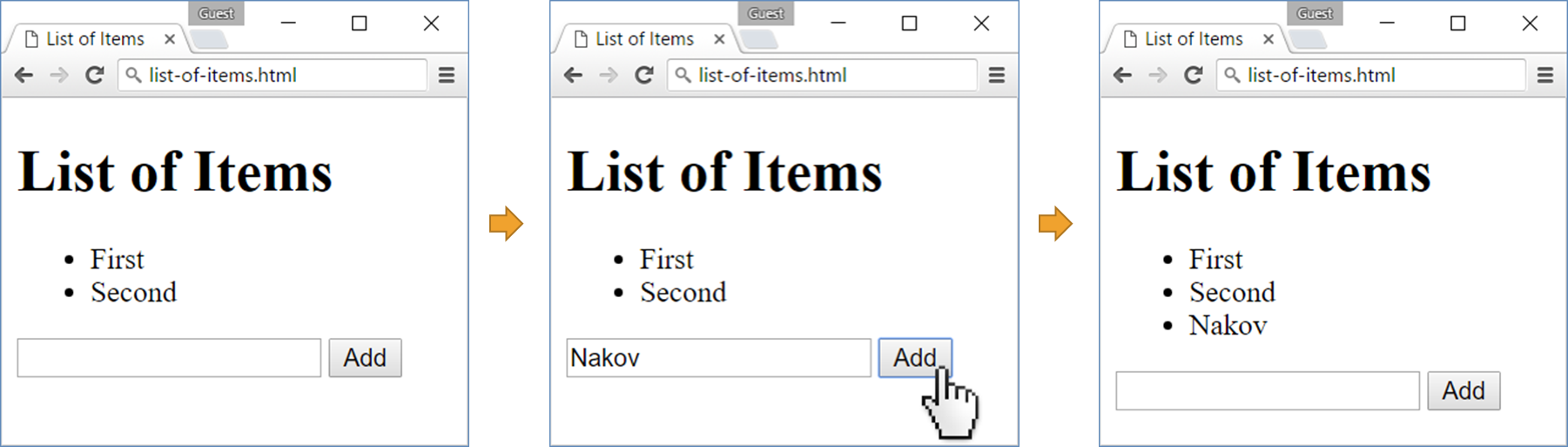
# Exercises: jQuery and AJAX

Problems for exercises and homework for the [“Software Technologies” course @ SoftUni](https://softuni.bg/courses/software-technologies).

