LAB 6: Single page application architecture

Contents

Introduction 1
Resources 1
Solutions 1
Task 1: Loading data 2
Task 2: Storing data 5
Task 3: Navigation 7

Introduction

In this lab you will build a single page application based on the MVVM pattern, using *Knockout.js*. The application will demonstrate loading and saving data and also navigating between different states.

The lab consists of three tasks.

Resources

- Lecture notes and sample code from Module Website
- Visual Studio and Firefox/Firebug (or any combination of editor/browser)
- JavaScript reference, e.g. **https://developer.mozilla.org/en/JavaScript/Reference**
- Other references as required, e.g. jQuery, Knockout documentation.

Solutions

A complete solution is included in the starter project for reference.

Task 1: Loading data

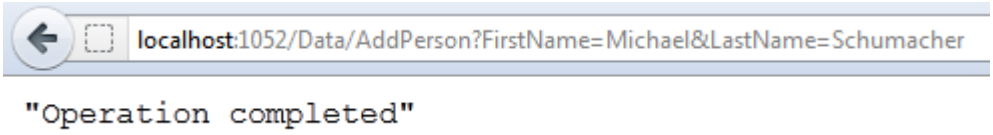
In this task you will begin building a single page application which displays data which is in the form of a list of names of persons, and provides a form which allows data for a new person to be entered and stored. Retrieving and storing data is done using Ajax calls, which interact with a database on the server side. Initially your application will simply display data.

1. Download and unzip *LAB_Knockout.zip*, and open the Visual Studio project which it contains. This project includes the following:

   - A model class *Person*
   - An data context class *LabKnockoutContext*
     (these classes together allow Entity Framework Code First to create a simple database to store instances of *Person*)
   - A controller *DataController*, which includes action methods which can be called to retrieve and store data using the data context
   - A controller *HomeController* which is there to allow your single page application to be accessed
   - *Index.cshtml,* in the Views/Home folder, which is the file you will be working with – there is also a file *Solution.cshtml* which is a completed version for reference

2. The focus in this task is on client code, so the server code has been completed for you. However, you need to use this to create the database and test the data access functionality on the server side before proceeding with client development.

   Run the application. The home page should show only a hyperlink, which will not do anything useful yet. Access the following URL, which sends parameters to an action method which should bind these to a Person object and store this in the database. You should see a response as shown below.

   Note that this make take some time as with Code First, the first time the data context is used it will create a new database, on the local SQLEXPRESS instance by default.

3. Access the following URL, which retrieves all data in the database and returns it as JSON. You should see a response as shown below, and if so the server side components are working correctly. You can add a few more people to the database as in step 2 to give an initial working data set.


```
[{"PersonId":1,"FirstName":"Michael","LastName":"Schumacher"}]
```

4. Open *Index.cshtml*. Add a script reference to *knockout-2.0.0.js*, which is provided for you in the *Scripts* folder.

5. In the script tag, immediately after the jQuery document ready function, add a constructor function *Person* which accepts parameters and assigns these to properties firstname and lastname. The properties need to be Knockout observables, e.g:

```
this.firstname = ko.observable(firstname);
```

6. Add a further constructor function ViewModel with:

   - a property *self*, which is a reference to the current value of *this* – you can use *self* to prefix other properties and methods and refer to these in your view model code using the prefix
   - a property *persons* which is an empty observable array
   - a method *loadPersons*, which sends an Ajax call to the URL */Data/GetPersons*, and uses the result to populate the *persons* array with a new instance of *Person* for each object in the returned JSON array. Refer to the lecture sample code (*PackageInfoKnockout.cshtml*) to help with this.

7. There is one Knockout template provided for you. Create, within the <head>, another template with id *personlist-template*, which displays a table row containing values of bound *firstname* and *lastname* properties.

