|  |
| --- |
|  |
| Workshop JavaScript |
| jQuery, jQueryUI, KnockoutJS |



|  |
| --- |
| Marco Franssen, Jan Saris, Sander van de Velde  10-5-2012  Translated by Aage ten Hengel |



Inhoudsopgave

[1. Basic jQuery 3](#_Toc328231406)

[1.1 Creating a MVC3 project 3](#_Toc328231407)

[2. jQuery UI 21](#_Toc328231408)

[2.1 Scripts 21](#_Toc328231409)

[2.2 Dialog 22](#_Toc328231410)

[2.3 Datepicker 24](#_Toc328231411)

[2.4 Buttons 26](#_Toc328231412)

[2.4.1 Standard button with icon 26](#_Toc328231413)

[2.4.2 Radio buttons 27](#_Toc328231414)

[2.5 Animations / effects 28](#_Toc328231415)

[2.1 Extra (optional) assignments 30](#_Toc328231416)

[3. Ajax within hands’ reach 31](#_Toc328231417)

[3.1 Terminology 31](#_Toc328231418)

[3.2 Implementation 32](#_Toc328231419)

[3.3 Testing 37](#_Toc328231420)

[3.3.1 Request 38](#_Toc328231421)

[3.3.2 Response 39](#_Toc328231422)

[4. Advanced Javascript 41](#_Toc328231423)

[4.1 Make your own jQuery function 41](#_Toc328231424)

[4.2 Create a JavaScript object 44](#_Toc328231425)

[4.3 Communication between objects with use of events 48](#_Toc328231426)

[5. MVVM with Knockout JS 51](#_Toc328231427)

[5.1 Model View ViewModel 51](#_Toc328231428)

[5.2 Simple bindings 51](#_Toc328231429)

[5.2.1 Getting Nuget packages and altering \_layout.cshtml 52](#_Toc328231430)

[5.2.2 Simple bindings implementation 53](#_Toc328231431)

[5.3 KnockoutJS and MVC 55](#_Toc328231432)

[6. References 63](#_Toc328231433)

# Basic jQuery

Welcome to this introduction to *jQuery*. This chapter will mainly be about browser DOM manipulation. To those that haven’t heard of this concept before: DOM stands for Document Object Model, or the tree-structure of all HTML elements on the screen. These can be nested: e.g.: a DIV can contain another element.

Before we begin, we first need to setup a project structure in VS2010. We’re going to use a ASP.NET MVC 3 project.

## Creating a MVC3 project

Create an empty ASP.NET MVC3 project and name it **JqueryWorkShopMvcApplication** in Visual Studio 2010. Select Razor as the view engine and check ‘Use HTML5 semantic markup’.



We could download *jQuery* from the websit e(<http://docs.jquery.com/Downloading_jQuery>), be we also could use Nuget to add jQueryas package to our project. But out-of-the-box there’s an older version of jQuery Nuget package in a MVC3 project, let’s use that for now.

Check whether \_Layout.cshtml (equivalent of the masterpage in Razor) (is available in /Views/Shared) with the proper references.

|  |
| --- |
| <!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-1.7.2.min.js")" type="text/javascript"></script> </head>   <body>     @RenderBody() </body> </html> |

**Tip1**: The standardly available *Modernizr* JavaScript library is left out (out-of-scope for this workshop)

**Tip2**: You could also reference /Scripts/jquery-1.7.2.js. This has *exactly* the same logic, but isn’t minified. A minified version is optimalized for sending over the internet (comments are removed, functionnames are shortened, etc.). For JavaSript debugging purposes you should use the non-minified version.

Add a HomeController (right-click the controllers folder).



Name this HomeController:



After the controller is generated, you can see there’s an Index method available immediately. This is a MVC action; a method that generally represents a HTML page and can be called via *http*.

Generate a view for this action (right-click on the method > Add View)



Use the standard settings:



A View named ‘Index’ has been created:

|  |
| --- |
| @{     ViewBag.Title = "Index"; }   <h2>Index</h2> |

Now add a file named ‘home.index.js’ to the scripts folder via Add > New Item. Here we’ll be adding jQuery statements, this code will be debuggable via breakpoints in the *gutter*.



It’s a JavaScript file:



The JavaScipt file is now open in the Editor. Drag the jquery-1.7.2-vsdoc.js to the editor. A line with “comments” now becomes available:

|  |
| --- |
| /// <reference path="jquery-1.7.2-vsdoc.js" /> |

This enables VS2010 to offer *IntelliSense* for jQuery, which will come in handy.

Add a empty function:

|  |
| --- |
| /// <reference path="jquery-1.7.2-vsdoc.js" />   function ExecuteOnStartup() {   // Add your script here } |

Reference from index.cshtml to the *JavaScript* file:

|  |
| --- |
| @{     ViewBag.Title = "Index";             <script src="@Url.Content("~/Scripts/home.index.js")" type="text/javascript"></script> }   <h2>Index</h2> |

Now everything’s set to start the workshop!   
**Just compile to see whether everything works ;-)**

Which should give you the following result:



The generated HTML is simple and clear HTML5 markup:

|  |
| --- |
| <!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-1.7.2.js" type="text/javascript"></script>  </head>  <body>  <script src="/Scripts/home.index.js" type="text/javascript"></script>    <h2>Index</h2>  </body>  </html> |

**Tip:** This is generated by combining \_Layout.cshtml, the index.cshtml, data from the controller and the Razor ViewEngine .

Now add a couple HTML objects and a bit of JavaScript to the 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>  <a id="link2" href="">Link2</a>   <script type="text/javascript">     $(document).ready(function () {         ExecuteOnStartup();     }); </script> |

Now we have two hyperlinks without any logic (nothing happens when clicked).

The JavaScript in the script block will execute once this page is loaded in the browser. The only function of this bit of jQuery is the following:

“When this page is fully loaded by the browser and the user could almost start accessing controls on the page, the ExecuteOnStartup method needs to executed”.

This is essential! De .ready(); will only fire when the page is loaded, which can vary between browsers! JQuery solves a lot of browser dependancies.

We’re going to change the implementation of ExecuteOnStartup() so something will happen:

|  |
| --- |
| function ExecuteOnStartup() {       // add to all links (the a elements) a click function     $("a").click(function () {         alert("Hello world! (from " + this.id + ")");     }); } |

Here we have the first example of a jQuery selector. For all elements that are a link (an ‘a’) we’ve added a function to their click event (which’ll execute when the link is clicked). This will pop-up a dialog displaying the text and the value of the *id* attribute of the element clicked.

**Start the application an check the results:**

**Tip:** Use Ctrl + F5 to refresh the page, i.e. get the new html from the server. (You also don’t need to build and restart the debugger for changes in html, css en JavaScript).



Let’s click on both elements:

|  |  |
| --- | --- |
| The H2 (header with ‘Index’ as text) | Nothing happens |
| First link |  |
| Second link |  |

Both links execute the specified function. A selector can return 0,1 or multiple elements that need additional functionality or attributes

jQuery has many varieties to selectors. There’s an extensive list found on <http://api.jquery.com/category/selectors/>

We can also add functions to other element events on the DOM. See <http://api.jquery.com/category/events/>

Next we’ll further look into responding to events. Let’s adjust the ExecuteOnStartup method a little more:

|  |
| --- |
| function ExecuteOnStartup() {       // add to all links (the a elements) a click function     $("#link1").mouseenter(function () {         $("#link2").hide(500);     });       $("#link1").mouseleave(function () {         $("#link2").show(1000);     }); } |

We now see that a specific element, namely link1, is being attached to two functions. (jQuery only selects a UNIQUE id (‘link1’) because we tell it so with the pound sign (‘#link1”)).

If the cursor moves over link1, then link2 is going to be made invisible. This will happen in 500 milliseconds (see the parameter passed to the *hide* function) and will be animated. If the cursor now leaves link1, jQuery will show link2 in 1000 milliseconds, also by animation.

**Try it:**



You’ll see the second link disappear faster than reappear.

Here we see two important features of jQuery in play. The *selectors* simplify finding a specific element, even if it’s nested in a table, for instance. jQuery provides simple animations which enhances the user experience.

There are a lot effects available, which we’ll delve into in chapter 3.5.

Furthermore, when you set a breakpoint in the [home.index.js](http://home.index.js), you’ll see VS2010 will stop here (this only works when running in Internet Explorer).



With the VS2010 *Immediate* screen or a mouse-over you can see the contents of the JavaScript elements.

Let’s try out some other things with the selectors. First add some paragraph elements after the *<a>* elements.

|  |
| --- |
| ...  <p>Hello</p>  <p>and</p>  <p>Goodbye</p>  ... |



And we’ll also add a new stylesheet file to the Content folder.



Name this: home.index.css:



Also add this to the index.cshtml, as a reference:

|  |
| --- |
| @{     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 the *CSS* the following *styles* need to be added:

|  |
| --- |
| p { margin: 8px; font-size:16px; }       .selected { color:blue; } |



Let’s also add the following line to the ExecuteOnStartup method:

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



Try to run this. You should be getting this alert:



Followed by the following view in the browser:



What happened here? Our changes created the following situation:

* All three paragraphs now have a different margin and font-height (standard CSS behavior).
* All last paragraphs (just one in this case) are represented by a variable (the selector only needs to traverse the DOM once, looking for elements).
* We fire an alert with the contents of the final paragraph.
* We’ve coded that the LAST paragraph gets the *selected* style.
* And we’ve replaced the text in the last paragraph.

**Tip**: Did you see the last line of the same paragraph was being manipulated twice in one statement? All functions (like the .addClass(“…”) and the .text(“…”)) manipulate the available list of objects and subsequently return them. These elements that have already been changed by the .addClass() method are manipulated by the .text() method immediately afterwards. This is what we call a *Fluent* notation, which you might know from *LinQ* and *Entity Frameworks 4.1 Code First*.

The .ready() can be seen as a kind of selector as well.

With jQuery we can read from and change attributes of (multiple or specific) elements. And we can read the CSS styles and change these as well.

It’s even possible to select elements which have a specific style, this makes jQuery very powerful. This way we can access single elements (with the same CSS style) and give them a *new* CSS look.

Let’s do some more stuff with selectors.

Add a table to the index.cshtml with with a columnname and four tablerows.

|  |
| --- |
| <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> |

Once again, add some JavaScript to the ExecuteOnStartup:

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



And we need some additional styling for the table in the home.index.css:

|  |
| --- |
| …  table {    background:#eeeeee;  }  … |

This gives us the following result:



What can we see here?