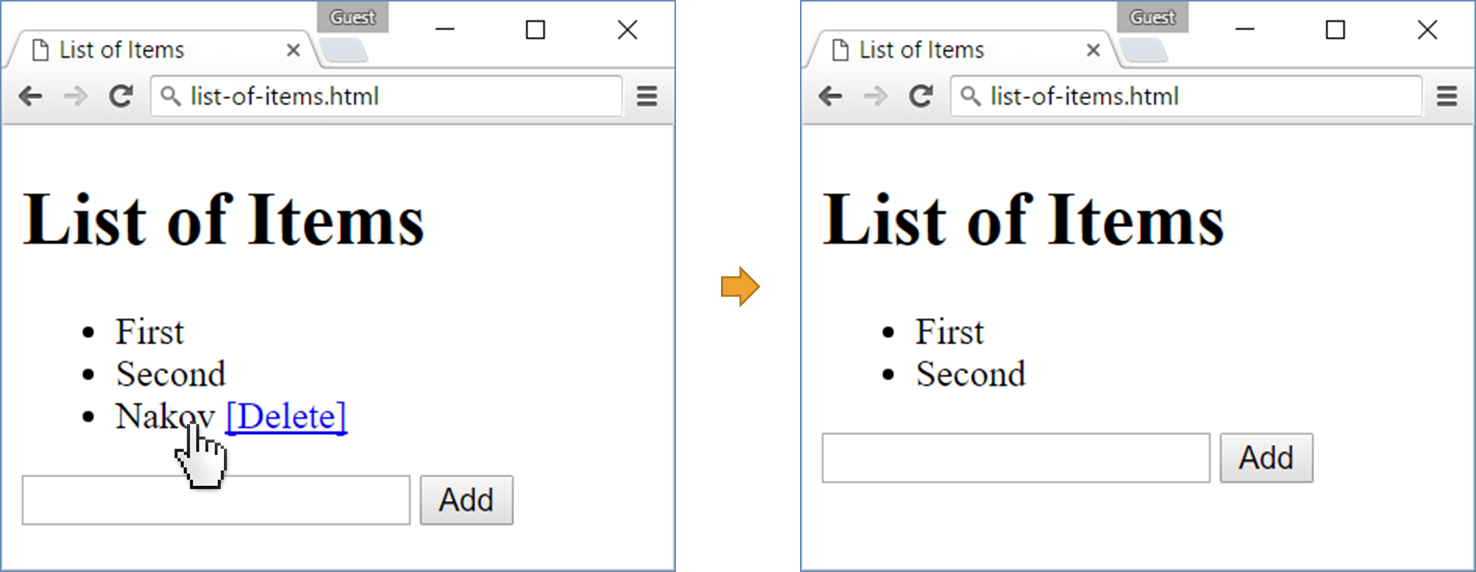
## Add / Delete Items

Create a dynamic HTML page to hold and edit a **list of items**. Implement **add** + **delete** functionality in JS.

**Adding** items could look like this:

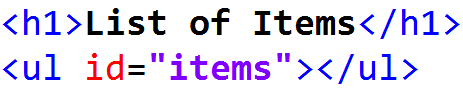


**Deleting** items could happen by showing a **[Delete]** link when the mouse is over some item:



### Hints

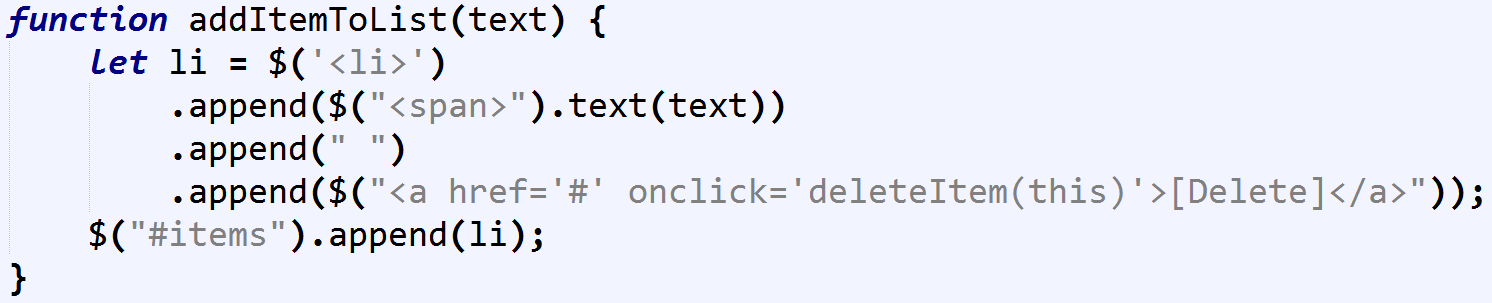
* Create your HTML page, e.g. items.html. Put an empty items list in the body:



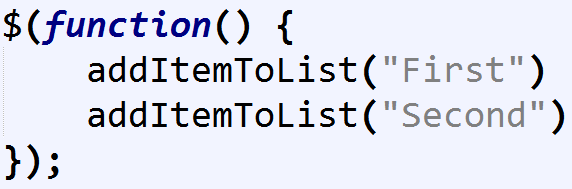
* Include the **jQuery** library to simplify the DOM manipulation:
  + Download the jQuery code from <https://code.jquery.com/jquery-3.0.0.min.js>.
  + Save jquery-3.0.0.min.js in the current folder (where items.html stays).
  + Reference jQuery in the <head> section of your HTML:  
    
* For each item in the list you should add in the DOM tree the **text** and **“delete” link** after it:
  + **<span>*item text*</span>**
  + **<a href="#" onclick="deleteItem(this)>[Delete]</a>**

Note that the function deleteItem() takes this as parameter, which holds the clicked hyperlink object from the DOM tree. It will be used later to find the list item for deleting.

**Adding a new item** to the list could be done by the following JS code using jQuery:

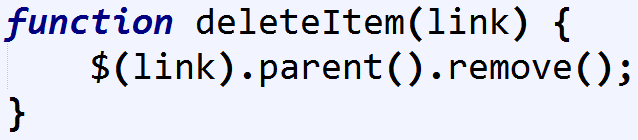


* Add some items in the list to test the above function. You may use the following code:

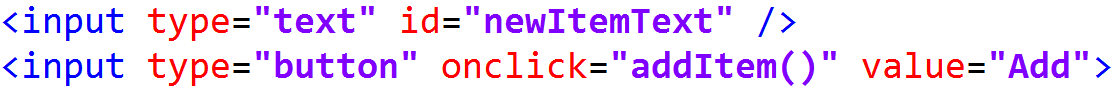


This function will be executed after the HTML page is fully loaded by the browser.

