Web Developer Conference 2013

SPA Workshop - Part 1: NoSQL

Table of Contents

1. Introduction
2. Preparing a simple model
3. Saving the model to MongoDB

1. Introduction

Let's review an important part of a modern website: the communication between client and server. It should be lightweight (no SOAP) and standard conform (RESTful). On client-side we will use jQuery and Knockout. The native data format of data in JavaScript is JSON. On the server side we have an ASP.NET MVC 4 / PHP (Zend Framework) website that serves more or less static HTML content. We are free to send our data with AJAX in any format. Why shoudn't we put our data directly in JSON format into our DB?

2. Preparing a simple model

For this workshop we will use a very simple model. It is a single C# class / PHP array that represents a sticky post. It can have categories, which are just plain strings for simplicity.

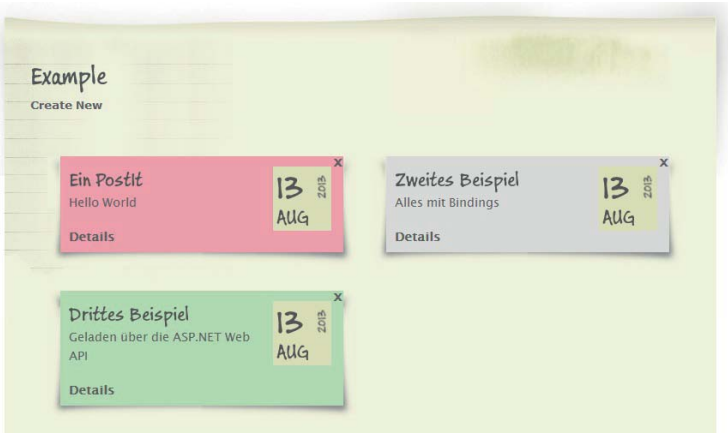
C#

```
public class Note
{
    public Note()
    {
        Categories = new List<string>();
    }

    public int Id { get; set; }

    public string Title { get; set; }

    public string Message { get; set; }

    public DateTime Added { get; set; }

    public IEnumerable<string> Categories { get; set; }
}
```

PHP

```
array(
    "Id" => "5282727b660b934d344ebbcd",
    "Title" => "Testeintrag",
    "Message" => "Ein gruener Postit",
    "Added" => "2012-06-12T22:00:00Z",
    "Categories" => ["hobby", "private"]
)
```

We will use this model to show a editable list of sticky notes:



3. Saving the model to MongoDB

Saving JSON data to MongoDB is very easy.
The following commands should be executed at the shell and will be ready to use for C# as well as PHP.

```
use WebNote
db.Notes.drop();

db.Notes.save(
  {
    "Title" : "Testeintrag",
    "Message" : "Ein gruener Postit",
    "Added" : new Date(2012, 05, 13),
    "Categories" : ["hobby", "private"]
  });

db.Notes.save(
  {
    "Title" : "Testeintrag 2",
    "Message" : "Ein roter Postit",
    "Added" : new Date(2012, 05, 14),
    "Categories" : ["important"]
  });

db.Notes.save(
  {
    "Title" : "Testeintrag 3",
    "Message" : "Ein privater Postit",
    "Added" : new Date(2012, 05, 14),
    "Categories" : ["private"]
  });
```

The following lines demonstrate the find() command. It will return an iterable cursor to one or more documents. On the command line, a cursor will be immediately iterated and displayed on screen.

```
db.Notes.find();
db.Notes.find({ Title: /Test/i });
db.Notes.find({ _id: 111 }).limit(1);
```

Web Developer Conference 2013

SPA Workshop - Part 2: REST

Table of Contents

1. Introduction
2. A RESTFul framework
3. Requesting data with jQuery
4. REST maturity and Hypermedia
5. More

1. Introduction

We decided to send and receive JSON data, lets review dedicated API that offers additional features such as a cleaner code, content-negotiation or improved routing. Both the **ASP.NET Web API** as well as the **Zend Framework** offer nearly similar features.

2. A RESTFul framework

In contrast to other technologies (e.g. the WCF - Windows Communication Foundation) the ASP.NET Web API as well as the Zend Framework are built with respect to the **"Convention Over Configuration"** principle.

