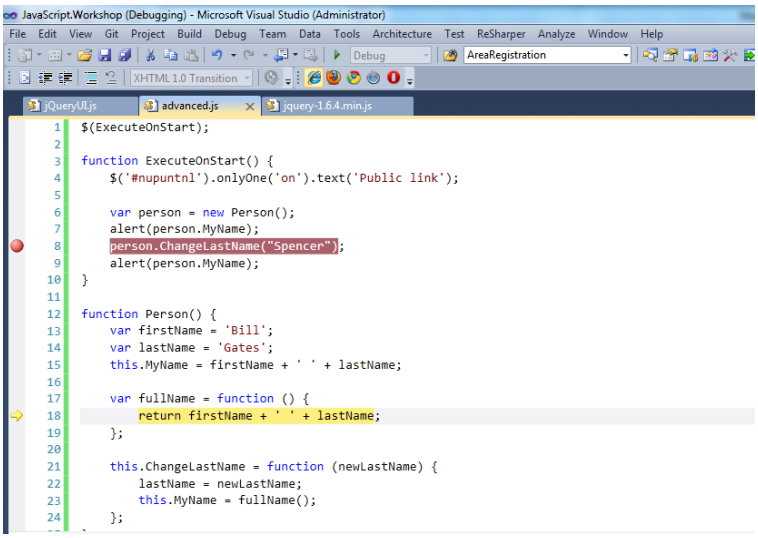
Zodra je de code nu uitvoert zal de 2e alert de volgende melding tonen:



En als we weer gaan debuggen en het object dan bekijken zien we dat er één functie is bijgekomen die we publiekelijk kunnen zien.



De private functies worden hier in de debugger ook niet genoemd, maar als je gaat steppen met F11 step je er wel doorheen.



Ok, nu zijn we al een heel eind. We hebben nu public en private variabelen en functies. Het enige waar we nu nog tegenaan lopen is dat de private functies geen gebruik kunnen maken van de public functies en variabelen. Probeer maar eens MyName aan te spreken binnen de functie fullName. Dat gaat niet werken zoals je zou willen.

Hoe lossen we dit probleem nu op? Voer het setten van public variabelen in publieke functies uit, zoals wij in het voorbeeld ook gedaan hebben met het bijwerken van `this.MyName`.

Stel dat het setten van een publieke variabele in private functie noodzaak is, dan is daar een oplossing voor, maar daar hebben we een referentie naar het eigen object voor nodig. Dan kun je een call maken naar zijn eigen public variabelen. In de volgende paragraaf kun je zien hoe je eenvoudig een referentie naar je eigen object toevoegt.

Extra informatie: Er zijn meerdere mogelijkheden om functies publiekelijk te maken. In dit voorbeeld hebben we `this` gebruikt, omdat we het dan expliciet zichtbaar maken. Een andere methode is om ze als private the declareren (dus met een `var`) en ze daarna op het einde te retourneren. In ons voorbeeld zou het er dan zo uit komen te zien

```
function Person() {
    var self = this;
    var firstName = 'Bill';
    var lastName = last;
    this.MyName = firstName + ' ' + lastName;

     var fullName = function () {
        return firstName + ' ' + lastName;
    };

    var changeLastName = function (newLastName) {
        lastName = newLastName;
        //this.MyName = fullName();
    };

    return {
        ChangeLastName: changeLastName
    };
};
```

Je definieert dan een publieke functienaam en koppelt deze aan een interne functie (zie `return`).
Let wel dat het setten van `this`.MyName dan niet meer mogelijk is omdat de functie intern private is geworden. Deze heb ik daarom dan ook in commentaar gezet.

5.3 Communicatie tussen objecten met behulp van events

We hebben nu onze eigen classes geschreven maar hoe past JQuery in dit plaatje?

We gaan nu kijken hoe we gebruik kunnen maken van jQuery Events om objecten met elkaar te laten communiceren.

Om een object events te kunnen laten produceren heeft het een referentie naar zichzelf nodig. Dit is eenvoudig op te lossen.

Bij het initialiseren gaan we de `this` variabele opslaan.
Voeg daarom bovenaan in de function Person de volgende regel toe:

```
function Person() {
    var self = this;
…
```

Nu hebben we dus `this` opgeslagen in een private variabele genaamd self. Deze gaan we straks gebruiken. Tijdens het constructen van een nieuw object, zal de variabele 'self' met de juiste referentie gevuld worden.

Voor communicatie hebben we uiteraard twee verschillende objecten nodig. We gaan zo meteen een object van het type Person laten communiceren met een object van het type Dog.