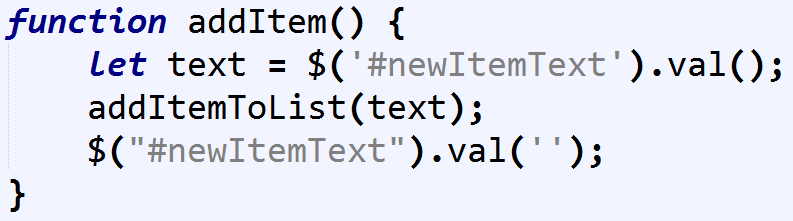
* On click on the **delete link**, delete its parent. Note that the function takes the hyperlink object from the DOM tree as input parameter:



* Test the delete link for the items on the page by clicking on it. It should work correctly.
* Now add the “**create new item**” functionality. First create the **input text box** and **[Add] button**:

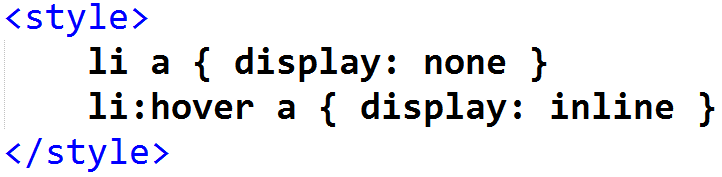


Next, write the JavaScript code to handle the **[Add] button** click:

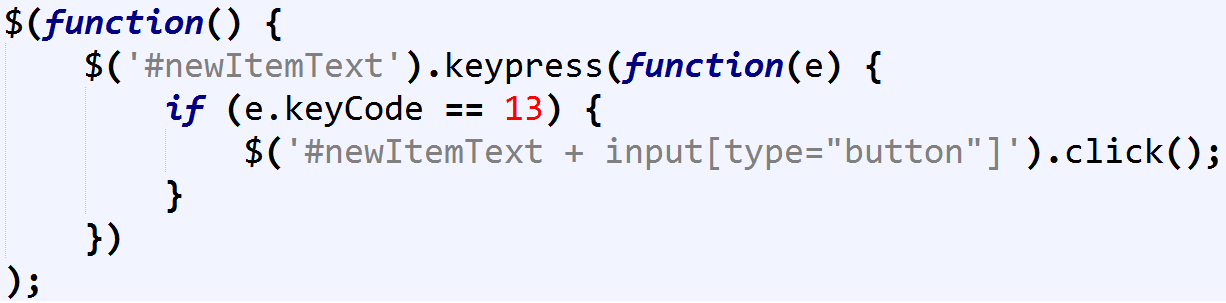


It finds the text typed in the text box and calls the addItemToList(text) function to add new item to the list, then clears the text box (puts empty text in it).

* Finally, make the **[Delete] links** invisible unless the mouse is positioned over some of the list items. The easiest way to do this is by adding the following **CSS style** in the HTML <head> section:



* Optionally, make pressing **[Enter]** key in the text box to add new item without clicking on the **[Add]** button:



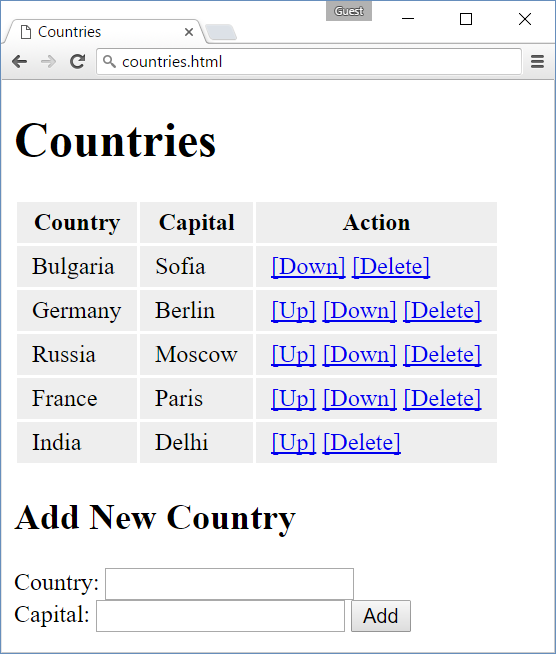
The above code attaches to handle the **“key press” event** for the input text box and when the **[Enter]** key is pressed (key code == 13), it finds the **submit button** on the right on the text box and clicks it.