Without any big configuration-ceremony we can just add a class to the project to offer one or more notes as a resource. (a resource is a source of specific information in a RESTful architecture)

This resource will be available under to URL **"/api/note"**.

In ASP.NET Web API as well as Zend the conventions start with the naming of the methods. To respond to a HTTP GET request, the method should start with the prefix "Get". Since the id was marked as optional, we can add two methods: one for a request without an ID in the URL and one for the optional id (eg. **"api/note/222"**).

C#

```
public class NoteController : ApiController
{
    public IEnumerable<Note> GetAll()
    {
        return NoteRepository.ReadAll();
    }

    public Note Get(int id)
    {
        return NoteRepository.Read(id);
    }
}
```

PHP

```
class NoteController extends AbstractRestfulController {

    public function getList() {
        return new JsonModel($this->repository->readAll());
    }

    public function get($id) {
        return new JsonModel($this->repository->read($id));
    }
}
```

RFC 2616 defines some more HTTP methods (also known as "verbs"). For the daily work the following HTTP verbs should be known.

HTTP method	Possible usage
GET	retrieve information
POST	create a new resource
PUT	update a resource
DELETE	delete a resource

3. Requesting data with jQuery

A very easy-to use AJAX api is provided by jQuery. Our simple Web API can be consumed with the following lines of code:

```
$.getJSON('/api/note').done(function (xhr) {
  console.log("AJAX Result: ", xhr)
});
```

The Web API will respond with JSON-formatted data so that jQuery can automatically convert the data to a JavaScript object. To get the note with the number 2, we just have to change the URL:

```
$.getJSON('/api/note/2').done(function (xhr) {
  console.log("AJAX Result: ", xhr)
});
```

4. REST maturity and Hypermedia

4.1 History

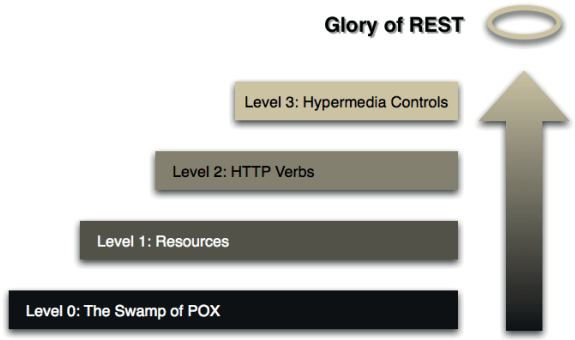
As we have seen, .NET as well as PHP offer a simple toolbox that helps us to create HTTP-based application interfaces. We were able to offer CRUD operations an a simple resource (which represented some sticky notes), but until not we did not investigated time in the general ideas behind REST at all. It's time to change this!

Roy Fielding is one of the authors of RFC 2616 (HTTP/1.1, from 1999). In his PhD thesis (from 2000) he generalized the Web's architectural principles and presented them as an architectural style, called REST (Representational State Transfer). His research about distributed application architectures and especially the chapter about REST explains the success of the web due to client-server architecture, statelessness, cacheability or layered systems. Well known building-blocks like resources and resource identifiers (URI) or representations should be used. However, due to its academic standard, it is very hard to build an valuable API just with the help of the thesis paper. REST was long forgotten, but frameworks like Ruby on Rails made it public to a wider audience.

In theory REST principles could be applied many protocols. In practice, REST is directly related to the web as we know it with its main protocol HTTP. To make things easier, Leonard Richardson proposed a classification for services on the web. Leonard's model suggest three levels of service maturity based on the support for URIs, HTTP, and hypermedia.

4.2 Richardson Maturity Model

The Richardson Maturity Model breaks down the principal elements of a REST approach into three steps.



4.3 Level Zero Services

Services of that level have a single URI and usually all requests are POST'ed to it. The old ASP.NET web services (*.asmx) or "Windows Communication Foundation (WCF)"* web services fall into that category. Client and server are usually tightly coupled to each other. Both parties must match exactly to each other. The most common protocol is SOAP which uses HTTP as a tunnel. The tight coupling mustn't be a disadvantage, since a WSDL file can describe such an API very well. However these remote procedure calls (RPC) have nothing to do with REST and are (in the authors opinion) hard to implement between different platforms.