* The entire table has received a different background color (standard CSS behavior).
* The *even* rows in the table head get a different color
* The *even* rows in the table body get a different color

We can also use the selector to search specifically nested. This is preferred for documents with a lot of DOM elements, where only a few objects need to be altered. Searching for elements is basically similar to traversing all elements and this takes time and energy (you wouldn’t want someone’s Smartphone being depleted after visiting your website).

The header row WON’t be manipulated via selection of *uneven* rows in the header:

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



This doesn’t give us a difference in color on the header as regards to the *total* background color.



The line with ‘header’ is the zeroth line, so an even line, not uneven…The selector returns ZERO (=none) in this case.

When we’re talking about the DOM , i.e. the tree-structure of elements as seen by the browser, we can also see this in modern browsers or plug-ins like *Firebug*.



As a final step we’ll manipulate some items in an ordered list.

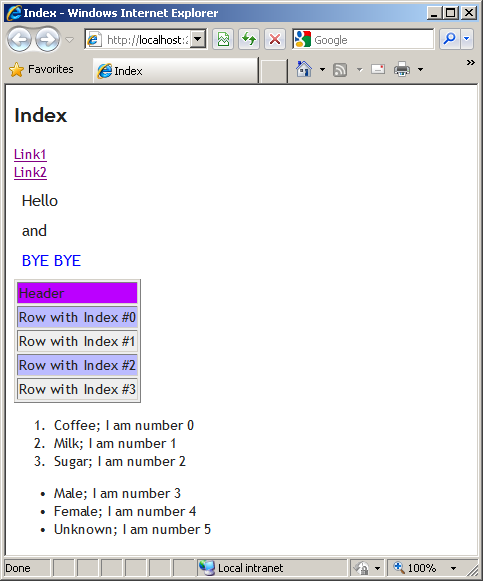
First add two ordered lists to the bottom of the index.cshtml. In HTML the we’ll see numbers or bullets in front of our list items, depending on the type.

|  |
| --- |
| <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> |

Then we’ll add the following lines of JavaScript to the ExecuteOnStartup:

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

When we run this we see that after every list item a text has been added:



The ordered lists are accessed via their *class* attribute. It’s interesting to see that the numbers count on instead of starting over (0,1,2,3,4,5 vs. 0,1,2,0,1,2). Why would that be?

Because the $(".orderedlist") represents a part of the DOM, two lists in this case. The .find("li") will return all items, six in total, not two times three. Keep this in mind.

**Conclusion**

It’s very simple to create attractive pages with only a few lines of jQuery code. Would you be able to do this using standard JavaScript? That would’ve worked on all browsers?

***Bonus assignment 1:***

|  |
| --- |
| How would you have the counting start over in the last assignment (0,1,2,0,1,2)? |

***Bonus assignment 2:***

|  |
| --- |
| Add a checkbox to the page which will show or hide the table when checked (hints: input, type , checkbox, .change() .toggle()) |

# jQuery UI

We now know how jQuery works in principal. We can adjust DOM elements and attributes and asynchronously get data from the server. Now it’s time to get down and dirty.

We’ll be using extensive UI controls that run in the browser. We want to get the feeling of working with a real Windows application.

Again we’ll start with the basis. And what’s a dedicated Windows application without an Error box??? ☺

## Scripts

jQuery has already done a lot of work for you en offers different UI controls. For this, we first need a jQuery JavaScript file and a jQuery CSS file.

Just like in chapter 1 we need the following:

1. Make sure jQuery UI is added to your project. This is standard when creating MVC3 project, because it has the designated Nuget package installed.
2. Add a controller and name it UIController. This has a standard Index Action.
3. Make a view named Index.cshtml in the /Views/UI folder
4. Add a jQueryUI.js file to the /scripts folder
5. Reference the jQueryUI.js from the index.cshtm

To view this newly created page we’ll navigate to the following URL (after compiling of course): <http://localhost:53890/ui/>

***Important:*** Change the port in the URL to the port your local website uses.

***Tip***: You can also compose a lightweight jQuery UI on their homepage. This could be necessary if you only use a specific set of functions. While increasing performance, this could add an extra dependency in maintenance.



Reference the JavaScript and CSS in the top of the \_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-1.8.19.min.js")" type="text/javascript"></script>  … |

We now have the *jQuery UI JavaScript Library* and accompanying CSS properly referenced. Here’s where things start to get really interesting!

Add the following to the index.cshtml:

|  |
| --- |
| <h2>JQuery UI</h2>    <button id="openDialogButton" type="button">Open dialog</button>    <div id="myDialog">      Error <br /> My first error dialog  </div>  <div id="dialogState">  </div>  <script src="@Url.Content("~/Scripts/jQueryUI.js")" type="text/javascript"></script> |

Notice that the reference to jQueryUI.js is included here.

Add the following code to the 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 } |

We’ll work in this file next. First have a look at the first few lines, where we’ll shorten the start of the script a little.

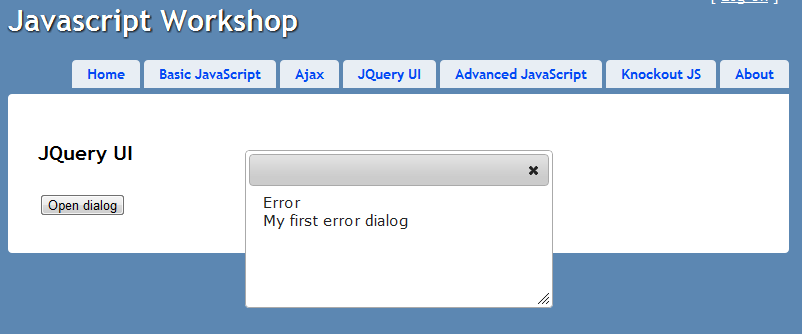
## Dialog

Okay, let’s make this page a little fresher and more interactive!  
We’ll transform a <div> element to a *dialog-box*.

We already had prepared the HTML as you saw. Now go to the JavaScript file (jQueryUI.js) and add the following line to the ExecuteOnStartup function:

|  |
| --- |
| $('#myDialog').dialog(); |

  
**Run the application and see the results:**



Thanks to the UI library of jQuery we now have a standard *dialog* plugin to work with, that turns a given DOM element into a dialog. All because we called the .dialog() function.

But, as you saw, this dialog is pretty barren. It didn’t yet have a title and we missed the ‘Ok’ button to close our little error dialogbox. *And* the dialog automatically opened, we would want to handle this ourselves.

In other words, we want to configure our dialog with preferences.

Right at the start of the logic of the ExecuteOnStartup method, create a config object and pass this to the initialization of the dialog. You can achieve this by adding the following code *before* the initialization of the dialog:

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

  
We now have a config object with a few properties which’ll are pretty much self-explanatory. The property “buttons” is another object (with properties, ‘Ok’ in this case). To the ‘Ok’ property we didn’t add a “primitive value” but a proper function, which will execute when the ‘Ok’ button is clicked.

We’ll now pass the config object to the initialization of the dialog. And we’ll connect the opening of the dialog to the button-click of the button visible on the page.

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



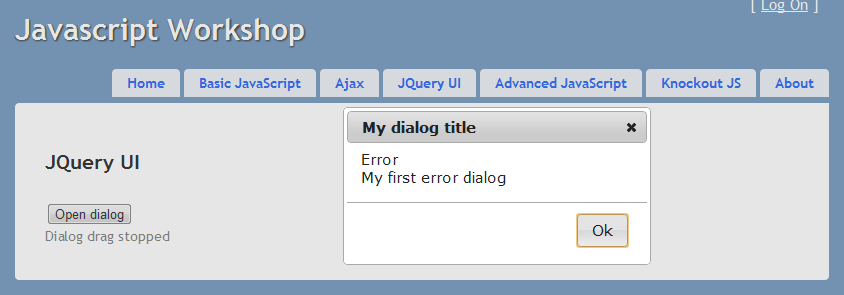
Let’s try and run this. Notice the dialog opening *on* the button-click.

A dialog will also generate different *events*. You can take advantage by subscribing to these. In this case we’ll show the state of the dialog right underneath the button.

Now add the following code to signal these events:

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

**Run the application and see the results:**



***Something’s not working, but what? Tip:*** *look at the setting of the config object!*

***Tip* 1:**Now we subscribe to the events *after* initialization, but – like you could see with the ‘Ok’ button – you can also include this in the config object. Your configuration would look something like this:

|  |
| --- |
| 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:** For an overview of all dialog functionality options, check out:

<http://jqueryui.com/demos/dialog/>

## Datepicker

You can tell that it’s remarkably easy to show a dialog! Let’s find out what else jQuery UI can do.

We will now add a Datepicker to our dialog. For this to work we need to add some HTML and an additional line of JavaScript.

Empty the <div> with the ‘myDialog’ id in the Index.cshtml. There you’ll add an input element of type ‘text’ with ‘myDate’ as the id.

|  |
| --- |
| <input type="text" id="myDate" /> |

Underneath in the ExecuteOnStartup method, you’ll add the initialization of the Datepicker in the Index.cshtml:

|  |
| --- |
| …  $('#myDate').datepicker();  … |

**Start de application, open the dialog and see the result:**



So, by literally adding *one* line of code, we can change a textbox into a Datepicker. The same applies here as for the dialog: we can configure the Datepicker by passing options to its initialization and/or subscribing to an event.

When you select a date, it’ll use a specific notation, which you might want to adjust to your own country’s standard. Furthermore you’ll notice the calendar starting on a Sunday. We’ll reconfigure these here.

Let’s change the firstDay and the date-notation (this example uses Dutch formatting). To avoid unnecessary lines of code, we’ll make the config object inline. We can then replace the initialization of the just now added JavaScript file with the following code:

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

  
**Start the application and see the results:**



The Datepicker also has a lot more possibilities. Check out demos and technical documentation on: <http://jqueryui.com/demos/datepicker/>

## Buttons

Nowadays we use MVC3 which has a modern appearance, to which we’ve added nice dialogs and date pickers. But now our buttons are starting to look a little simple by comparison. Another example of jQuery’s simple solutions.

### Standard button with icon

Let’s style the button we already have on the page. Let’s make it nicer with an icon.

Let’s go back to the jQueryUI.js file and add the following code:

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

  
**Run the application and see the results**:



See that one button is styled really nicely with two different icons. You can add an icon before the text *and* after as well. Suit yourself. We’ve restyled it by only calling the .button() function.

Because we passed a configuration object to this function with *one* property (‘icons’), jQuery automatically shows *both* icons.