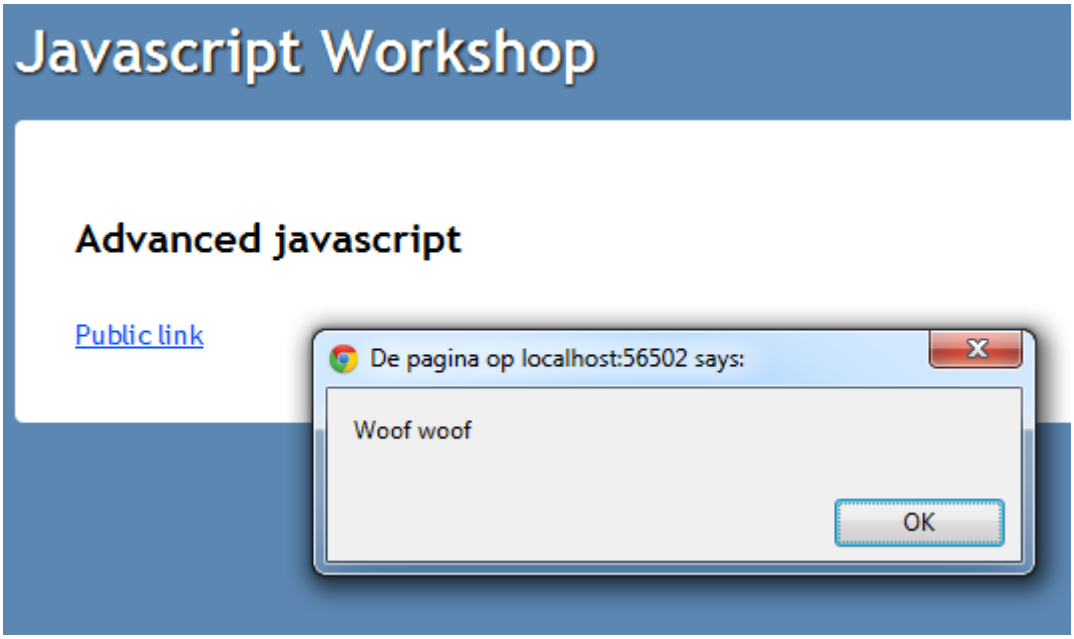
Neem de code van Dog over en plaats die achter Person:

```
function Dog() {
    var self = this;
    var state = 'Barking';
    var timer = null;

    var loop = function () {
        if (state == 'Barking') {
            alert('Woof woof');
        }
        setTimeout(loop, 2500);
    };

    var listenToWord = function (speaker, word) {
        switch (word) {
            case 'Quiet': state = 'Quiet'; break;
            case 'Bark': state = 'Barking'; break;
        }
    };

   this.Listen = function (boss) {
        $(boss).bind('shout', listenToWord);
    };
     //Start living
    loop();
};
```

En voeg achteraan de functie "ExecuteOnStart" de initialisatie van een Dog object toe.

```
var dog = new Dog();
```

Start nu de applicatie en bekijk het resultaat:



De hond blijft blaffen!

Als je de code bestudeert, zie je dat ook Dog een referentie naar zichzelf bevat. Dog bevat verder een aantal functies. Zolang de state van Dog op "Barking" staat zal hij iedere 2,5 seconde één keer blaffen.

In de listenToWord methode luistert de hond naar de twee woorden: "Bark" en "Quiet" welke zijn huidige staat kunnen wijzigen.

Maar het belangrijkste zit hem in de publieke Listen functie. Hier geef je een person object genaamd "boss" aan mee waar de Dog naar zal gaan luisteren.

Met behulp van de volgende regel code: **"$(boss).bind('shout', listenToWord);"** abonneer je je op het **"shout"** event van de boss. Zodra bij de boss het **"shout"** event afgevuurd wordt, zal de functie listenToWord aangeroepen worden.

We hebben nu dus een Dog die in staat is om te luisteren naar een boss, maar uiteraard moeten we dit nog wel met code regelen.

Voeg daarom de volgende publieke functie toe, aan de Person class:

```
this.Shout = function (word) {
    $(self).trigger('shout', word);
};
```

Met behulp van de **".trigger"** functie kun je een event afvuren. Hier vuurt de persoon op zichzelf het **"shout"** event af.

We hebben nu dus al een Dog die kan luisteren en een Person die kan schreeuwen, nu hoeven we alleen deze twee objecten aan elkaar te verbinden en ervoor te zorgen dat het person object ook daadwerkelijk gaat schreeuwen.

Voeg nu eerst de volgende regels code toe aan de Index.cshtml

```
<div class="clear"></div>
The dog listens to the following words: Quiet, Bark
<button id="shoutQuiet" style="margin-left: 10px;">Shout Quiet</button>
<button id="shoutBark" style="margin-left: 10px;">Shout Bark</button>
```

En in het JavaScript bestand in de "ExecuteOnStart" functie voegen we de volgende regels code toe:

```
$('#shoutQuiet').click(function () { person.Shout('Quiet'); });
$('#shoutBark').click(function () { person.Shout('Bark'); });
dog.Listen(person);
```

Start nu de applicatie en test met behulp van de knoppen of je de hond ook stil krijgt.

Extra opdracht:

Laat de hond een event triggeren zodra zijn status wijzigt.
Plaats een op het scherm waar je de status van de hond in gaat weergeven.
Gebruik dan de **.bind** functie om de nieuwe status van de hond weer te geven in de span.

6. MVVM met Knockout JS

6.1 Model View ViewModel

Theoretische uitleg is hier te vinden: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

Praktische uitleg:
Het basis idee achter MVVM is dat er een model (class object met properties) is waar de voorkant (pure UI) aan gekoppeld wordt. In de code gaan we de UI niet wijzigen zoals we dat uit oude technieken gewend zijn, maar we wijzigen properties op ons model. De UI is dan alleen gekoppeld aan deze properties en het framework zorgt er dan voor dat de UI wordt bijgewerkt zodra een van de properties wijzigen.
Voorbeeld:
Doel: Je hebt een textbox die je visible wilt maken nadat er op een knop is gedrukt.
Oplossing:
Op het model maak je dan een TextboxVisible property van het type boolean. Op UI niveau koppel je het Visible property van het tekstbox aan de TextboxVisible boolean van het model.
Je abonneert je op het click event van de desbetreffende knop. In het click-event wijzig je de boolean TextboxVisible op het model naar True. Het framework zal dit detecteren omdat deze de property in de gaten houdt en zal dan de textbox zichtbaar of onzichtbaar maken.