4.4 Level One Services: Resources

The first REST level offers many URIs but only a single HTTP verb (HTTP verb == HTTP method). Due to the lack of verbs APIs of that level tend to have numerous URIs to call. It is very easy to build such an API with plain **ASP.NET MVC** actions that return JSON. Resources are usually build with a questions mark in it, where GET-parameters are used, e.g. `GET /Home/GetAllNotes?limit=200&search=test` or `GET /Home/DeleteNote?id=22`. Level one services are very hard to explore and require a very well written documentation.

4.5 Level Two Services: Verbs

Level two services host numerous resources (identified via URIs) which offer several HTTP verbs. By definition HTTP GET is a safe operation while verbs like PUT, POST or delete are unsafe operations meant to change the resources state. Level two services should also use HTTP status codes to coordinate interactions. Without any further work, the typical **ASP.NET Web API** controller with its CRUD operations (Create, Read, Update, Delete) lives at level two.

4.6 Problems at Level Two

As long as four methods (CRUD) are really enough, the default ASP.NET Web API plays very well. But for our stick note example: how should we architect a simple "deactivate" operation?

Simple Solution - URI

We could switch back to level one and build special URIs for special operations. Developers of a client will have to know in advance, in which situation those URI should be called. A well written documentation is required, but the approach works and is widely used.

Dirty Solution - POST

We could enhance the POSTed data with a new property. Instead of sending:

```
{
  "Title": "Note 1",
  "Description": "A long text"
}
```

we could hack this into our data:

```
{
  "Title": "Note 1",
  "Description": "A long text",
  "IsActive": false
}
```

Hell will freeze if we are designing an API in that fashion. This approach opens the door we countless pitfalls, eg. data loss or data inconsistency, wrong operations (eg. are we allowed to deactivate the resource by setting the value?) and very tight coupling. The deactivate operation is mixed within the data, so the client developers have to know everything about deactivation and the server developers have to reverse-engineer the lost intent from the given data! You should read more about this Data/Actions Impedance Mismatch in Sergey Shishkins blogpost.

Better Solution - PATCH

RFC 5789 (PATCH Method for HTTP) introduces a new verb that reduces the problems. A PATCH request changes just a part of the data of a resource. A PATCH in JSON format (draft!) could look like this:

```
{
  {"replace": "/IsActive", "value": false}
}
```

You can learn more about PATCH in this blog post from Mark Nottingham. However, PATCH was not intended to mask operations. We are still facing the data/actions impedance mismatch.

4.7 Level Three Services: Hypermedia

As we have seen, we need a new communication direction. Until now, the client was forced to know everything about the data and the operations that are available. A better approach would an API where the server tells the client which data and operations he wants to offer the the client. This approach leads to he most web-aware level of service supports: the notion of **hypermedia as the engine of application state** (HATEOAS). In that level resources contain URI links to other resources that might be of interest for the next action. HTML offers all required controls (such as links <a> or lists <ul>) but it was designed to present data and operations to humans, not machines. At the moment there is no (final) standard which defines an definite set of hypermedia controls, that are designed to be consumed by machines.

REST-developers must offer a starting point for a machine that wants to follow a trail of resources. To navigate through the trails we could use one of these formats. (or build our own domain-specific format)

- Collection+JSON - designed by Mike Amundsen
- **Hypertext Application Language (HAL)** - designed by Mike Kelly
- JSON-Home - designed by Mark Nottingham.

For a valuable API the RFC 6570 (URI Template) should be considered, since it describes how URI can be generated via placeholders. (e.g. `GET /api/Note?limit={limit}&search={searchPhrase}`) You should also avoid to reinvent the wheel by using already existing microformats to describe your data.

It seems that **HAL** is going to be the next widely adopted standard, since it is submitted as a internet-draft and already implemented by several libraries. The ASP.NET Web API can be easily taught to speak HAL, by implementing a dedicated MediaTypeFormatter. HAL can be expressed as JSON or XML, but until now all examples were written in JSON format, so let's continue with that.