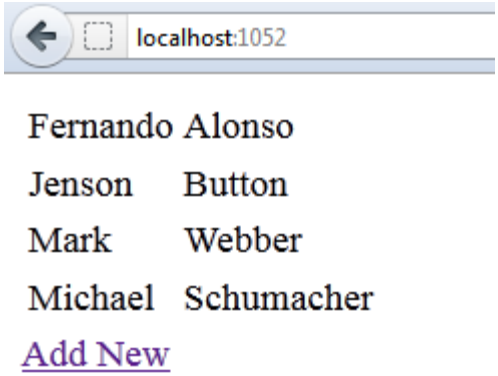
8. Add the following HTML within the <body> to create a table element which uses this template with a *foearch* binding to bind each element of the *persons* array in your view model to a row of the table.

```
<table data-bind="template: { name: 'personlist-template',
   foreach: persons }">
</table>
```

9. Add the following code within the document ready function. This creates the view model object and applies bindings, then loads data.

```
var viewModel = new ViewModel();
ko.applyBindings(viewModel);
viewModel.loadPersons();
```

   Access the default URL. You should see a listing of the data you stored in steps 2&3, similar to the following:



You have now built an application which uses a view model and a template to load and display data.

Task 2: Storing data

In this task you will extend your single page application to include a form for defining and storing new persons.

1. Extend your *ViewModel* constructor function with the following:

   - a property *person* which is a new instance of *Person*
   - a property *message* which is an empty observable
   - a function *addPerson* which sends an Ajax POST request to the URL */Data/AddPerson*, where the posted data is the a reference to the *person* property of the view model. Note that this will work because that *person* property will be bound to the input elements displayed in an HTML form, and so will represent the data entered by the user. The success callback should:
     - use the response data to set the *message* property
     - set the *firstname* and *lastname* properties of the *person* property to "" (to clear the form once the data has been stored)
     - show an alert containing the *message* property (just for simplicity here – this should really be bound to a UI element for display)
     - calls *loadPersons* to load the current data – note that this will populate the *persons* array with the updated data, and since this array is bound to the UI, will display it immediately

   Note that setting values of properties which are observables is done like these examples:

   ```
   self.message(json);
   self.person.firstname("");
   ```

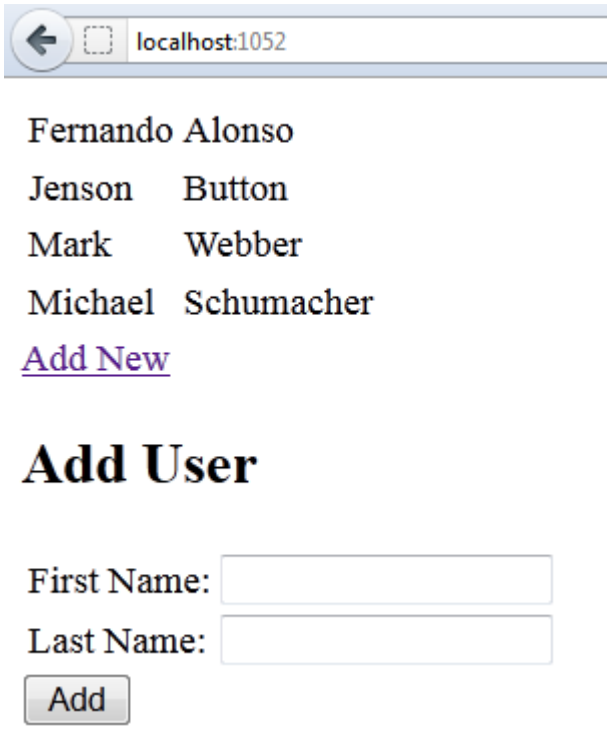
2. Add a new template called persondata-template within the <head>. This template should display two table rows, each of which displays a label and an input box (type="text"). The id's of the inputs should be "FirstName" and "LastName". The value of each input should be bound to properties firstName and lastName. This template will actually be nested inside the provided template personform-template, which displays a table and a button. When nested, your template will be applied to the person property of the view model, so the firstName and lastName properties are accessible within the your template.

3. Look at the provided template personform-template. Note the the click event of the button is bound to the addPerson method of the view model.

