Knockout Webinar 2 - Documentation

Single Page Extra

Table of Contents

1. Introduction

Until now the second webinar mainly concentrated on bindings for forms and validation. This extra document shows a simple but 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). The "@section scripts" can be removed.

```
<!-- 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);
});
```

6. More

This was a very rough overview. There are several details to discover.
You should start by downloading the sources of the Single Page app.

» Download Demo-Code (.zip)