JSON-HAL (application/hal+json) is just plain old JSON, with two reserved properties: `_links` and `_embedded`. A valid JSON-HAL representation could be this document:

```
{
  "Title": "Note 1",
  "Description": "A long text",
  "_links": {
    "self":   { "href": "/api/Note/1" },
    "next":   { "href": "/api/Note/2" },
    "search": { "href": "/api/search/Note/{searchPhrase}",
                "templated": true }
  }
}
```

The HAL draft is designed to work on GET and does not cover an easy way to describe NON-Get operations. JSON-Home goes a step further and introduces another set of vocabulary:

```
"/api/Note/1": {
   "hints": {
     "allow": ["GET", "POST", "DELETE"],
     "formats": {
       "application/json": {}
     },
     "accept-post": ["application/xml"]
   }
 }
```

I would be happy to see an API where both formats would be thrown together. Here is our domain-specific hypermedia format that would server our needs. Why should call it "application/hal+home+json-webnote"!

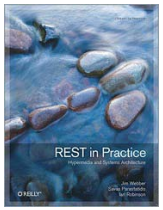
```
{
  "Title": "Note 1",
  "Description": "A long text",
  "_links": {
    "self":   { "href": "/api/Note/1" },
    "next":   { "href": "/api/Note/2" },
    "search": { "href": "/api/search/Note/{searchPhrase}",
               "templated": true },
    "deactivate": { "href": "/api/Note/2",
                    "allow": ["POST"],
                    "accept-post": ["application/json-post-webnote"]
                  }
  }
}
```

At the moment all formats are just proposals or drafts. As long as you document your API very well, I would consider the code-snipped above as a real RESTful API.

You can lean more about HAL by using the HAL browser, that helps to discover HAL APIs (including an inbuilt demo).

4. More

A great book is "REST in Practice". It will teach you the spirit of REST while being easy to understand and read!



Web Developer Conference 2013

SPA Workshop - Part 3: Single Page Application

Table of Contents

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

1. Introduction

The **C# and PHP 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. One possible framework is Knockout.js. 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.

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:

```
<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. 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 modular JavaScript should be preferred for various reasons. Therefore 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 parameter in the anonymous function to a shorter version "i". We then call the method "init" of the "indexPage" module.

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 product 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 fragment:

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

Web Developer Conference 2013

SPA Workshop - Part 4: Single Page Application

Table of Contents

1. Introduction
2. Basic Knockout Setup
3. Form bindings
4. Custom bindings
5. Submitting form data
6. Validating form data
7. Styled messages

1. Introduction

In the last part we created a first index page, which showed a list of notes with the help of the foreach-binding. This time we want to show one note and **edit** its content. During the following chapters we will learn some new bindings that are handy for processing form data.

2. Basic Knockout Setup

Similar to the last time, we start with a plain HTML page, that is going to be our Knockout-View.

```
<div id="edit_template">

    <h1>Header</h1>
    <div class="drop_shadow bigpostit">

        <form>
            <fieldset>

                <label class="rowLeft">Title:</label>
                <input class="rowRight" type="text" />

                <label class="rowLeft">Message:</label>
                <textarea class="rowRight"></textarea>

                <div class="rowLeft">Categories:</div>
                <ul class="rowRight">
                    <li>
                        <input type="checkbox" name="Categories">
                        <label>Checbox</label>
                    </li>
                </ul>
                <p>
                    <input type="submit" value="Save">
                </p>

            </fieldset>
        </form>
    </div>
</div>
```

This HTML renders to a simple form:

Again we need a module (called **'editPage'**) to wire up the ViewModel with the HTML-View. It is nearly identical the the **'indexPage'** that was shown last time. (We will care about this duplicate code later on!)

