Knockout Webinar - Documentation

Table of Contents

1. Introduction
2. Just Plain HTML
3. Sections and Require.js
4. Bootstrapping Knockout
5. Fetching data from the Server
6. Applying Bindings

1. Introduction

This ASP.NET MVC Prototype demonstrates the usage of some frameworks, which are considered as best-practice within the community. Basic functionality can be solved via jQuery plugins, but sophisticated solutions will demand solutions that leverage the MVVM pattern to ASP.NET MVC.

One possible framework is Knockout.js, since it is widely accepted and already build into the standard ASP.NET Web API template from Visual Studio.

Global JavaScript code should be avoided; therefore we need a framework that helps us to write modular JavaScript. The most used solution is require.js. It is script loader that uses the AMD module format.

In the JavaScript world the JSON format is used to exchange data. In earlier versions of ASP.NET MVC the standard controllers hat to be used to receive and return JSON. With the introduction of the ASP.NET Web API the overall amount of work to handle JSON in a RESTful way was highly simplified.

2. Just Plain HTML

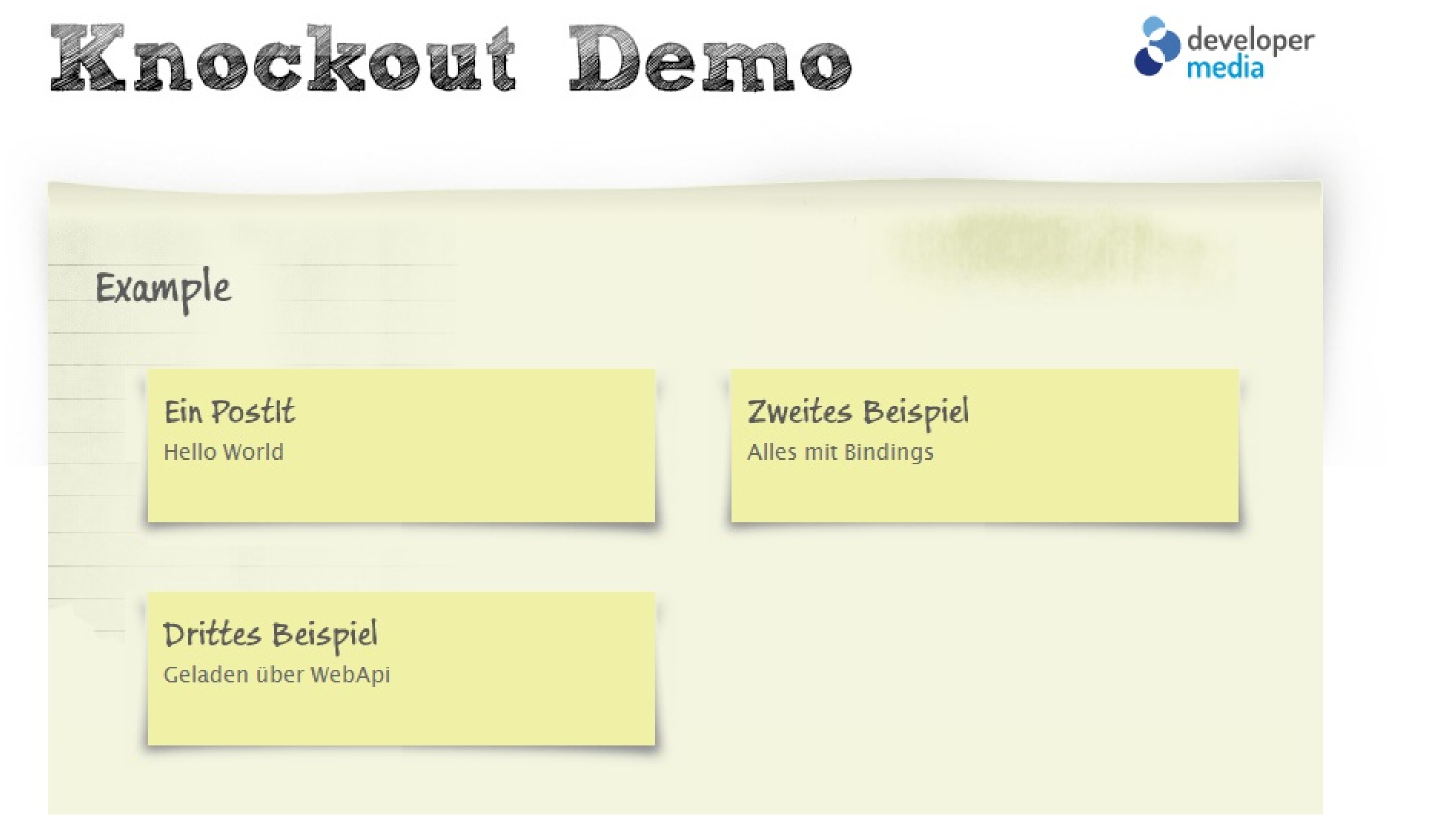
One of the biggest advantages of Knockout is the ability to directly use HTML elements. No ASP.NET user controls or heavy-loaded vendor-specific controls are required. Just good-old plain HTML:

```html
<div id="index_template">

    <h1>Header</h1>

    <div>
        <div class="drop_shadow postit">
            <h2>Title</h2>
            <p class="message">Message</p>
        </div>
    </div>
</div>
```

Together with some stylesheets this results in a bare website.