The icons are already present in our project by adding jQueryUI. This is extended by default with several CSS styles and images, which are configurable.

Underneath the *themeroller* site you can see all available icons: <http://jqueryui.com/themeroller/#themeGallery> . When you hover over an item, a name appears which you can include in the config object.

***Tip***: The styling of a button works on both the <input type=”button”> HTML element as on the<button> HTML5 element. **But the adding of icons only Works on <button> elements! The following code styles all button elements in your entire page:**

|  |
| --- |
| $(“button”).button(); |

This states that for all (on that time present) buttons within the entire DOM, the .button() function will be called. This ensures a neat styling from the jQueryUI CSS for all those buttons. The reference to the icons earlier, was an added bonus.

This will create one uniform look for all your pages.

### Radio buttons

Another type of button that’s used quite often is the *radio button*. We also want to finetune these. Let’s start by adding several radio buttons to the page. Add the following lines to the Index.cshtml between the <h2> and <button> elements:

|  |
| --- |
| …  <div id="radio" style="float: left">  <input type="radio" id="radio1" name="radio" value="fade" checked="checked" /><label for="radio1">Fade out</label>  <input type="radio" id="radio2" name="radio" value="bounce" /><label for="radio2">Bounce</label>  <input type="radio" id="radio3" name="radio" value="explode" /><label for="radio3">Explode</label>  </div>  … |

**Run the application and see the results**:



This isn’t bad at all, but it doesn’t fit the current design. We want to have three buttons that are mutually exclusive (the selection of one deselects the previous one chosen). To make this work we’ll apply the styling to this as well.

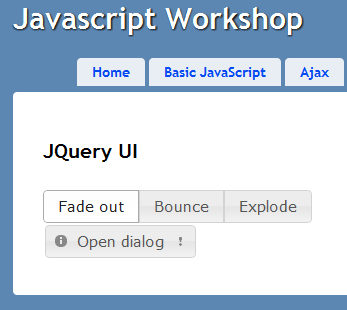
Like you can see in the newly added HTML we’ve grouped the radiobuttons in one <div> with ‘radio’ set as its id.

This wasn’t accidental; jQuery needs this.

Add this line of code to the ExecuteOnStart function:

|  |
| --- |
| $( "#radio" ).buttonset(); |

**Rerun the page and see its beautiful new look!**



## Animations / effects

Yeah, let’s get this party started!

We’ll make our application even nicer by adding some charming animations. But never make anything too tacky, okay? But for this demo it can’t do any harm, so go right ahead. On your local machine dress codes don’t apply! ;-)

First we’ll add an extra button and another HTML element which’ll bee subject to our animations.

Add a animation button, RIGHT AFTER the radiobutton:

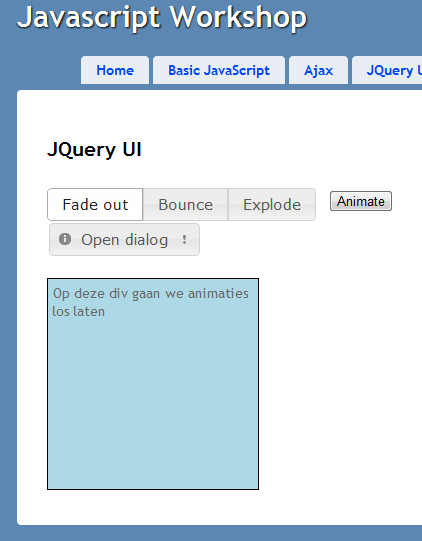
|  |
| --- |
| …  <button id="animationButton" type="button" style="float:left; margin-left: 10px;">Animate</button>  … |

On the bottom of the page we’ll add another <div> which we’ll use for animations.

|  |
| --- |
| … <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:Styling is done inline in this example, but to keep things neat you would want to do this in a CSS file!**

**See the results:**



**Hey! The ‘Animate’ button doesn’t have any styling! Use the tip from paragraph from 2.4.1 here.**

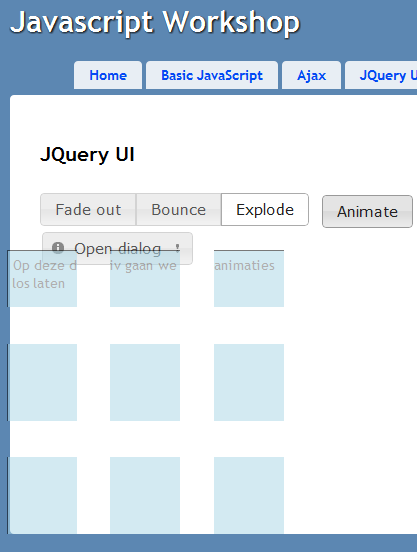
Then, add the following code to the end of the ExecuteOnStart function:

|  |
| --- |
| …  $('#animationButton').click(animateDiv);  … |

And underneath in the JavaScript file (outsite of the ExecuteOnStart) add the following function:

|  |
| --- |
| function animateDiv() {      var checkedButton = $(':checked', '#radio');      var selectedEffect = checkedButton.val();        $("#animationDiv").effect(selectedEffect, {}, 500 );  } |

Play with the different buttons and start the different animations:



This code is pretty self-explanatory. By usage of the .checked selector jQuery finds the element with the *checked value*.It’ll only look within child-elements of the #radio (because I’m certain there will always be *one* that’s checked). Now we can call the first object in the switch statement and ask its id.

Finally, the effect will run with an empty config and the number of milliseconds. I’ve conscientiously chosen effects that don’t need configuration. For a list of all effects and configs check out: <http://jqueryui.com/demos/effect/#option-effect>

## Extra (optional) assignments

 ***Assignment 1:***

|  |
| --- |
| Display other icons on the ‘#openDialogButton’ when selecting *one* of the radiobuttons. |

 ***Assignment 2:***

|  |
| --- |
| Open the dialog with an effect. Tip: use configuration for this. |

 ***Assignment 3:***

|  |
| --- |
| Try to use the ‘size’ effect. Mind you: you need a config for this! |

# Ajax within hands’ reach

In the last chapter we’ve seen how we can traverse the DOM and manipulate elements with jQuery. In this chapter, we want to show you exactly how easy it is to make AJAX calls from within a website.[[1]](#footnote-1)

## Terminology

We’ve all seen the possibility to do a POST in the browser. A lot of screens with input options, even log-in screens, usually have several textfields and a submit button. In general all input is sent to the server, which will be processed by logic on the server and a new screen will be generated. This mechanism is what we call a **post-back.**

In this chapter we’ll execute a **call-back**. The user (or a timer; in modern browsers this could be an event on the Host OS, like a change in location) will initiate the execution of some JavaScript. For instance, some data has been entered in several input fields. This data will then be passed to the server, but only this data. Which will in return process this data and the call-back will return information. Which will, in general, be some data and no markup.

That callback could fail or not respond right away. That’s why we’d take of this **asynchronously.** The browser will fire a call and *won’t* wait for an answer. No, it’s assumed that an answer *will be there* at some point, which would need to execute some JavaScript code when it actually arrives. These are callback functions, and the attributes of these functions are the answer the server provides.

Ok, not too difficult. But what does this communication between browser and server look like? In general this is taken care of with a *querystring* (or a *fat URL*) for the call and the response is in **JSON** format. JSON is nothing but an alternative notation, like XML. But XML is less applicable, since this can cause the messages to be bigger (XML trees are bigger due to every element having a closing tag as well).JSON is lightweight and has another big advantage; JSON can easily be converted to JavaScript variables, without any real impact on the CPU.

## Implementation

Let’s start with offering data for AJAX calls.

We add an extra action (method) to the HomeController. Notice MyJsonResult the class? We’ll make this in the next step.

|  |
| --- |
| using JQueryWorkshopMvcApplication.Models;  public class HomeController  {  //Index action etc……  [HttpPost] public JsonResult DoSomething(string name, string phone, int? zip, string email) {             var myJsonResult = new MyJsonResult();             try             {                 // do some filtering                 Thread.Sleep(2000);                   var 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 Json(myJsonResult);  }  } |

A method is provided here to pass name, phone, zip and e-mail. Because this method has the Post attribute, this could mean that we’ll change data on the server. This *wouldn’t* be possible with a spider.

***Tip***: The *nullable int* is made nullable to prevent a HTTP Error 500 (internal server error) from occurring when data is presented that’s *undefined* in JavaScript. The textfield *could* be left empty, for all we know. With a nullable int the input *can* be mapped to the parameters.

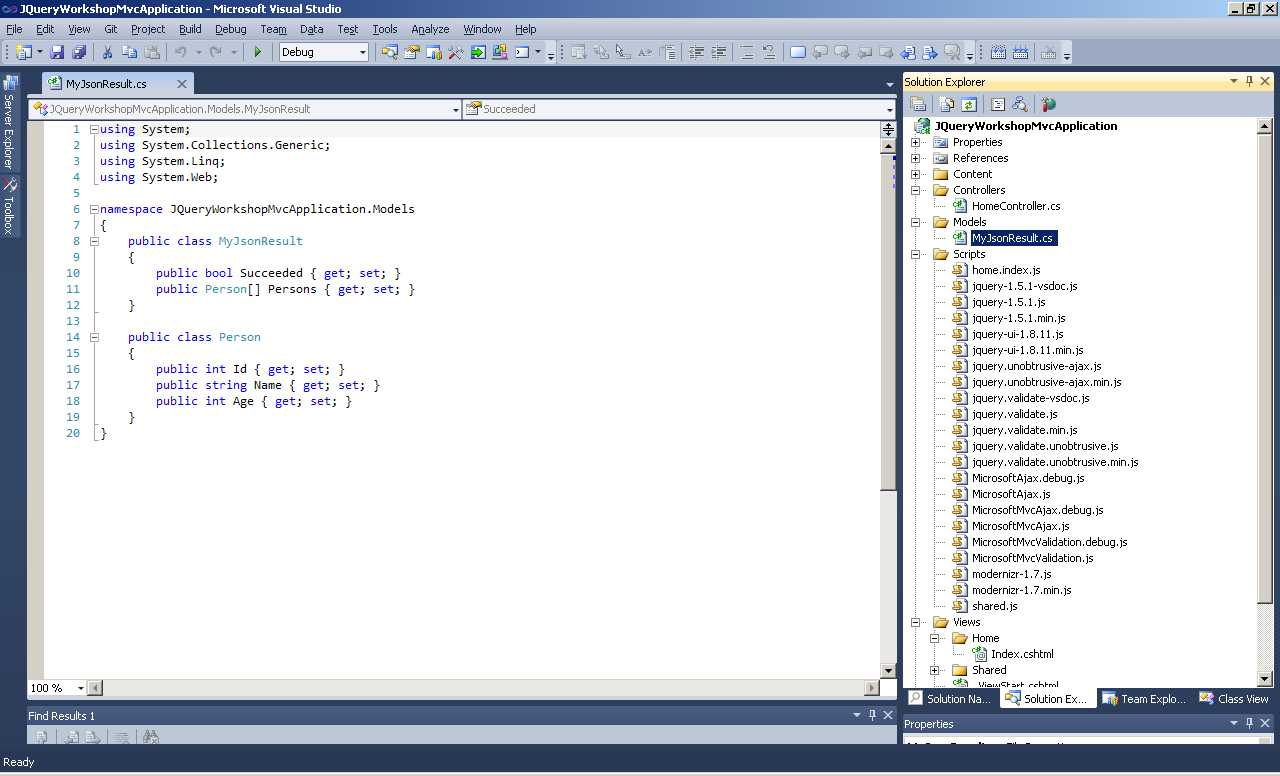
***Tip***: We should still test whether everything is being passed correctly, but that’s always going to be the case with AJAX: never trust data from outside!