```
// editPage.js
```

```
define(['knockout', 'jquery', 'app/EditPageViewModel'], function (ko, $, EditPageViewModel) {

    var init = function (id) {

        var model = new EditPageViewModel(id);
        ko.applyBindings(model, $('#edit_template').get(0));
        model.loadData();
    };

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

As we can see, the init-Method of the editPage-Module as well as the constructor of the ViewModel accepts an **id** as a parameter. It is just a value that we have to set up in the page with the help of a classic server-side technology. This approach is very rough and does not leverage any of the advanced features that a true JavaScript-driven page could offer. (please stay tuned)

```
@section scripts  {
    var id = 22; // 22 comes from a server side technology (bad)!
    require(['app/editPage'], function(i) {
        i.init(id);
    });
}
```

It is a good idea to start with an empty ViewModel. This allows us to immediately show some initial data to the user. As soon as the initial binding was processed an AJAX call can be placed. As soon as the data arrives, the ViewModels values can be updated with the new data. Since all properties of the ViewModel are observables (which mean that changes are tracked) changes to the ViewModels properties are immediately reflected in the View.

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

    var EditPageViewModel = function(id) {

        var self = this;

        self.Id = ko.observable();
        self.Title = ko.observable();
        self.Message = ko.observable();
        self.Categories = ko.observableArray();

        self.loadData = function () {
            $.ajax('/api/note/' + id).done(function (xhr) {
                self = mapping.fromJS(xhr, {}, self);
            });
        };
    };

    return EditPageViewModel;
});
```

3. Form bindings

Until now we have only seen one usage of bindings. Data from the ViewModel changes the visible content of the View. But bindings do not only work in one direction. The content from the View can alter the ViewModel, too. One of these *two-way bindings* is the value-binding.

```
<input class="rowRight" data-bind="value: Title" type="text" />
<textarea class="rowRight" data-bind="value: Message"></textarea>
```

As soon as we change the value of the input or textarea element, the ViewModel will change accordingly.
We can check the result by adding this line of code to watch the current data of the ViewModel:

```
<div data-bind="text: ko.toJSON($root)"></div>
```

A more complex scenario can be resolved by using the checked-binding. Knockout will set an radiobutton or checkbox to be checked if the value matches an item in an array. When the user checks the associated form control, this updates the value on your ViewModel. Likewise, when you update the value in your ViewModel, this checks or unchecks the form control on screen.

First we should define the array, it holds all possible values. In our case it stores the three possible categories that a note can have. An adequate position for that array would be the ViewModel itself. Since these three possible categories never change, we can use a native JavaScript array:

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

    var EditPageViewModel = function(id) {
        /// [...]
        self.CategoryChoices = ['important', 'hobby', 'private'];
        /// [...]
    };
    return EditPageViewModel;
});
```

We can now iterate over the defined values with the help the foreach-binding to render three checkboxes (and a nice label for convenience).

```
<div id="edit_template">

    <h1 data-bind="text: 'Details of Note No. ' + Id()"></h1>
    <div class="drop_shadow bigpostit">

        <form>
            <fieldset>

                <label class="rowLeft">Title:</label>
                <input class="rowRight" data-bind="value: Title" type="text" />

                <label class="rowLeft">Message:</label>
                <textarea class="rowRight" data-bind="value: Message"></textarea>

                <div class="rowLeft">Categories:</div>
                <ul class="rowRight" data-bind="foreach: CategoryChoices">
                    <li>
                        <input type="checkbox"
                               name="Categories"
                               data-bind="attr: { value: $data, id: 'label_categories_' + $data},
                                          checked: $root.Categories">

                        <label data-bind="attr: { for: 'label_categories_' + $data }, text: $data"></label>
                    </li>
                </ul>
                <p>
                    <input type="submit" value="Save">
                </p>

            </fieldset>
        </form>
    </div>
</div>
```

The **$data** variable is referring to the current array entry. **$index** refers to the current zero-based index of the array item. You can use **$parent** to refer to data context from outside the foreach. Since foreach-bindings can be nested, the **$root** context always refers to the topmost context.

4. Custom bindings

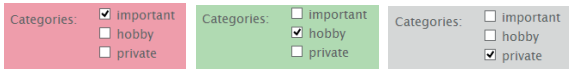
There is no limitation to use the built-in bindings like text, click, value, and so on — you can create your own ones. Let's interact with the ViewModels categories in a visual way. We want to color the note in **red**, if the category is 'important', in **green**, if the category is 'hobby' or in **gray** if the category is 'private'. If the there are multiple choices, red will be chosen before green and green will be chosen before gray.