Het lijkt een beetje op toveren, maar laten we het eens gaan toepassen, misschien dat je dan bovenstaand stukje beter begrijpt.

6.2 Eenvoudige bindings

Laten we eens beginnen met het zojuist omschreven voorbeeld uit te werken met KnockOutJS.

Voer eerst de volgende stappen uit:

1. Maak een controller aan genaamd KnockoutController.
   Deze bevat standaard een Index Action
2. Maak een view aan genaamd Index.cshtml in de map /Views/Knockout
3. Maak in de /scripts map een myKnockout.js aan
4. Vul de myKnockout.js met dezelfde basislogica als in hoofdstuk 4.1.
5. De verwijzing naar het js bestand gaan we zometeen maken!
6. Corrigeer de route die standaard gebruikt wordt door de Global.asax.cs te openen en daar de volgende regel code:

```
new { controller = "Home", action = "Index", id = UrlParameter.Optional } // Parameter defaults
```

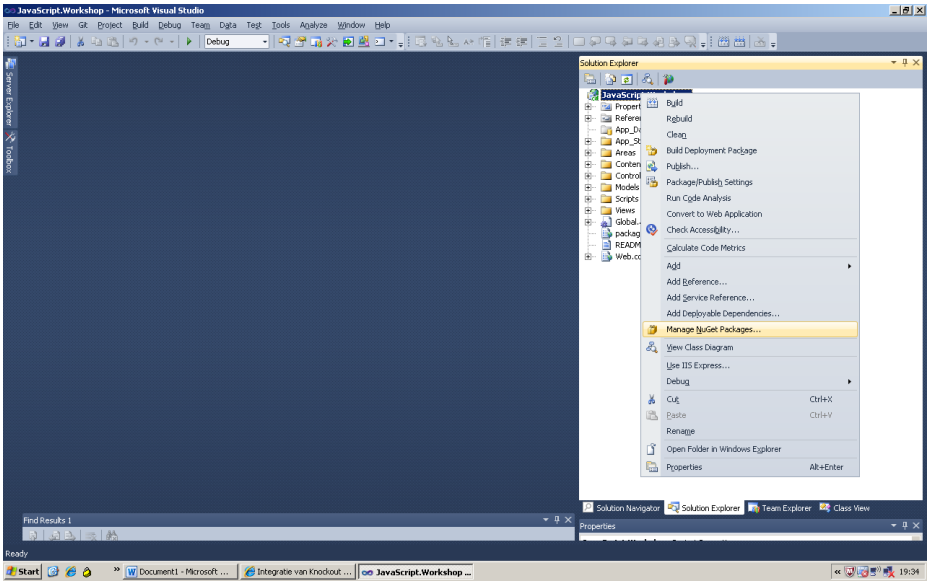
Te vervangen door:

```
new { controller = "KnockoutController", action = "Index", id = UrlParameter.Optional }
```

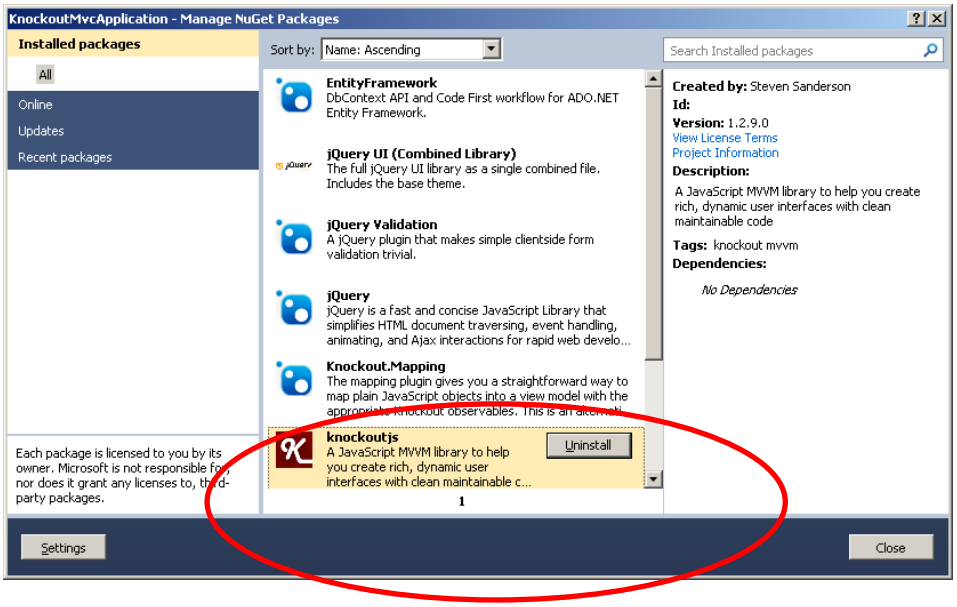
6.2.1 Nuget packages ophalen en _layout.cshtml bijwerken

De volgende Asp.Net MVC3 stappen zijn de voorbereiding op deze oefening. Om Knockout goed te demonstreren moeten we die basis neerzetten. We gaan er vanuit dat de voorgaande hoofdstukken en/of ervaring met Asp.Net MVC3 voldoende zijn.