For our workshop the functionality will be a little simpler: we’re going to wait two seconds en return a list of not so youngies.

The most important of this action logic running on the server is written on the last line of the action. Here we return a special inheritance of ActionResult, namely our objects wrapped as JsonResult. The C# objects we return are defined like this on the 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; }     } |

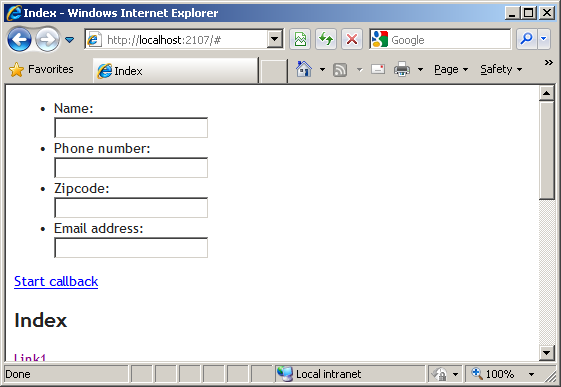
Make these classes in the Models folder of our MVC application:



In the index.cshtml we need to prepare the call. First make sure some fields are available for the user to fill in stuff. By preference do this AT THE TOP of the indx.cshtml. This will result in a better effect later on…

|  |
| --- |
| …  <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>  … |

 This should generate the following screen.



Here the user can submit some data.

***Tip***: Here we use a link to start the call. However, we should really use a submit button. This prevents spiders from accidentally firing the AJAX call and polluting our database.

Now the submitted information needs to be sent to the **DoSomething** action in the HomeController (the zipcode is an integer, by the way). We will construct the address by the *viewengine* on the server, before showing the page to the user. In order to do that, we first compose the URL of the AJAX call on the index.cshtml and this address will be made available as JavaScript parameter of the ExecuteOnStartup. So adjust the parameter accordingly:

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

***Tip***:By constructing the URL this way, we make sure the AJAX call keepts working if this code is deployed on the server (read: on another machine).

The function needs to receive this parameter:

|  |
| --- |
| function ExecuteOnStartup(callbackUrl) {  … |

The only thing to do now, is make the call itself. Let’s use the .ajax() for this.

This call is quite extensive, but gives a good impression of would be happening on a server. 

Add the following code to the ExecuteOnStartup method:

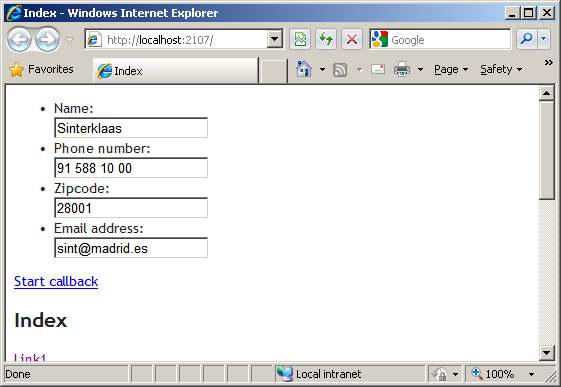
|  |
| --- |
| $('#errorInfo').hide(); $('#name').val("Sinterklaas"); $('#phone').val("91 588 10 00"); $('#zip').val("28001"); $('#email').val("sint@madrid.es");  $('#btnStartCallback').on('click', function (e) {     var name = $('#name').val();     var phone = $('#phone').val();     var zip = $('#zip').val();     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);         }     }); }); |

What do see here? Let’s first have a look at ‘errorInfo’. This is a <div> where you can display errors and other messages. We hide this during the .ready.

Next up, we want populate the ‘Value’ attribute input controls with different values (this is done with the .val(‘some value) method).

Then, the click event is bound to a function within the start URL. This is an alternative notation of the .click() (this could also be a *tap* on the iPhone ☺). De .on() is a more generic way of *binding* functions to events.

Eventually, the screen is displayed *before* the callback is being executed:

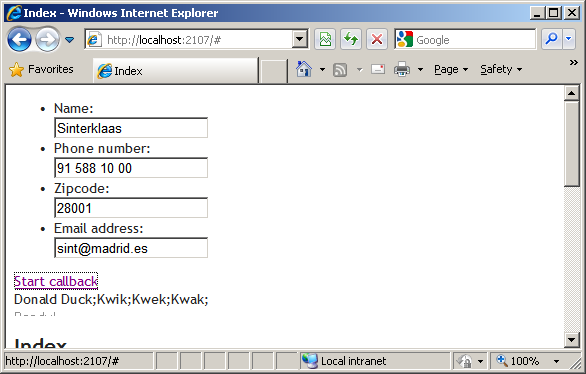


If the click event fires, the following code is being executed:

1. We read the different values from the input controls via the .val() method. All of these values are combined in *one* JavaScript variable.
2. We make the AJAX call, where we set the following:

|  |  |
| --- | --- |
| url | The to be called URL for the callback. In this case /Home/DoSomething. |
| data | The to be passed data |
| dataType | The format in which the communication will happen, in this case Json. |
| type | The type of HTTP call, in this case a POST (the same as the attribute on the action in the controller). |
| beforeSend | The function that needs to be executed before the call. Here we use it to show the text “Busy…”. This is done in Fluent notation. We also empty the to be filled results <div> with the .empty(). This removes all text and HTML elements within the <div>. |
| Success | The function that needs to be executed when the AJAX call is successful.  The provided result (myJsonResult) is passed here as ‘result’. We can immediately read the result.Succeeded en then iterate through the result.Persons with the $.each().This will have every item execute a function where ‘this’ represents that specific ‘person’. |
| error | This function is executed when something does go wrong during communication. This could happen when the wrong action is being called or if incorrect parameters are passed. |

Now execute this code. See that the fields are populated. You could alter the input, but then you need to click ‘Start callback’. We now get to the server where the *sleep* makes us wait a bit. After returning, names are shown and the text “Ready” will appear, which will then slowly disappear.

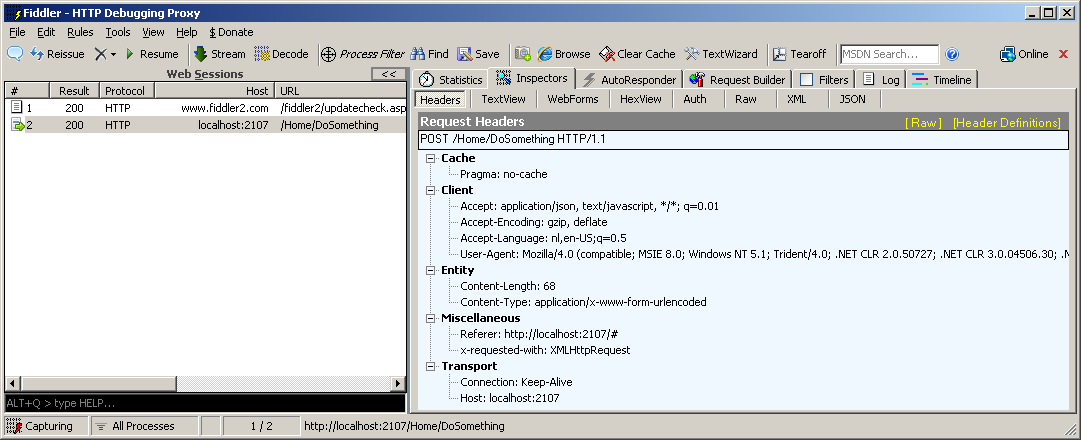


## Testing

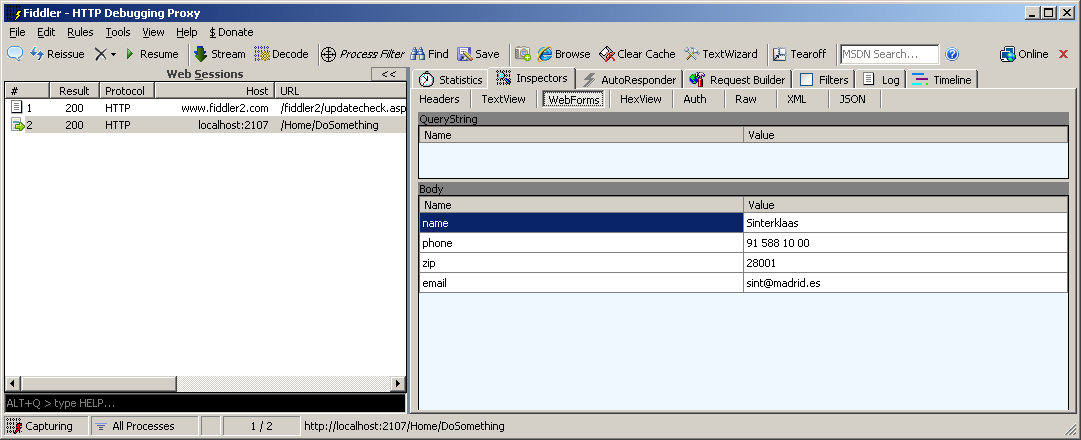
It’s a best practice to build AJAX logic *and* testing it with a tool like *Fiddler* (modern browser have this function built-in; try an press ‘F12’ in your browser ☺). Fiddler Works like an extra local proxy where all traffic of the browser is being eavesdropped. This is perfect for “listening” to the questions of the browser and the answers of the server. Find out what Fiddler has to say about this callback.

