



# Knockout Webinar 2 - Documentation

## 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

Last time 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>

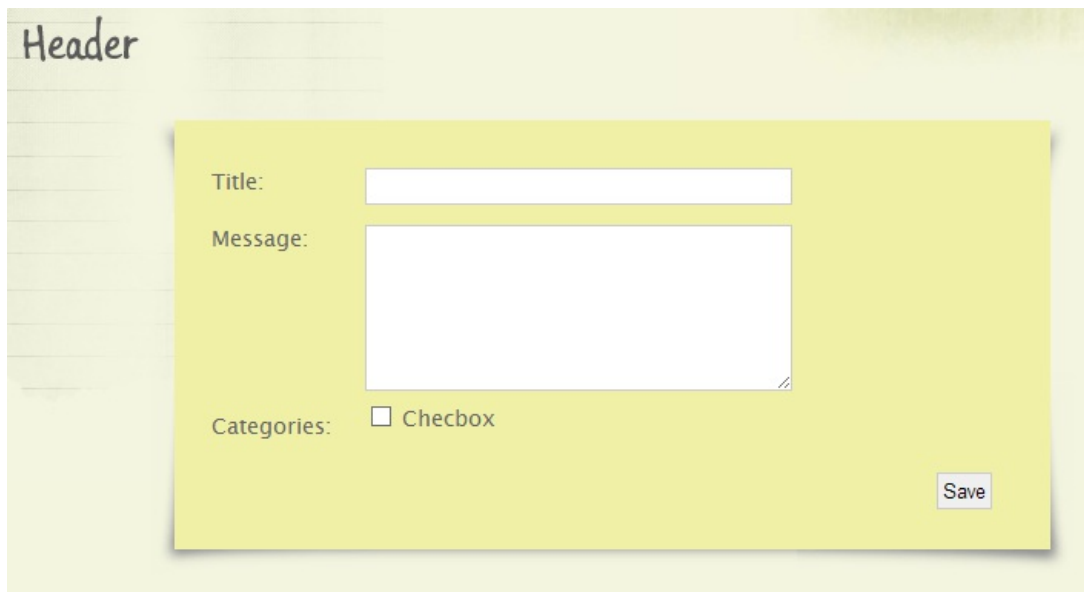
        <label class="rowLeft">Title:</label>
        <input class="rowRight" type="text" />

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

        <div class="rowLeft">Categories:</div>
        <ul class="rowRight">
          <li>
            <input type="checkbox" name="Categories">
            <label>Checkbox</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 set up in the [@section](#) of start the page with the help of a Razor-engine placeholder. 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 = @ViewBag.Id; // Razor engine here
    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 of 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 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 its CSS class immediately after a change of the checkboxes.

Categories: ☒ important  
☐ hobby  
☐ private

Categories: ☐ important  
☒ hobby  
☐ private

Categories: ☐ important  
☐ hobby  
☒ private

## 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 your 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, so we should grab it via

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

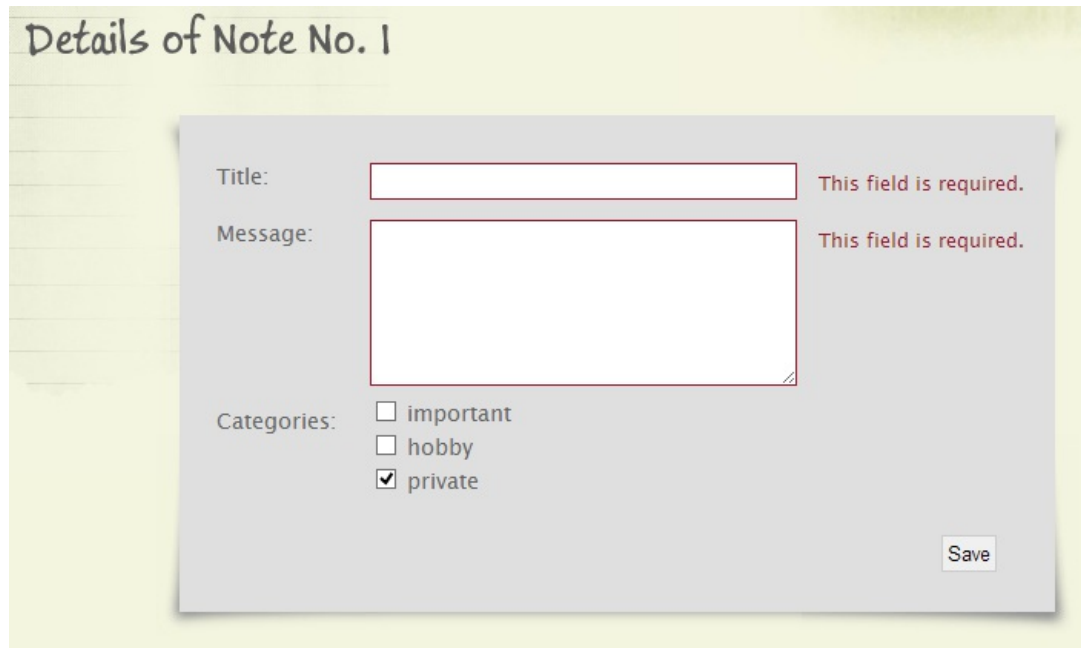
Knockout validation uses [extenders](#) to augment already existing observables. We can define our validation rules by extending the observables with one of the [list of predefined rules](#):

```
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 figure 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 <div> elements that are styled via CSS:

```
<div class="success">Data was sucessfully 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.



Data was successfully saved!



There was an error during saving!



Data was automatically saved!

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>
```