4. Add the following HTML element within the <body> after the existing elements. This displays the contents of the form template within a div element.

   ```
   <div data-bind="template: { name: 'personform-template'}"></div>
   ```

5. Reload the page in the browser. A form should now be displayed after the list.



6. Enter some data and click Add. You should see an alert box, and your new data should be added to the database and loaded back immediately and displayed in the list.

   You have now built an application which uses a view model and templates to load, store and display data.

Task 3: Navigation

In this task you will extend your single page application to include distinct states, one for listing data and one for entering data, and enables navigation between these states. Navigation will follow a natural sequence in which the user clicks a link to get to the form and submitting the form navigates back to the list. However, the browser back/forward buttons and directly entered URLs will also navigate between the states.

To achieve this you will use a routing library called *Sammy.js*. Note that this is essentially doing the job which Microsoft's SPA template gives to *History.js/Nav.js*.

1. Add a script reference to *sammy.js*, which is provided for you in the *Scripts* folder.

2. Add the following methods to the view model constructor. Note that these simply set the fragment identifier in the browser's address bar, and do nothing else.

   ```
   self.showform = function () { location.hash = "form"; };
   self.showlist = function () { location.hash = "list"; };
   ```

   Arguably, this pollutes the view model with DOM references. However, this is contained inside methods which clearly have a navigation purpose, and the window.location object could be mocked for testing purposes.

3. Find the link element in the HTML, and bind its click event to the *showform* method of the view model.

4. Add a call to the *showlist* method in the success callback of *addPerson* in the view model.

5. Reload the page in the browser and click the link. You should see *#form* appended to the URLSubmit some data. You should see *#list* in the URL instead of *#form.*. Nothing else new will happen as you haven't defined the UI states yet.

6. In the HTML code, wrap the markup for the list and the markup for the form in separate elements, with id's list and form respectively. Each of these divs will contain a distinct state of the UI.

7. Add the following code in the document ready function. This code uses *Sammy.js* to define routes which map URL fragments to states. For each route, there is a function which puts the page into the required state, in this case simply by hiding one div and showing the other. A default route is also defined, which maps a URL with no fragment to the state represented by the *#list* fragment.

```
Sammy(function () {
    this.get('#form', function () {
        $("#list").hide();
        $("#form").show();
    });

    this.get('#list', function () {
        $("#form").hide();
        $("#list").show();
    });

    this.get('', function () { this.app.runRoute('get', '#list') });
}).run();
```

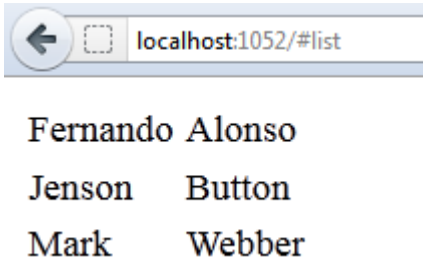
8. Reload the page in the browser. Make sure before you do so that the address bar contains a URL with no fragment. You should see only the list of persons displayed, not the form.

9. Click the link. You should now only see the form, and the corresponding fragment should be shown in the URL.

10. Look in the console in Firebug, if you are using Firefox. You should see routing events logged by Sammy.

```
[Thu Mar 08 2012 00:04:10 GMT+0000 (GMT Standard Time)] body runRoute get /
[Thu Mar 08 2012 00:04:10 GMT+0000 (GMT Standard Time)] body runRoute get #list
[Thu Mar 08 2012 00:05:58 GMT+0000 (GMT Standard Time)] body runRoute get /#form
```

11. Enter some data and click the button. You should now see only the list, with the new data included, and again the corresponding fragment should be shown in the URL.



12. Try clicking the browser Back and Forward buttons, which should navigate between states (without posting data). Try changing the URL fragment by typing directly in the address bar. Again, this should allow you to navigate between states.

    You have now built a single page application which uses a view model and templates to load, store and display data, and integrates a routing engine with your view model to allow navigation between page states in a natural flow of events and also using the unique URLs and browser history.