Voeg eerst aan je Asp.Net MVC3 project twee Nuget packages toe. Geef dus een rechtermuisklik op het project:



Selecteer zowel het package **Knockoutjs** als het package **Knockout.mapping**:



Alternatief: Stel dat je nuget niet geïnstalleerd hebt, dan is het verstandig om dit eerst te doen! Maar uiteraard is het mogelijk om de js bestanden ook van de knockout site af te halen.

Knockout.js: https://github.com/downloads/SteveSanderson/knockout/knockout-1.3.obeta.js
Knockout.mapping.js: https://github.com/SteveSanderson/knockout.mapping/blob/master/build/output/knockout.mapping-latest.debug.js

Pas vervolgens de _layout.cshtml aan zodat de packages toegepast worden:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.6.4.min.js")" type="text/javascript"></script>

    <script src="@Url.Content("~/Scripts/knockout-1.3.0beta.js")" type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/knockout.mapping-latest.js")" type="text/javascript"></script>
    @RenderSection("Scripts", false)
</head>

<body>
    @RenderBody()
</body>
</html>
```

Refereer hierin dus naar de twee Knockout bibliotheken.

Wat hier opvalt is dat we een extra Scripts sectie toegevoegd hebben. Dit is niet zo spannend maar wel een goede werkwijze. Deze sectie gaan we binnen de View gebruiken, dus zorg dat hij ook bij jou in de _layout.cshtml staat.

Tip: Controleer nog even of hier de juiste versies van de bestanden/packages genoemd zijn. Er komen tenslotte regelmatig nieuwe versies uit.

6.2.2 Simple bindings implementatie

We beginnen eerst met een verwijzing te maken naar ons myKnockout.js script. Dit doen we door de volgende regels bovenaan toe te voegen aan Index.cshtml

```
@section Scripts{
    <script src="@Url.Content("~/Scripts/myKnockout.js")" type="text/javascript"></script>
}
```

Je ziet meteen dat Razor de section herkent en hem geel maakt. Nu hebben we er voor gezorgd dat we script files kunnen toevoegen op onze views en dat ze tijdens het renderen toch in de Head terecht komen.

Voeg nu onderaan de Index.cshtml de volgende regels code toe:

```
<input type="text" id="myInput" data-bind="visible: textboxVisible "/>
<button id="switchVisibility">
    <span data-bind="text: switchVisibilityText"></span>
</button>
```

Zoals je ziet zijn het gewone html elementen zoals we die al heel lang kennen alleen staan er op twee elementen een speciaal attribuut genaamd: `data-bind`
In dit attribuut staat de binding (koppeling) tussen een eigenschap van het element en een property op het model. Voordat we dieper in dit attribuut gaan duiken, laten we eens eerst kijken naar het model.

Voeg aan myKnockout.js in de ExecuteOnStart function de volgende code toe.

```
var viewModel = {
    textboxVisible: ko.observable(false),
    switchVisibilityText: ko.observable('Show')
};

ko.applyBindings(viewModel);
```

Zoals we bij jQuery het **$** of **jQuery** object kennen, kennen we binnen KnockoutJS het **ko** object.

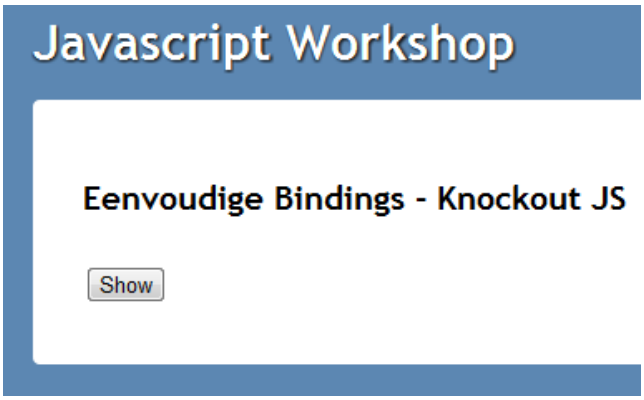
Als we dan de code gaan bekijken zien we eerst dat er een model object wordt aangemaakt met twee variabelen. Deze zijn beide van het type ko.observable, de een met als initiële waarde `false` en de ander met waarde `'Show'`

Omdat ze van het type ko.observable zijn kan KnockoutJS in de gaten houden of ze van waarde veranderen, zodat het framework erop kan anticiperen.

Het model is nu dus aangemaakt. Het enige wat nog gedaan moet worden is het model ook koppelen aan de UI, dit doen we door de functie ko.applyBindings(viewModel) aan te roepen waar we ons model dus als parameter mee geven.

Omdat we die `data-bind` attributen op onze html elementen hebben gezet doet KnockoutJS de rest voor ons. Bij het textbox element hebben we dus de boolean uit ons model gedatabind aan "`visible`" op de UI. En de "`text`" van het span element in de button is nu gedatabind aan de string uit ons model.

Start de applicatie nu maar eens en bekijk het resultaat:



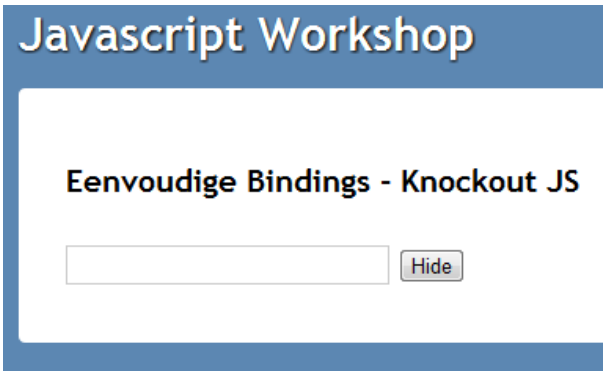
Het blijkt dus dat onze initiële bindings zijn gelukt. Nu moeten we alleen nog de button click afvangen en de property op het model wijzigen.

Voeg achteraan de ExecuteOnStart de volgende code toe:

```
$('#switchVisibility').click(function () {
        var isVisible = viewModel.textboxVisible();
        viewModel.textboxVisible(!isVisible);
        viewModel.switchVisibilityText(isVisible ? 'Show' : 'Hide');
});
```

Het enige wat hier opvalt, is dat de properties functies zijn geworden. We lezen waardes uit door ze als een functie aan te roepen. Zodra je een waarde mee geeft aan de functie zal deze intern geset worden, daarbij wordt direct de UI bijgewerkt.

Test nu nog eens de code en zie het resultaat:



6.3 KockoutJS en MVC

Knockout is een fantastische uitbreiding op Asp.Net MVC3. Hoewel er ook andere alternatieven zijn, is dit één van de bekendere MVVM uitbreidingen in JavaScript.

Maak het viewmodel aan voor de toekomstige controller action door de class KnockoutEditViewModel toe te voegen aan de /Models map en deze te vullen met deze code:

```
public class KnockoutEditViewModel
{
    public int Id { get; set; }
    public string FamilyName { get; set; }
    public bool IsScary { get; set; }
    public Address Address { get; set; }
}

public class Address
{
    public string Street { get; set; }
    public int Number { get; set; }
}
```

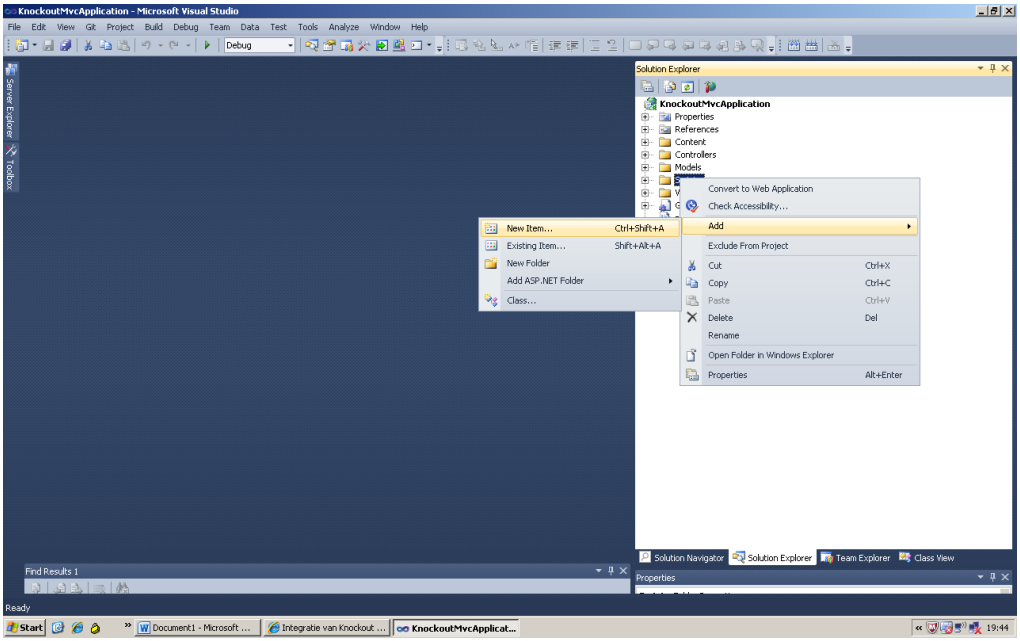
Dit is gewoon een viewmodel met daarin enkele simpele properties en ééntje van een samengesteld type. Voor de goede orde, dit viewmodel zal alleen tijdelijk op de server leven, tijdens de opbouw van de HTML.

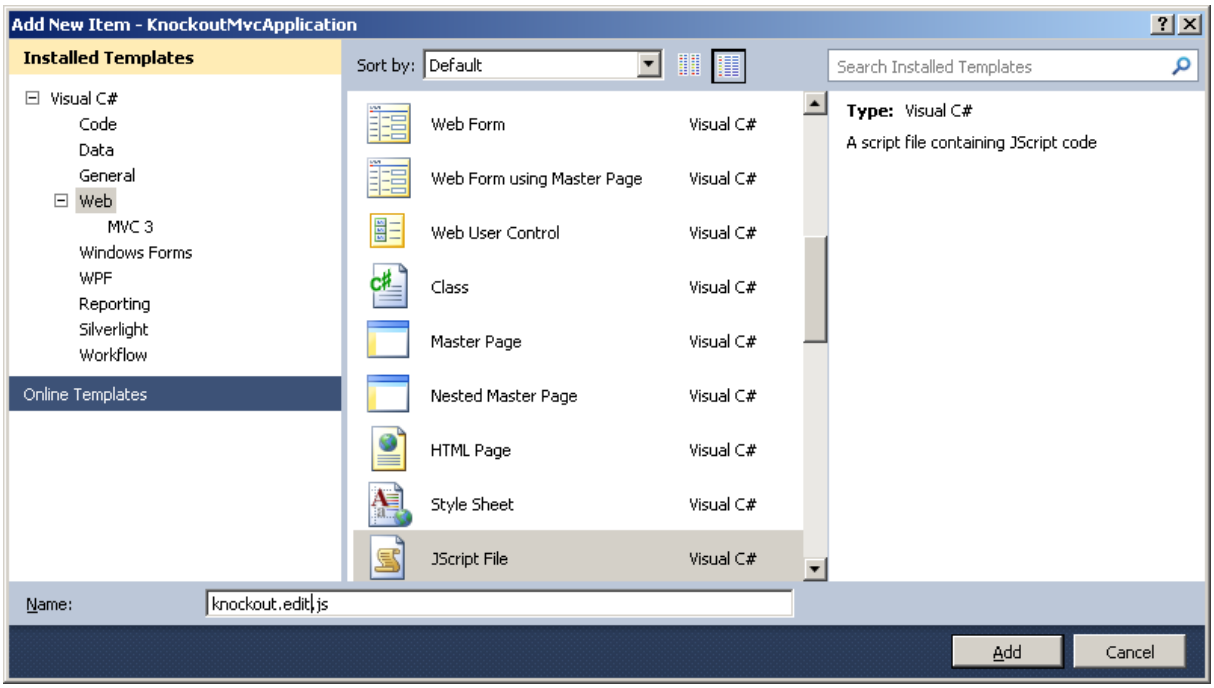
Maak een nieuwe controller aan genaamd KnockoutController. Deze zal standaard de action Index bevatten. Hernoem die naar Edit. Vul deze controller actions met de volgende code en pas het viewmodel toe:

```
public class KnockoutController : Controller
{
    public ActionResult Edit()
    {
        var model = new HomeEditViewModel();
        model.Id = 42;
        model.FamilyName = "The Adams family";
        model.IsScary = true;
        model.Address = new Address
        {
            Street = "Cemetery Lane",
            Number = 1313
        };
        return View(model);
    }

    [HttpPost]
    public ActionResult Edit(HomeEditViewModel model)
    {
        if (!ModelState.IsValid)
        {
            // do something with the data provided
        }
        return View(model);
    }
}
```

De essentie is hier dat er data verzameld wordt en dat de Edit View hiermee gevuld gaat worden door de ViewEngine. De PostBack methode is toegevoegd zodat te controleren is of de Submit van een gewijzigde View ook correct verwerkt wordt.

Voeg een extra JavaScript bestand genaamd knockout.edit.js toe aan de \scripts map.





Deze laten we nog even leeg... ☺

Maak de Edit view aan voor de action:

```
@model Models.KnockoutEditViewModel
@{
 ViewBag.Title = "Demo of MVC edit view combined with KnockoutJs (MVVM implemention)";
}
@section Scripts{
 <script src="@Url.Content("~/Scripts/knockout.edit.js")"
         type="text/javascript"></script>
 <script type="text/javascript">
     //todo
 </script>
}

<h2>MVC edit view combined with KnockoutJs (MVVM implemention)</h2>

@using (Html.BeginForm()) {
  @Html.ValidationSummary(true)
  <fieldset>
  <legend>KnockoutJS</legend>
  @Html.HiddenFor(model => model.Id)
  @Html.LabelFor(model => model.FamilyName)
  @Html.TextBoxFor(model => model.FamilyName)
  <label>@Html.CheckBoxFor(model => model.IsScary)Is scary</label>
  <hr/>
  @Html.LabelFor(model => model.Address.Number)
  @Html.TextBoxFor(model => model.Address.Number)
  <br/>
  @Html.LabelFor(model => model.Address.Street)
  @Html.TextBoxFor(model =>
    model.Address.Street)
  <p>
  <input type="submit" value="Save" />
  </p>
  </fieldset>
}
```

Hier zien we een aantal zaken:

- De knockout.edit.js wordt gerefereerd.
- Er kan JavaScript uitgevoerd worden in de HTML pagina zelf.
- Er wordt een HTML form getoond die alle elementen van het viewmodel wijzigbaar maakt.
- Er is een submit button om het scherm 'op te slaan'

Controleer nu even of dit allemaal werkt. Als het goed is dan wordt het edit scherm getoond en zullen wijzigingen aan de server doorgegeven worden op de POST action.

So far, so good.

Laten we nu Knockout.js introduceren. Het relevante aan Knockout is dat alle onder houdbare data in de browser gecentraliseerd wordt in een 'viewmodel'.

Hé, er was toch al een viewmodel? Ja, dat klopt, maar deze knockout ViewModel leeft in de browser en niet op de server. We moeten dus een copy van de server ViewModel in de browser stoppen. Dit doen we door de `//todo` in de edit.cshtml te vervangen met de volgende code:

```
var serverViewModel = @Html.Raw(Json.Encode(Model));
```

We vullen een JavaScript variabele met alle data van het viewmodel. Dit wordt gedaan door de Razor viewengine. Dus als de pagina eenmaal in de browser leeft, hebben we hier een JavaScript object met alleen maar properties.

Vervolgens passen we de knockout.edit.js aan:

```
/// <reference path="jquery-1.6.4-vsdoc.js" />
/// <reference path="knockout-1.3.0beta.debug.js" />
/// <reference path="knockout.mapping-latest.debug.js" />

$(function () {
    var clientViewModel = ko.mapping.fromJS(serverViewModel);

    clientViewModel.FullAddress = ko.dependentObservable(function () {
        return this.Address.Number() + ' ' + this.Address.Street();
    }, clientViewModel);

    clientViewModel.FamilyInfo = ko.dependentObservable(function () {
        return this.Id() + ' ' + this.FamilyName();
    }, clientViewModel);

    clientViewModel.doSomethingNifty = function () {
        if (this.IsScary()) {
            alert("Booh!");
        }
        else {
            alert("I could do some Ajax stuff here with Id "
                  + this.Id() + ' and family ' + this.FamilyName());
        }
    };

    ko.applyBindings(clientViewModel);
});
```

Wat zien we hier? De volgende zaken zijn hierin vermeld:

- Bovenin staan enkele referenties in commentaar om VS2010 de 'code insight' voor JavaScript te laten aanbieden
- Er staan een war code die uitgevoerd wordt zodra de pagina geladen is en aan de gebruiker getoond wordt (de .ready())
- De door de ViewEngine gegenereerde variabele in de browser wordt gemapped naar een knockout viewmodel. Dit wordt automatische gedaan door de ko.mapping.fromJS().
- Het Knockout ViewModel wordt uitgebreid met nog een paar properties (zoals Fulladdress) en een functie (.doSomethingNifty())
- Het Knockout ViewModel wordt via de applyBindings 'observarable' gemaakt.
- En die laatste stap is HEEL belangrijk. Alle properties van het originele viewmodel EN de nieuwe properties zijn hier nu geïnstrumenteerd met "ko.dependentObservable()". Dit betekent dat attributen van willekeurige DOM elementen afhankelijk worden van waarden in die properties.

Laten we dit eens toepassen. Verander de onderste code van de Edit.cshtml:

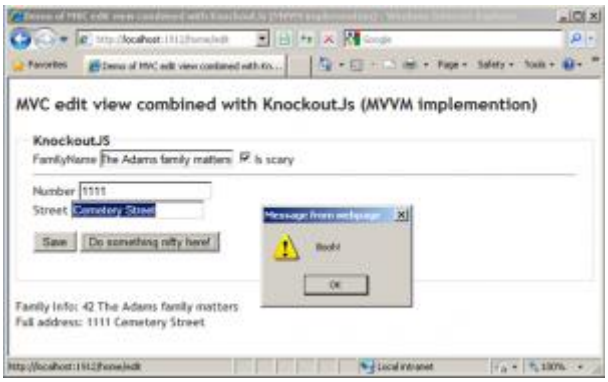
```
@using (Html.BeginForm()) {
  @Html.ValidationSummary(true)
  <fieldset>
        <legend>KnockoutJS</legend>
        @Html.HiddenFor(model => model.Id)

        @Html.LabelFor(model => model.FamilyName)
        @Html.TextBoxFor(model => model.FamilyName, new { data_bind = "value: FamilyName" })

        <label>
             @Html.CheckBoxFor(model => model.IsScary, new { data_bind = "checked: IsScary" })
             Is scary
          </label>

        <hr/>

        @Html.LabelFor(model => model.Address.Number)
        @Html.TextBoxFor(model => model.Address.Number, new { data_bind = "value: Address.Number" })

        <br/>

        @Html.LabelFor(model => model.Address.Street)
        @Html.TextBoxFor(model => model.Address.Street, new { data_bind = "value: Address.Street" })

        <p>
           <input type="submit" value="Save" />
           <button data-bind="click : doSomethingNifty">Do something nifty here!</button>
        </p>
  </fieldset>
}