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

    var colorMapping = [
        { category: 'important', color: "red" },
        { category: 'hobby',     color: "green" },
        { category: 'private',   color: "gray" }];

    ko.bindingHandlers.choseCategoryColor = {
        update: function (element, valueAccessor) {

            var chosenColor, categories = ko.utils.unwrapObservable(valueAccessor());

            // remove all already existing CSS classes
            $.each(colorMapping, function (index, mapping) {
                $(element).removeClass(mapping.color + "Color");
            });

            // find new class
            $.each(colorMapping, function (index, mapping) {
                if ($.inArray(mapping.category, categories) !== -1) {
                    chosenColor = mapping.color;
                    return false;
                }
            });

            if (chosenColor) {
                $(element).addClass(chosenColor + "Color");
            }
        }
    };
});
```

This new binding can be used like the internal ones. So if we can change `<div class="drop_shadow bigpostit">` to

```
<div class="drop_shadow bigpostit" data-bind="choseCategoryColor: Categories">
```

The result is a nicely formatted sticky note that changes it CSS class immediately after a change of the checboxes.



5. Submitting form data

A traditional HTML form has the big disadvantage that all form data is basically just a bunch of strings. The internal format of the data as well as types are lost during the conversions. With the MVVM pattern we go a big step forward. The HTML form and the ViewModel are in sync with each other. Since they are in sync, we do **NOT** need to submit the original form. Instead of the real form we can send to ViewModel's data to the server!

Usually a form can be submitted by a click on the "submit-button" or by hitting enter in a text field. We could use a click-binding on the submit button. However, the submit-binding has the advantage that it also captures alternative ways to submit the form.

```
<form data-bind="submit: saveForm">
```

Of course, the "saveForm" method must be defined in the ViewModel, too.

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

    var EditPageViewModel = function(id) {

        var self = this;

        self.Id = ko.observable();
        self.Title = ko.observable();
        self.Message = ko.observable();
        self.Categories = ko.observableArray();
        self.CategoryChoices = ['important', 'hobby', 'private'];

        self.loadData = function () {
            $.ajax('/api/note/' + id).done(function (xhr) {
                self = mapping.fromJS(xhr, {}, self);
            });
        };

        self.saveForm = function () {

            $.ajax({
                url: '/api/note',
                type: 'put',
                data: ko.toJSON(self), // <-- !!!!
                contentType: 'application/json'

            }).fail(function () {
                alert('error');
            }).done(function () {
                alert('success');
            });
        };
    };

    return EditPageViewModel;
});
```

6. Validating form data

Many frameworks have been written to validate form data. No one will ever satisfy all requirements. Since the topic is knockout we will look on YET ANOTHER solution to solve this immortal hubris on client side. You should be warned that client-side validation adds comfort and responsiveness to you application. *But in every real life code server-side validation is still required, to avoid tampered data!*

We will use the popular Knockout Validation plugin. It is highly configurable and plays well with MVVM pattern. It's also available via Nuget (Nuget is available for the .NET guys only), so we should grab it via

```
PM> Install-Package Knockout.Validation
```

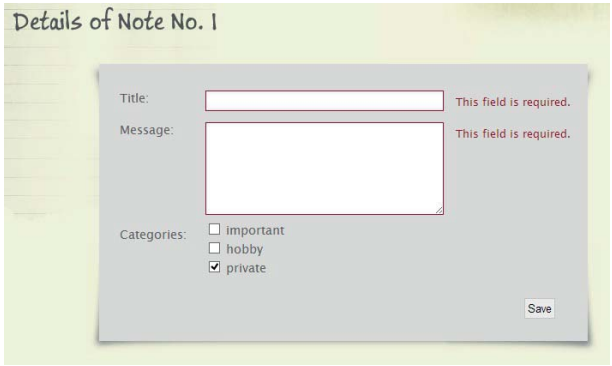
Knockout validation uses extenders to augment already existing observables. We can define our validation rules by extending the observables with one of the list of predefined rules:

```
var EditPageViewModel = function(id) {

    var self = this;

    ko.validation.configure({ decorateElement: true });

    self.Title = ko.observable().extend({ required: true });
    self.Message = ko.observable().extend({ required: true, minLength: 3, maxLength: 1000 });
};
```