### Request

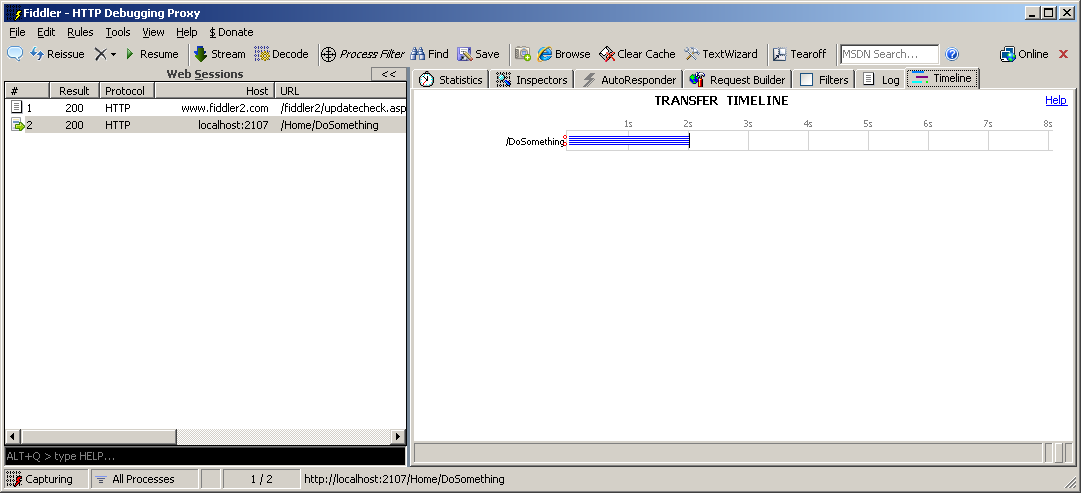
The call was a POST to /Home/DoSomething:



The values that were being passed were:

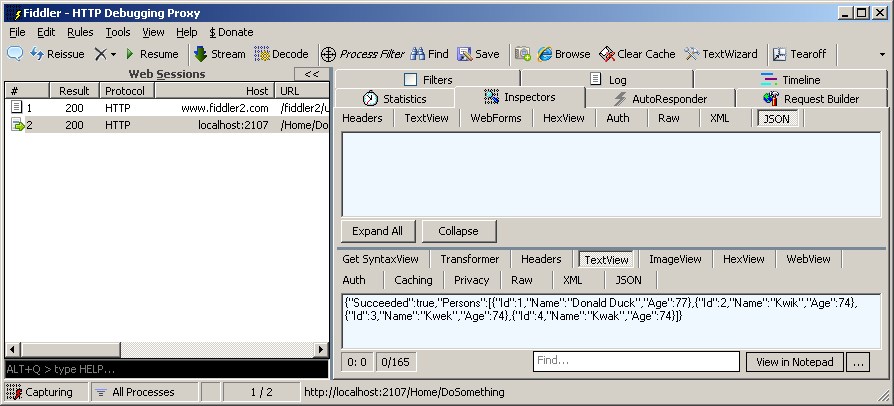


The call took about 2 seconds (because of the ‘*sleep*’ on the server):

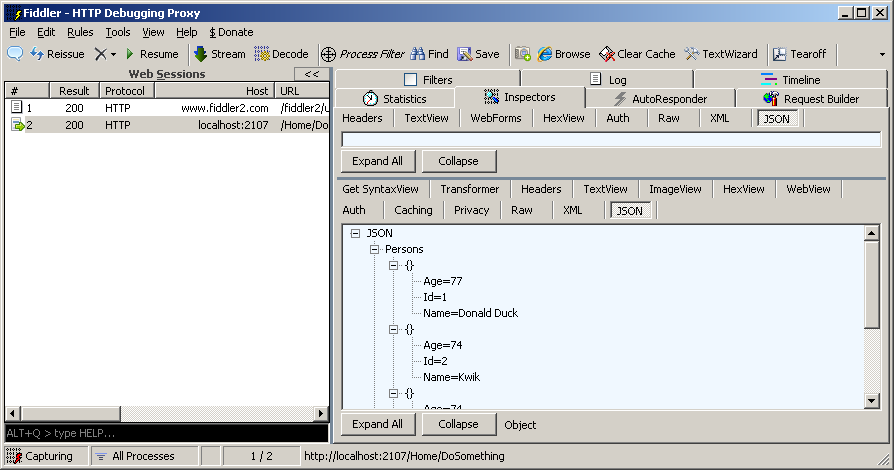


### Response

That the call returned with a 200 was already visible. Below we can see raw Json return:



For more complex constructions another notation is also possible:



**Conclusion**

Execution of an asynchronous callback is simplified by combining ASP.NET MVC3 and jQuery. Be sure to test what happens on the line and for all circumstances.

For more info on the .ajax():

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

***Tip***: For the more advanced developer; there also are the $.Get and the $.Post(). These are simplified notations for the same call, but are beneficial in that you’re less likely to make a typing mistake.

***Extra assignment 1:***

|  |
| --- |
| Try to change the code, in a way that you get to see the error function as well as different scenarios in the Success function |

***Extra assignment 2:***

|  |
| --- |
| Try to do a GET |

***Extra assignment 3:***

|  |
| --- |
| Make a button out of the link. |

***Extra assignment 4:***

|  |
| --- |
| In this chapter our actions requested 4 parameters. Change this code so that it expects 1 parameter that is a class you create. The 4 parameters need to be properties of this class. Notice the functionality remains the same. |

# Advanced Javascript

## Make your own jQuery function

We’ve now used multiple functions of jQuery, for instance, the $.ajax() and the $(‘’).button().

But how to write our own function which we can reuse? That’s what this paragraph is about.

First do the following:

1. Make a controller and name it AdvancedController. This has an Index Action by default.
2. Make a view and name it Index.cshtml and place it in the /Views/Advanced folder
3. Make an advanced.js in the /scripts folder
4. Fill advanced.js with the same JavaScript as chapter 2.1
5. Reference advance.js from Index.cshtml
6. Correct the standard route in the Global.asax.cs by replacing:

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

For:

|  |
| --- |
| new { controller = "Advanced", action = "Index", id = UrlParameter.Optional } |

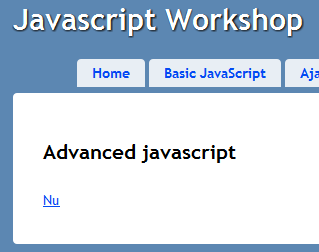
******  
Add the following line to the Index.cshtml:

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

And to the ExecuteOnStart function in advanced.js the following code (spelling error included!!!):

|  |
| --- |
| $('#nuputnl').text('Public link'); |

See the results:



We expected “Public link” to appear and not “Nu”.

**Can you tell what the problem is?**

The reference above to the *id* of the link isn’t copied correctly.

In this case we knew while writing that we wanted to manipulate one *and only one* control. With a little more effort we could’ve made this happen because now we have an incorrect situation. We now manipulate ZERO elements instead of one.

**What’s the solution?**

It’s very simple to write jQuery functions yourself that .NET developers would recognize as ExtensionsMethods. These can be called in fluent notation on the $(…).

We’ll now write a function that alert us if zero or multiple objects are found.

Add the following code in the bottom of advanced.js ( **outside** of the ExecuteOnStart function).

|  |
| --- |
| (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); |

This function is named .onlyOne() and accepts a string as a parameter to manipulate.

***Tip***: Normally you’d write a function in a separate JavaScript file that would be referenced, but for now this is sufficient.

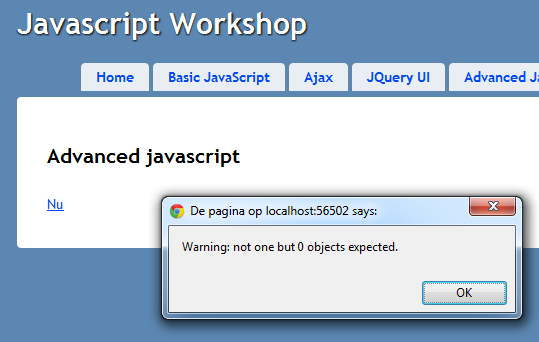
The (function( $ ){ … })( jQuery); is meant to prevent ambiguity with other (jQuery) libraries.

We’ll now apply this function directly and expand our code with the following:

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

Within the onlyOne function there’s a lot of use of the this keyword. Here this is the result of the selector$(‘#nuputnl’). To maintain the fluent notation you always return $(this)(or an edited version of those elements if you make a filter).

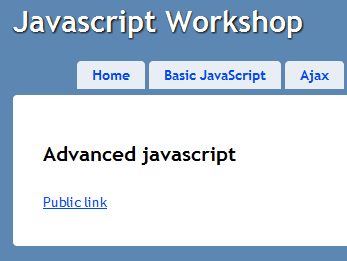
Run the application and see the results:





Correct your test code.

You should see the following:



***Extra assignment:***

|  |
| --- |
| Rewrite the function so that it only throws an error when NO elements are found. With multiple elements it should work correctly. |

## Create a JavaScript object

In previous chapters we’ve written our first JavaScript functions. But if anybody starts writing functions that end up in the *global* you’ll quickly find out that it can be quite messy if you copy functions, just like that. Not very handy!

We want to introduce structure in our code, but how? JavaScript isn’t an *Object Oriented* language. Instead of **Class** based programming it’s called *constant* ***Prototype*** *based* programming, but what on Earth is that?

This is what Wikipedia mentions on the subject:

Wat zegt Wikipedia over Prototype-gebaseerd programmeren:

|  |
| --- |
| **Prototype-based programming** is a style of **object-oriented programming** in which **classes** are not present, and **behavior reuse** (known as inheritance in class-based languages) is performed via a process of **cloning** existing objects that serve as **prototypes**. This model can also be known as **classless**, **prototype-oriented** or **instance-based** programming. **Delegation** is the language feature that supports prototype-based programming |

A form of object oriented? No classes? Object, copy?

We’re going to make a function where we’ll use functions and variables.Of this function we can then make an object, like we’re used to in C#. We’ll get the feeling of working with neat classes after all.

Let’s start and find out how this works.

We’ll begin by defining a function with the name “Person” outside the scope of the ExecuteOnStart() method. We will be using this function as our class of which we’ll make objects.

|  |
| --- |
| function Person() {  }; |

Now we first add two priavet variables to this Person function, namely “firstName” and “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.

But how do we make a public variable? That you do by using this. We now add a public property “MyName”.