Family info: <span data-bind="text: FamilyInfo"></span>
<br/>
Full address: <span data-bind="text: FullAddress"></span>
<br/>
```

Wat we hier zien is dat we van een aantal schermelementen bepaalde properties gebonden hebben aan het Knockout viewmodel. Dit is gedaan via de data-bind, een nieuw HTML5 attribuut. Dit attribuut wordt door Knockout angstvallig in de gaten gehouden om te controleren of elementen onderling gesynchroniseerd moeten worden. Als namelijk meerdere elementen dezelfde Knockout viewmodel properties 'data-binden' dan moeten ze allemaal bijgewerkt worden als de waarde veranderd.

In dit geval data-binden we het value attribuut van de input elementen, het checked attribuut van de checkbox, de text attributen van de twee spans onderaan en de klik op de button.

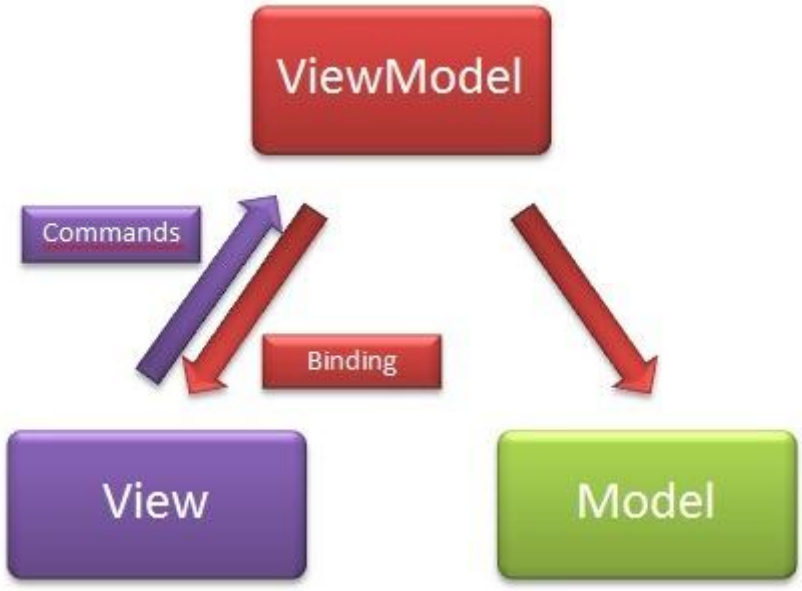
Tip: de data-bind van de input elementen is wat ongelukkig met die _ (underscore). De Razor viewengine kan namelijk niet omgaan met een – (minteken) in property namen. Dit wordt bij conventie opgelost als de ViewEngine zijn werk doet. De _ wordt een -.

In wezen hebben we nu de werking van Knockout te pakken. Voer de code uit en zie dat family info en Full Address netjes bijgewerkt wordt als de gerelateerde invoervelden gewijzigd worden.



Zie ook dat de klik op de tweede button code uitvoert binnen het Knockout viewmodel waarbij data uit het viewmodel wordt uitgelezen.

Dus dit is nu MVVM in de browser? Ja, dit is MVVM, in de browser. Kijk maar eens naar dit plaatje:



De View is hierbij de input controls en andere DOM elementen op de HTML pagina. Het ViewModel is het Knockout ViewModel die de bindings ondersteunt. En die wrapt uiteindelijk het model wat ooit uit de controller doorgegeven is.

Scheiding tussen ViewModel en DOM elementen

Er is wel een valkuil waar je in kunt trappen bij Knockout, maar dat kan bij iedere MVVM implementatie. Kijk maar eens naar de volgende opzet.

Stel dat we een extra DIV element willen tonen afhankelijk van de checkbox. Voeg dus een extra DIV element toe op de Edit.cshtml:

```
<div data-bind="visible: IsScary">
  Did you know? Addams's original cartoons were one-panel gags.
  <br/>
  The characters were undeveloped and unnamed until later versions.
</div>
```

Zie dat deze direct op de checkbox gaat reageren.

Maar wat als we die DIV via code willen manipuleren? Stel dat we deze met een .show() of een .hide() willen verbergen of zichtbaar willen maken?

Dan moet de DIV losgekoppeld worden van het ViewModel. Haal de binding weg en voeg een 'Id' toe:

```
<div id="didYouKnow">
  Did you know? Addams's original cartoons were one-panel gags.
  <br/>
  The characters were undeveloped and unnamed until later versions.
</div>
```

Dan zouden we de DIV kunnen manipuleren via een extra observable property op het ViewModel:

```
// slecht voorbeeld
$(function () {
    var clientViewModel = ko.mapping.fromJS(serverViewModel);

    //de overige ViewModel code staat hier
     clientViewModel.DoSomethingWhenTheScarybooleanChanges
                      = ko.dependentObservable(function () {
                          if (this.IsScary()) {
                              $('#didYouKnow').show(500);
                          }
                          else {
                              $('#didYouKnow').hide(250);
                          }
                          return 'dummy';
                      }, clientViewModel);

    ko.applyBindings(clientViewModel);
});
```

Dit werkt prima! Maar dit is een slecht voorbeeld! Waarom? Er is nu binnen het viewmodel een directe relatie gelegd DOM elementen. En dit is niet wenselijk, het vervuilt het ViewModel met extra logica. Het ViewModel heeft geen weet van DOM elementen. Punt.

Hoe moet dit dan opgelost worden? Dit kan veel eleganter…

Kijk, de code moet uiteindelijk ergens staan. Maar waar? Als het niet binnen het ViewModel mag staan, dan zetten we het er buiten. En die code mag alleen uitgevoerd worden als er een (bepaalde) wijziging in het ViewModel optreedt.

Wellicht herken je dit als een Event en dat is juist wat we gaan toepassen. Dit heet bij Knockout een .substribe(). Vervang de code uit het vorige voorbeeld door deze code:

```
$(function () {
    var clientViewModel = ko.mapping.fromJS(serverViewModel);

    //de overige ViewModel code staat hier
    var isScarySubscription =
        clientViewModel.IsScary.subscribe(OnIsScareChanged);

    ko.applyBindings(clientViewModel);
});

function OnIsScareChanged(isScary) {
    if (isScary) {
        $('#didYouKnow').show(500);
    }
    else {
        $('#didYouKnow').hide(250);
    }
}
```

Voer dit ook eens uit en zie dat we exact dezelfde werking hebben.

7. References

http://marcofranssen.nl

http://sandervandevelde.wordpress.com/

http://addyosmani.com

http://jquery.com

http://jqueryui.com

http://knockoutjs.com

http://github.com/atosorigin/javascriptWorkshop