It a common scenario to find out, if on of the observable is currently invalid and therefore if the whole ViewModel is invalid, too. For this purpose Knockout validation introduces the 'validatedObservable' which can be used like this:

```
self.watchValid = ko.validatedObservable({
    Title: self.Title,
    Message: self.Message
});

// valid or invalid?
var valid = self.watchValid.isValid()
```

Knockout validation does a lot of work in the background and displays a predefined error message next to the control that holds invalid data. The config option 'decorateElement' makes sure that the invalid control gets a new CSS class that can be used to style it.



7. Styled messages

The visible-binding can be used to show simple but but peachy confirmation messages. Imagine three elements that are styled via CSS:

```
<div class="success">Data was successfully saved!</div>
<div class="error">There was an error during saving!</div>
<div class="info">Data was automatically saved!</div>
```

They produce the following output.

We should introduce a new property to represent the current **status** of an operation. This property could be a simple string or an complex object for advanced options. Now we can avoid silly alert-messages and inform the user by just changing the property.

```
var EditPageViewModel = function(id) {

    var self = this;
    self.status = ko.observable('');

    self.saveForm = function () {

        if (!self.watchValid.isValid()) {
            self.status('error');            // <-- !!!!
            return;
        }

        $.ajax({
            url: '/api/note',
            type: 'put',
            data: ko.toJSON(self),
            contentType: 'application/json'

        }).fail(function () {
            self.status('error');            // <-- !!!!
        }).done(function () {
            self.status('success');          // <-- !!!!
        });
    };
};
```

The corresponding knockout binding is simple but effective:

```
<div data-bind="visible: status() == 'success'" class="success" style="display: none">Data was successfully saved!</div>
<div data-bind="visible: status() == 'error'" class="error" style="display: none">There was an error during saving!</div>
<div data-bind="visible: status() == 'info'" class="info" style="display: none">Data was automatically saved!</div>
```

Web Developer Conference 2013

SPA Workshop - Part 5: Single Page Application

Table of Contents

1. Introduction
2. All templates together
3. The app
4. The app state
5. Events

1. Introduction

Until now we mainly concentrated on bindings for forms and validation. Lets review one possible effective architecture for a Knockout-driven single-page application **(SPA)**.

2. All templates together

Normal websites force the users to make small breaks each time a new page is loaded. This pattern is not acceptable if we want to call our product an "application". For a (desktop) application the reaction time is usually very fast. Any user action should directly result in visible output. The previous demo had a classical server-side routing and a continued delivery of HTML for each rendered page. Bundling is the key. To deliver all views together we do not have to change very much of the current architecture.

We are going to rename and edit both views (index.cshtml & edit.cshtml).

```
<!-- before: Index.chtml -->
<div id="index_template">
  [...]
</div>
```

```
<!-- after: _index.chtml -->
<section id="index_view"
         data-view-model="IndexPageViewModel"
         data-bind="template: { name: 'index_template' }">
</section>

<script type="text/html" id="index_template">
  [...]
</script>
```

The type "text/html" is unknown for the browser. So all it the content within that <script$gt; tag will be ignored. But we can still use it as described in the documentation of the template-binding. The attribute "data-view-model" is not knockout-related. We will use it as a hint to chose the correct ViewModel.

We can new load all <section> tags (which are initially completely empty and therefore invisible) together with a new init-script:

```
<!-- new: Index.chtml -->
@Html.Partial("~/Views/Home/_index.cshtml")
@Html.Partial("~/Views/Home/_edit.cshtml")

@section scripts  {
    require(['singlePage/appState', 'knockout.bindings'], function(appState) {
        appState.init();
    });
}
```

3. The app

The **app** is the central module of the website. It replaces both 'indexPage.js' and 'editPage.js' which had nearly duplicate content. With each 'loadView' call the app shows one of the <section> tags, applies the already known bindings to it and hides the previews shown section.