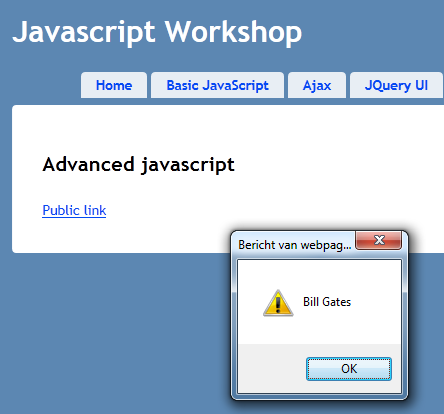
|  |
| --- |
| function Person() {      var firstName = 'Bill';      var lastName = 'Gates';      this.MyName = firstName + ' ' + lastName;  }; |

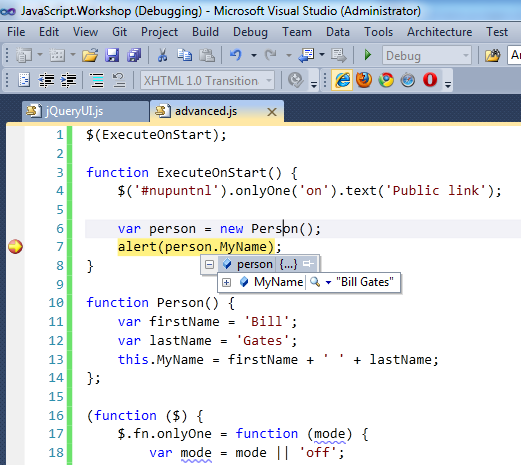
Sof ar, the making of a class…

Now we try to make an instance of this and use it. Instanciate an object at the end of ExecuteOnStart() and use an alert to display its name.

|  |
| --- |
| var person = new Person();  alert(person.MyName); |

Run and see the results:



Set a breakpoint on the line with the alert and look at the object in the debugger

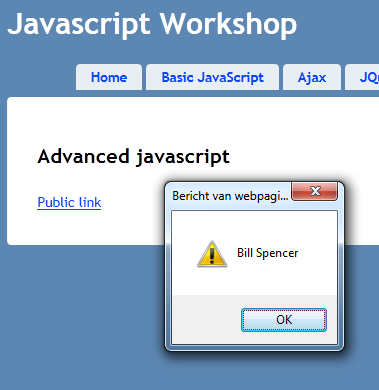
You can really only see the public variable we made!

That’s how it works with (almost) all internal functions.  
We now add several functions within the Person function:

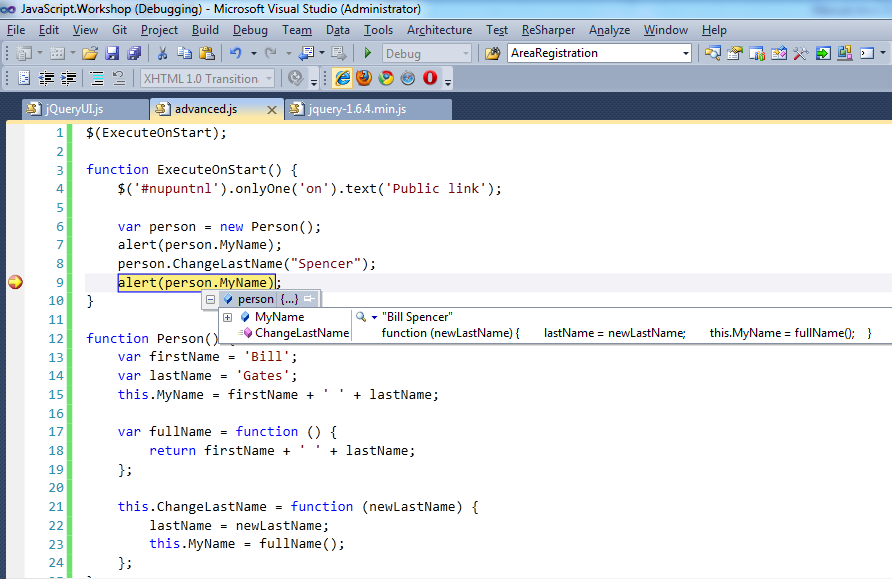
|  |
| --- |
| var fullName = function () {      return firstName + ' ' + lastName;  };    this.ChangeLastName = function (newLastName) {      lastName = newLastName;      this.MyName = fullName();  }; |

We’ll obviously call the public function ChangeLastName() as well.  
So add this to the  ExecuteOnStart method after our last alert().

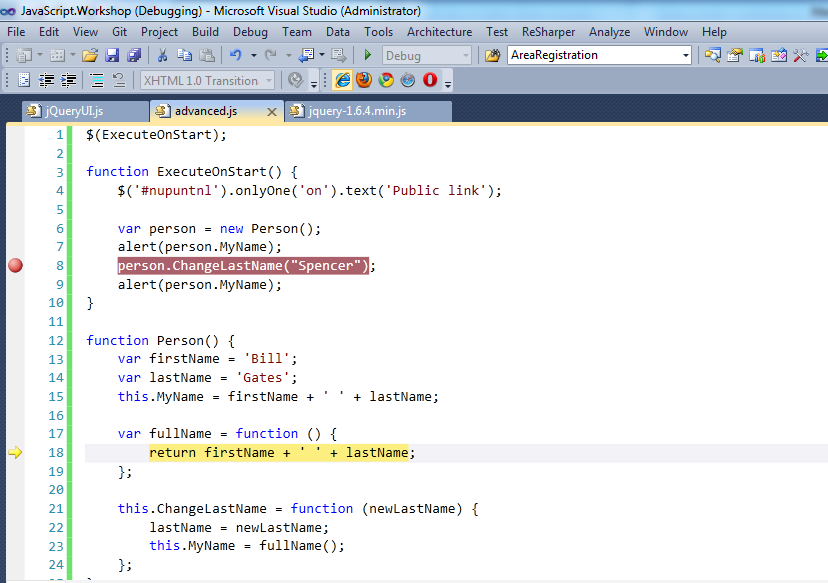
|  |
| --- |
| person.ChangeLastName("Spencer");  alert(person.MyName); |

When you run this code the second alert will display the following message:

When we start debugging again and have a look at the object, we’ll see that *one* function has been added to be viewed publicly.



The private functions aren’t mentioned in the debugger as well, but if you step through (F11) you do step through these.



Okay, we’ve come a long way already. We now have public and private variables and functions. The only obstacle now is that the private functions *can’t* make use of public functions and variables. Try and access the MyName from the fullName(). This doesn’t work like you’d want it to.

How do we solve this? We can *set* public variables with public functions, like we did with setting this.MyName.

Imagine that the setting of a public variable from a private function is necessary, then there will be a solution for that, but we need a reference to the object itself for this. Then it can access its own public properties. In the next paragraph you can see how simple it is to reference itself in an object.

**Extra information:** There’re various ways to make a function public. In this example we’ve used this, to make it explicit. Another method is to declare them as private (with var) and then returning them in the end. In our example this would look like this:

|  |
| --- |
| 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      };  }; |

You define a public function name and connect this to an internal function (see the return). **Notice** that the setting of the this.MyName is no longer possible, since the function is made private internally. That’s why it’s in comments.

## Communication between objects with use of events

We’ve now written our own classes, but where does jQuery come in?

We’re now going to make use of jQuery Events to enable objects to communicate to each other.

To have an object producing events it needs a reference to itself. This is done simply. When the object initializes, we save this in a local variable.  
Add the following line in the Person function:

|  |
| --- |
| function Person() {      var self = this;  … |

Now we saved this in a variable named “self”, which we’ll be using later. In the construction of an new object, the ‘self’ will have the correct reference.

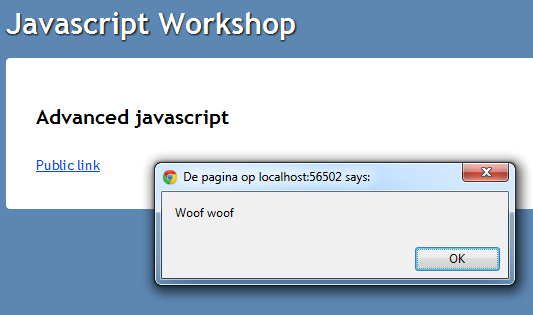
For communication we obviously need two different objects. We’re going to have an object of type Person communicate with an object of type Dog.  
Copy this code and place it after Person.

|  |
| --- |
| 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).on('shout', listenToWord);      };      //Start living      loop();  }; |

Add to the end of ExecuteOnStart () the initialization of a Dog object.

|  |
| --- |
| var dog = new Dog(); |

Run and see the results:



The Dog barks!

If you study this code, you can see that the dog has a reference to itself. Dog also has several functions. As long as the state of Dog is ‘Barking’ he will bark once every 2,5 seconds.

In the listenToWord() method the Dog listens to two words: “Bark” and “Quiet”, which can change its active state.

But the most important bit is in the public function Listen(). Here you pass a Person object named “boss” to which the Dog will listen.

With help of the following line: **$(boss).on('shout', listenToWord);** you subscribe to the ‘shout’ event of the boss. When on the boss the ‘shout’ event fires, the listenToWord function will be called.

We now have a Dog that can listen to a boss, but we would need to take care of this in code first.



Add the following public function to the Person class:

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

By using the .trigger() function you can fire an event. Here the person fires the ‘shout’ event on itself.

We now have a Dog that can listen and a Person that can shout, all we need to do now is connect these objects to make sure the person *will* shout.

Add the following Lines of code to the 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> |

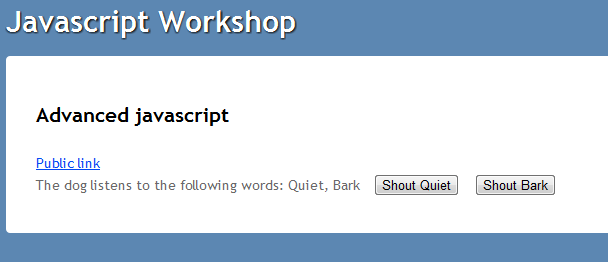


And in the JavaScript file the ExecuteOnStart() needs the following code:

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



Run the application and see if you silence the dog with the buttons



***Extra assignment:***

|  |
| --- |
| Have the Dog trigger an event to change his state. Place a <span> on the screen where you’ll show the state of the Dog. Use the .on() function to show the new state of the Dog. |

# MVVM with Knockout JS

## Model View ViewModel

Theoretic background is found here: <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>

Practical explanation:  
The idea behind MVVM is that a model (class object with properties) is connected to a View (pure UI). In the code we won’t change the UI in ways we’re used to in old techniques, but with properties of our model. The UI is only connected to these properties and the framework makes sure the UI is updated when one of the properties change.