3. Sections and Require.js

For our JavaScript logic we need a starting point. We could start with same nasty inline <script> right next to our previous HTML. But first of all we will use a Razor @section, which can be rendered in an arbitrary location using the RenderSection method

```
@section scripts {
    alert("hello world");
}
```

It is considered as best-practice to place such a script section at the bottom of a page. But just adding some hacky JavaScript at the bottom of the page won't make things that much better. That’s why we will define our dependencies with require.js.

Let's start with a snipped that loads the module "indexPage", which would be usually located in a file with the same name and the .js file ending.

```
@section scripts {
    require(['indexPage'], function(i) {
        i.init();
    });
}
```

Here we load the module "indexPage" and decide to name the corresponding param-eter in the anonymous function to a shorter version "i". We then call the method "init" of the "indexPage" module. In a perfect world this would be complete setup. But often we have to deal with version numbers in files, obscured paths or files that don’t implement the AMD pattern. For all these proposes require.js can be configured. It’s a good idea to put that configuration into a new file, e.g. called require.config.js:

```
requirejs.config({
    baseUrl: "Scripts",
    paths: {
        'jquery': 'jquery-1.9.1',
        'knockout': 'knockout-2.2.1'
    },
    shim: {
        'knockout': { deps: ['jquery'] }
    }
});
```

Here we define paths and file names if the file does not match the convention. There is also a "shim" defined, that indicates that the module "knockout" is dependent upon "jquery". Require.js will make sure, that jQuery will be loaded before Knockout. Shims are also often used to point to a global variable that and AMD-ingorand script will create. (via exports)

4. Bootstrapping Knockout

A typical MVVM driven-website has three team players:

1. The model, which usually represents some business data and business logic. In our case the model "lives" in the C# world on the server side. Subsets of the data can be serialized to JSON and will be sent over the wire with the help of the ASP.NET Web API.
2. The view, it is built with plain HTML and some extra HTML5 data attributes that help Knockout to do its job.
3. The ViewModel, which glues everything together. The ViewModel is responsible for talking into both "worlds". It represents a chuck of the model data, eg. one prod-uct or a paged list of products. This data can be bound to the HTML so that the data gets visible. The ViewModel also exposed all methods that the view should call to operate on the model / business data.

The already introduced "indexPage" module will be very short. It will create a new ViewModel and apply it to the HTML.

```
define(['knockout', 'jquery', 'IndexPageViewModel'], function (ko, $, IndexPageViewModel) {

    var init = function() {

        var model = new IndexPageViewModel();
        ko.applyBindings(model, $('#index_template').get(0));
        model.loadData();
    };

    return {
        init: init
    };
});
```

5. Fetching data from the Server

Our first view model will be simple, it just loads some data and stores the content into its own property "notes".

```
define(['jquery', 'knockout', 'knockout.mapping'], function ($, ko, mapping) {

    var IndexPageViewModel = function () {

        var self = this;

        self.header = ko.observable("Example");
        self.notes = ko.observableArray(
-               [{  Title: "Notizen", Message: "werden geladen..." }]);

        self.loadData = function () {

            $.ajax('/URL').done(function (xhr) {
                self.notes = mapping.fromJS(xhr, {}, self.notes);
            });
        };
    };

    return IndexPageViewModel;
});
```

At this position many Knockout examples show an AJAX call to the server.After the AJAX call returns some data, those examples create a

ViewModel with has some initial data. In the authors opinion this violates the MVVM pattern. Loading the data should be encapsulated within the ViewModel itself. With respect to the pattern we start with an empty ViewModel and apply that empty ViewModel to the HTML. One of the biggest benefits is the subscriber / observer pattern for all ViewModels in Knockout. As soon as the ViewModel changes, the HTML will change, too.

So we are save to ask the model to load its data after the initial binding.

As soon as we get data, the HTML will magically show it.

The initial empty data is created with ko.observable and ko.observableArray. LoadData triggers jQuery to load data via GET.

After the JSON data returns from the server, jQuery will parse it and provide an object (called xhr here), that holds all the data. To save some time, we use the Knockout Mapping plugin. It will update the observableArray with the fresh JSON data. Knockout will now render a second time.

6. Applying Bindings

A test with the browser will show that the page does absolutely nothing. The table is still totally empty. This effect is by design. Knockout will not guess were to apply the ViewModel data. We will add some HTML5-compatible declarations to the html (marked green):

```
<div id="index_template">

    <h1 data-bind="text: header"></h1>

    <div data-bind="foreach: notes">
        <div class="drop_shadow postit">
            <h2 data-bind="text: Title"></h2>
            <p data-bind="text: Message" class="message"></p>
        </div>
    </div>
</div>
```

The most simple binding is the "text" binding. It just adds a text-child to the given HTML-DOM element. Knockout will check, if the ViewModel has a property that is called "Title", "Message" and so on. If it exists, it will call as a function.

In example model.Title() could return a string like "Hello World", which would then result into the following final HTML fragement:

```
<h2 data-bind="text: Title">Hello World</h2>
```