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

    var currentView;
    var events = $({});

    var loadView = function(viewId, param) {

        unloadCurrentView();
        currentView = $("#" + viewId + "_view");           // loads one of the sections, eg. 'index_view'

        var viewModelName = currentView.data("viewModel"); // retrieves the ViewModel name...
        if (viewModelName) {                               // ...as stored in data-view-model="IndexPageViewModel"

            events.trigger('loadView');
            require(['app/' + viewModelName], function (ViewModelConstuctor) {

                var model = new ViewModelConstuctor(param);    // same pattern as in the old 'indexPage' module
```

```
                ko.applyBindings(model, currentView.get(0));
                currentView.show();

                model.loadData(function() {
                    events.trigger('viewLoaded');
                });
            });
        }
    };

    var unloadCurrentView = function() {                    // cleans up to avoid memory-leaks
        if (currentView) {
            currentView.hide();

            ko.cleanNode(currentView.get(0));
            currentView.unbind();
            currentView = undefined;
        }
    };

    return {
        loadView: loadView,
        events: events
    };
});
```

4. The app state

The application should know which page is currently shown. It should also know which pages are in the browser history and how a click on the "browser-back" button should be handled. Let's call this the "state" of the application. The **appState** module internally uses the small but powerful framework Sammy for a client-side routing. This routing is similar to the ASP.NET MVC routing, but it works completely on the client with the help of internal anchor #links. This can be seen on the browsers URL which changes from `http://localhost/#/` to `http://localhost/#edit/1` without any real page reload.

```
// singlePage/app.js
define(['singlePage/app',
        'jquery',
        'sammy',
        'singlePage/bindLoadingIndicator',                      // see chapter Events
        'singlePage/bindRefreshPage'], function (app, $, sammy) {

    var sammyApp;

    var init = function() {

        // Client-side routes
        sammyApp = sammy(function () {

            this.get('#/', function () {
                app.loadView('index');                          // <-- !!!
            });

            this.get('#:viewId', function () {
                app.loadView(this.params.viewId);
            });

            this.get('#:viewId/:param', function () {
                app.loadView(this.params.viewId, this.params.param);
            });

            this.notFound = function() {
                app.loadView('page404');
            };

        }).run('#/');
    };

    var changeState = function (newViewId, newParam) {

        var newLocation = !newParam ? "#" + newViewId :
                                      "#" + newViewId + "/" + newParam;
        sammyApp.setLocation(newLocation);
    };

    var reload = function() {
        sammyApp.refresh();
    };

    return {
        init: init,
        changeState: changeState,
        reload: reload
    };
});
```

5. Events

You probably have noticed, that were trigger two jQuery events (loadView & viewLoaded) to indicate a change on the displayed views. The first events fires immediately, the second one fires after the content was loaded. To archive this delay, we use a callback that we added to the method 'loadData':

```
var IndexPageViewModel = function () {

    var self = this;

    self.loadData = function (callback) {

        $.ajax('/api/note').done(function (xhr) {
            self.notes = mapping.fromJS(xhr, {}, self.notes);
            callback.call(self);                                // <-- !!!
        });
    };
};
```

These two events leverage a flexible way to do additional tasks.
In example, if we load this module it will automatically show a loading indicator during the waiting time:

```
// bindLoadingIndicator.js
define(['jquery', 'singlePage/app', 'jquery.loadingIndicator'], function ($, app) {

    var bindLoadingIndicator = function () {

        var main = $('#main');

        app.events.bind('loadView', function() {

            if (!main.data('loadingIndicator')) {
                main.loadingIndicator();
            }
            main.data('loadingIndicator').show();
        });

        app.events.bind('viewLoaded', function () {

            if (!main.data('loadingIndicator')) {
                return;
            }
            main.data('loadingIndicator').hide();
        });
    };

    $(bindLoadingIndicator);
});
```

We might also want to run additional code after a view was rendered.
So let's just wait for the 'viewLoaded' event:

```
// bindRefreshPage.js
define(['jquery', 'singlePage/app', 'jquery.plugins'], function ($, app, cufon) {

    var bindRefreshPage = function () {

        app.events.bind('viewLoaded', function () {
            $.refreshPage();
        });
    };

    $(bindRefreshPage);
});
```