**Example:   
Goal:** You have a textbox that you want to make visible if a button’s pressed.

**Oplossing:** On the model you make a ‘TextboxVisible’ property of type boolean. On UI level you connect the Visible property of the textbox to the TextboxVisible boolean of the model. You subscribe to the click event of that button. Within the click event you set the TextboxVisible property of the model to ‘true’. The framework will detect this, because it watches this property and will make the property visible or invisible accordingly.

It might look like magic, but let’s try and apply this, maybe it’ll make more sense then.

## Simple bindings

Let’s begin working out the scenario we’ve just decribed with *KnockOutJS*.

First do the following:

1. Make a controller named KnockoutController.   
   This has an Index Action by default
2. Make a view named Index.cshtml in the folder /Views/Knockout
3. In the /scripts folder make a myKnockout.js file
4. Fill the myKnockout.js with the same JavaScript code as in chapter 2.1.
5. We’ll make the reference to the file later!
6. Correct the code in the route in Global.asax.cs te from:

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

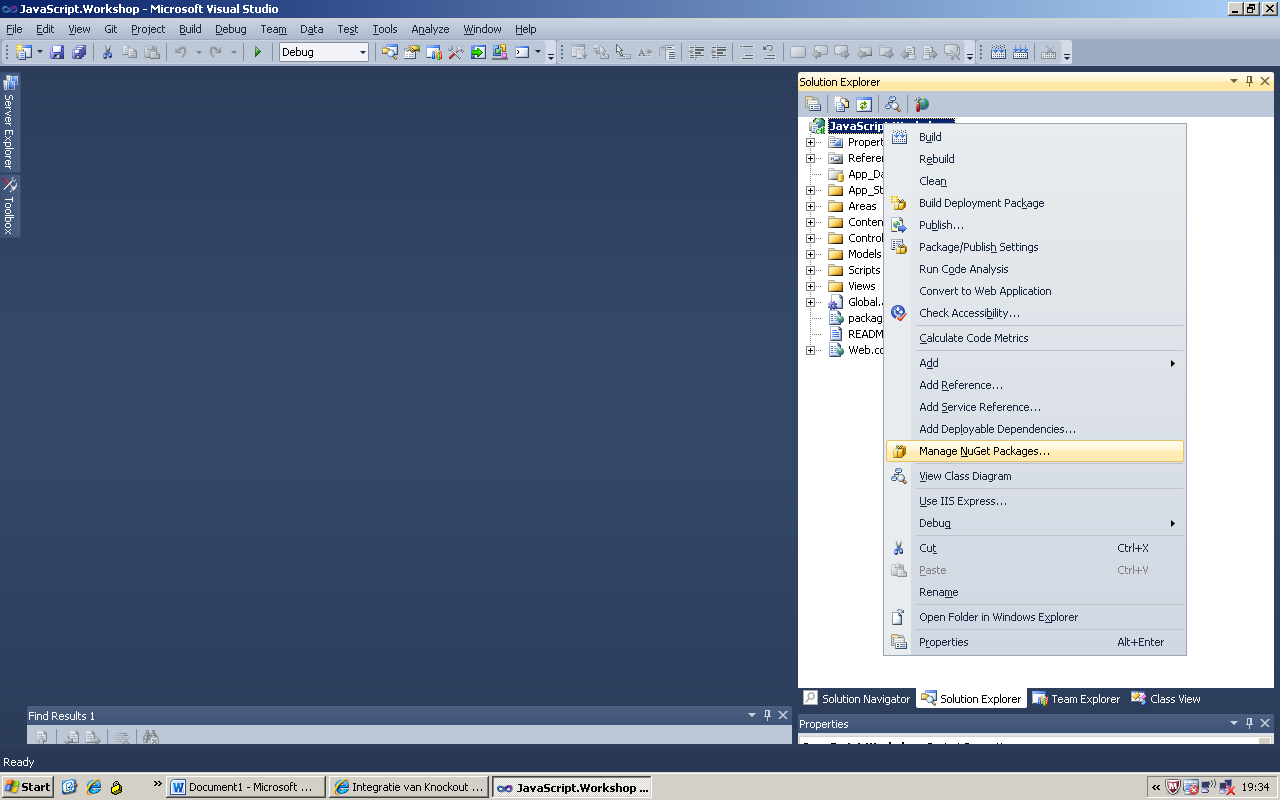
To:

|  |
| --- |
| new { controller = "Knockout", action = "Index", id = UrlParameter.Optional } |

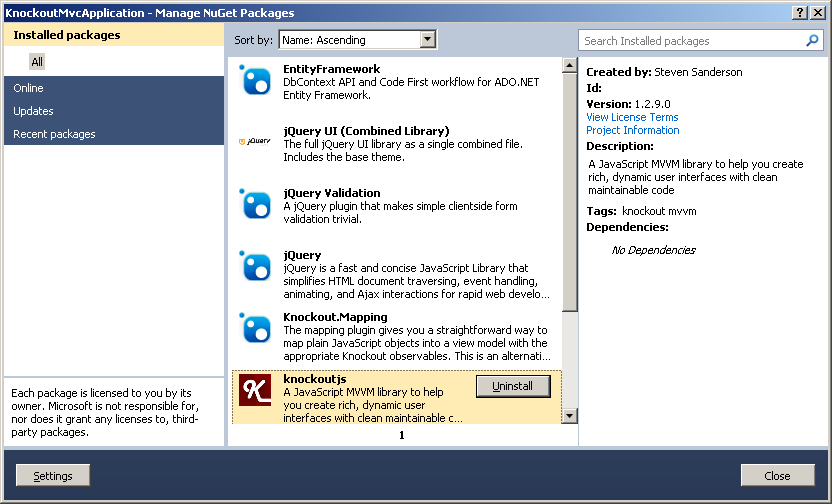
### Getting Nuget packages and altering \_layout.cshtml

The next ASP.NET MVC3 steps are preparation to this exercise. To demonstrate *KnockOut* correctly let’s setup the basis as it’s supposed to be. We assume that the previous chapters and/or experience with ASP.NET MVC3 are sufficient.

First add two Nuget packages to your ASP.NET MVC3 project. So right-click the project:



Select the **Knockout.js** package and the **Knockout.mapping** package:



***Alternative***: When you haven’t installed Nuget, it’s wise to this first! But it’s also possible to get the .js files from the Knockout site:

Knockout.js: <https://github.com/downloads/SteveSanderson/knockout/knockout-1.3.0beta.js>  
Knockout.mapping.js: <https://github.com/SteveSanderson/knockout.mapping/blob/master/build/output/knockout.mapping-latest.debug.js>

Alter the \_layout.cshtml so the packages are applied:

|  |
| --- |
| <!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-1.7.2.min.js")" type="text/javascript"></script>       <script src="@Url.Content("~/Scripts/knockout-2.0.0beta.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> |

So, reference to the two Knockout libraries.

Note that added an extra Scripts section. This isn’t that exciting but is a best practice. This section we will be using in the View, so make sure that it’s in the \_layout.cshtml.

***Tip***: Check if the correct versions of the packages/files are referenced. New versions are released regularly.

### Simple bindings implementation

We begin with a reference to our myKnockout.js script. We do this by adding the following lines at the top of Index.cshtml

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

You can see Razor recognizing the section and highlighting it. Now we made sure that we can add script files to our views and that they end up in our <head> section during rendering.

Now add the following lines at the bottom of the Index.cshtml:

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

As you can see they’re ordinary HTML elements but on two elements are special attributes named data-bind. This attribute enables the binding (linking) between two properties of the element and a property of the model. Before we delve deeper in this attribute, let’s first look at the model.

Add the following code to the ExecuteOnStart() in myKnockout.js:

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

Like with jQuery where we recognize the **$** or **jQuery** object, within KnocjoutJS we have the **ko** object.

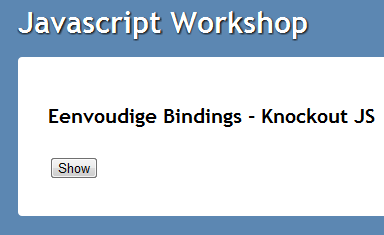
If we view this code we see that there’s a model object with two variables. These are both of type **ko. observable**, one with the initial valuefalse and one with the value 'Show'.

Because they’re of type ko.observable KnockoutJS can watch these for a change in value, so the framework can anticipate on this.

The model has been made. The only thing what needs to be done now is linking the model to the UI, which we’ll do by calling the ko.applyBindings(viewModel) function and passing our model as a parameter.

Because we’ve placed our data-bind attributes to our HTML elements, KnockoutJS will do the rest. In the textbox element the boolean of our model is now databound to the visible in the UI. The text of the <span> element in the button is now databound to the string of our model.

Run the application and see the results:



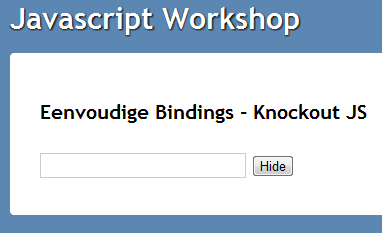
Our initial bindings have worked! Now we need to catch the button-click and change the property on the model.

At the end of the ExecuteOnStart() add the following code

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

The only thing that you should notice here, is that the properties have become functions. We read values by calling them as a function. When you pass a value to a function this will internally set the property, in doing so updating the UI.

Run this and see the results:



## KnockoutJS and MVC

*KnockoutJS* is a MVVM JavaScript framework.

**Make a ViewModel for a future controller action by adding class KnockoutEditViewModel to the /Models folder and adding this 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; }  } |

This is an ordinary viewModel with some simple properties and one of a composite type. To be clear, this ViewModel will only temporarily live on the server, while building the HTML.

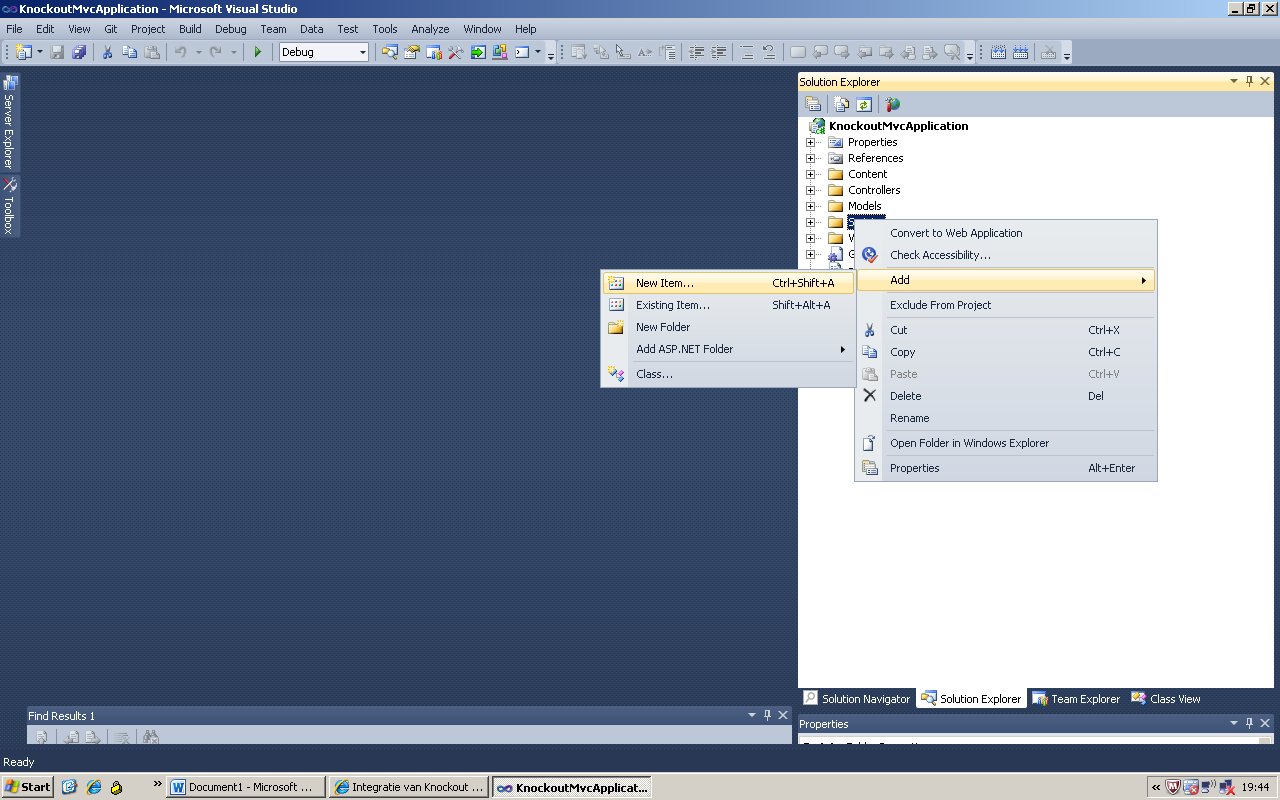
Make a new controller named KnockoutController. This will have a default Action Index. Rename this to “Edit”. Fill this controller actions with the following code and apply the ViewModel:

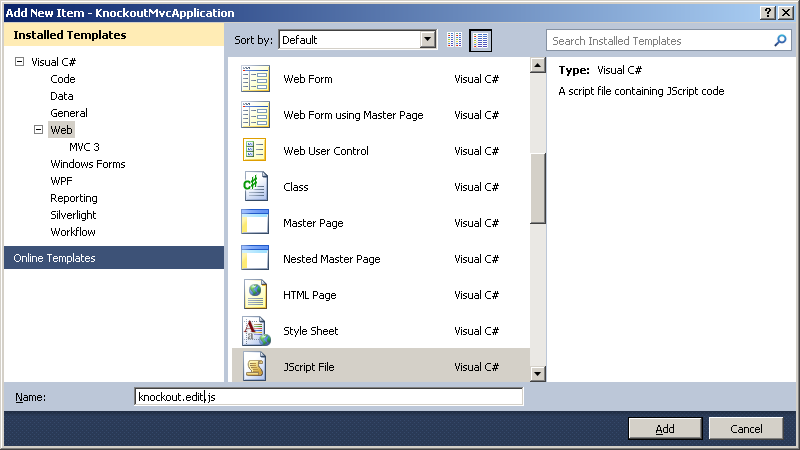
|  |
| --- |
| public class KnockoutController : Controller  {      public ActionResult Edit()      {          var model = new KnockoutEditViewModel();          model.Id = 42;          model.FamilyName = "The Adams family";          model.IsScary = true;          model.Address = new Address          {              Street = "Cemetery Lane",              Number = 1313          };          return View(model);      }        [HttpPost]      public ActionResult Edit(KnockoutEditViewModel model)      {          if (!ModelState.IsValid)          {              // do something with the data provided          }          return View(model);      }  } |

The essence is that data is collected here and that the Edit View will be populated with this by the ViewEngine. The PostBacl method is added to check whether the Submit of an altered View can also be processed.

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.

Ad dan extra JavaScript file named knockout.edit.js to the /scripts folder.





Which we will leave empty for now… ☺

Make an Edit view for the 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>  } |

A few things stand out:

* Knockout.edit.js is referenced.
* JacaScript can be run in the HTML itself.
* A HTML form is displayed that makes all elements of the ViewModel changeable
* You can “save” a submit button on the screen

Check if everything works. There should be an edit screen visible and changes should be sent to the server in the POST action

So far, so good.

Let’s now introduce Knockout.js. What makes Knockout relevant is that all maintainable data in the browser is centralized in a ViewModel.

Hey! Didn’t we already have a ViewModel? Correct, but this *knockout* ViewModel lives on the browser and not on the server. We need to make a copy of the *server* ViewModel and put it in the browser. This is done by replacing the //**todo** in the edit.cshtml by the following:

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

We’ll fill a JavaScript variable with all data from the ViewModel. This is done by the Razor viewengine. Once a page is living in the browser, we’ll have a JavaScript object with only properties.

Then we’ll change the knockout.edit.js.

|  |
| --- |
| /// <reference path="jquery-1.7.2-vsdoc.js" />  /// <reference path="knockout-2.0.0.debug.js" />  /// <reference path="knockout.mapping-latest.debug.js" />    $(function () {      var clientViewModel = ko.mapping.fromJS(serverViewModel);        clientViewModel.FullAddress = ko.computed(function () {          return this.Address.Number() + ' ' + this.Address.Street();      }, clientViewModel);        clientViewModel.FamilyInfo = ko.computed(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);  }); |

What can we see here? Take note of the following:

* At the top some references are commented out to have VS2010 offer IntelliSense for JavaScript
* There’s code being executed when the page is loaded (the .ready()).
* The by the ViewEngine generated variable in the browser is mapped to a Knockout ViewModel. This is done automatically by a ko.mapping.fromJS().
* The Knockout ViewModel is expanded with some additional properties (like ‘Fulladdress’) and a function named .doSomethingNifty().
* The Knockout ViewModel is made *observable* via the applyBindings().
* And the final step is VERY important. The *computed* properties make sure that, if a change occurs in one of the dependant properties, the computed property will be updated as well.

Let’s apply this. Change the bottom line of the Edit.cshtml

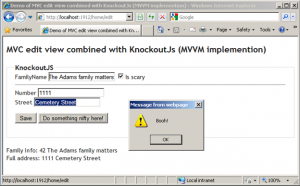
|  |
| --- |
| @using (Html.BeginForm()) {   @Html.ValidationSummary(true)   <fieldset>  <legend>KnockoutJS</legend>  @Html.HiddenFor(model => model.Id)    @Html.LabelFor(model => model.FamilyName)  @Html.TextBoxFor(model => model.FamilyName, new { data\_bind = "value: FamilyName" })    <label>  @Html.CheckBoxFor(model => 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/> |

What we see here is that we’ve bound several elements on the screen to specific properties in the Knockout ViewModel. This is done via the data-bind, a new HTML5 attribute. This attribute is being carefully watched by Knockout to check if elements need to be synchronized. Because, if multiple elements databind the same Knockout ViewModel properties, they will all be updated when the value changes.

In this case we databind the ‘value’ attribute of the input elements, the ‘checked’ attribute of the checkbox, the ‘ text’ attributes of the two <span> elements and the button-click.

***Tip***: The data-bind of the input elements has an underscore. De Razor ViewEngine can’t deal with a hyphen in property names. This is done by convention when the ViewEngine does its work; the ‘\_’ becomes a ‘-‘.

This is how Knockout works in essence. Run this code and see that family info and Full Address are being updated neatly when the related input fields are being changed.

[](http://sandervandevelde.files.wordpress.com/2011/09/knockout08.png)

See that the click on the second button also runs code within the Knockout ViewModel where data is being read from the ViewModel.

So this is MVVM in the browser? Yup, this is MVVM, in the browser. Look at this image:



The View here are the input controls and other DOM elements on the HTML page. The ViewModel is the Knockout ViewModel that supports bindings and wraps the model that was passed from the controller.

**Separation between ViewModel and DOM elements.**

When using Knockout there’s a pitfall to whatch out for, but that applies to all MVVM implementation. Look at the following setup.

What if we want to show an extra <div> depending on the checkbox. So, let’s add this extra <div> to the 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> |

See that this responds to the checkbox immediately.

What if we want to manipulate this <div> via code? What if we wanted to hide/show this with the .hide() or .show()?

Then the <div> needs to be disconnected from the ViewModel. Remove the binding and add an id:

|  |
| --- |
| <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> |

Then we could manipulate this <div> via an extra observable property on the ViewModel:

|  |
| --- |
| // bad example  $(function () {      var clientViewModel = ko.mapping.fromJS(serverViewModel);        //remaining ViewModel code here      clientViewModel.DoSomethingWhenTheScarybooleanChanges                        = ko.computed(function () {                            if (this.IsScary()) {                                $('#didYouKnow').show(500);                            }                            else {                                $('#didYouKnow').hide(250);                            }                            return 'dummy';                        }, clientViewModel);        ko.applyBindings(clientViewModel);  }); |

This works fine! But this is a bad example! Why? Within the ViewModel a direct link has been added to DOM elements. You don’t want his; it pollutes the ViewModel with extra logic. The ViewModel doesn’t know about DOM elements. Period.

But how can we solve this then? In a much more elegant way…

Look, the code needs to be somewhere. But where? If we can’t place it within the ViewModel we’ll put it outside of it. And that code can only run if a certain change happens in the ViewModel.

You might recognize this as an event and that’s what we’ll apply. With Knockout this is called a .subscribe(). Replace the code of our last example with the following:

|  |
| --- |
| $(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);      }  } |

Run this and see that this still works.

# 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>

1. Communication between websites is seen a security risk. However, it is possible when using Json. This makes it possible to pass a callback function to the request. The requested site will then return the result via aformentioned function. This is outside the scope of this workshop. [↑](#footnote-ref-1)