## Countries Table

Create a dynamic HTML page to hold **countries** with their **capitals** in a table (see the screenshot). Implement the following functionality:

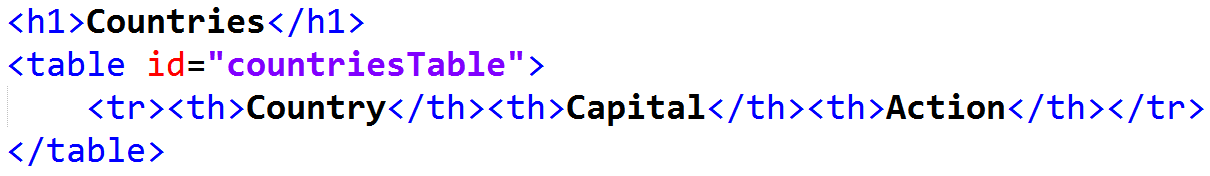
* Users can **add** new country, move country **up** / **down** and **delete** existing country.
* Each country can be **moved up** (except the first), **moved down** (except the last) and **deleted** (after confirmation).

Use **JavaScript** and **jQuery** to implement the requested functionality.



### Hints

* Create your HTML page, e.g. countries.html. Put an empty table with column headers in the body:

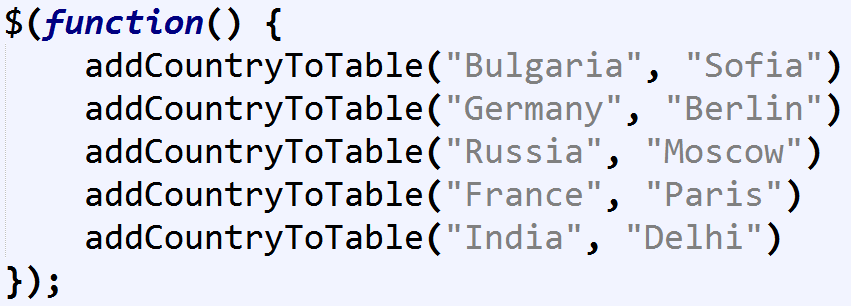


* Include the **jQuery** library to simplify the DOM manipulation:
  + Download the jQuery code from <https://code.jquery.com/jquery-3.0.0.min.js>.
  + Save jquery-3.0.0.min.js in the current folder (where items.html stays).
  + Reference jQuery in the <head> section of your HTML:  
    
* Create a JavaScript function to **add a country** to the table (country + capital + action links):

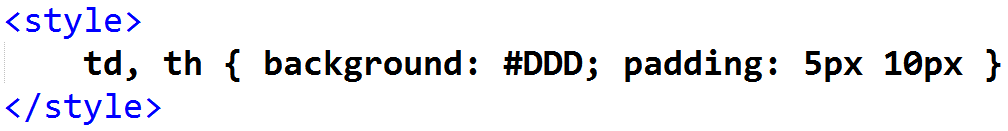


It returns the new added row as a result. You may need it later.

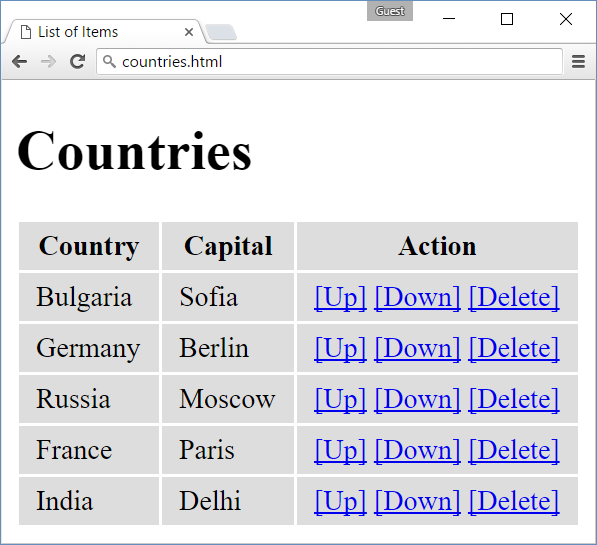
* Add a few countries to test the above code:



* Add some CSS styles in the table **<head>** section to make the table look better:

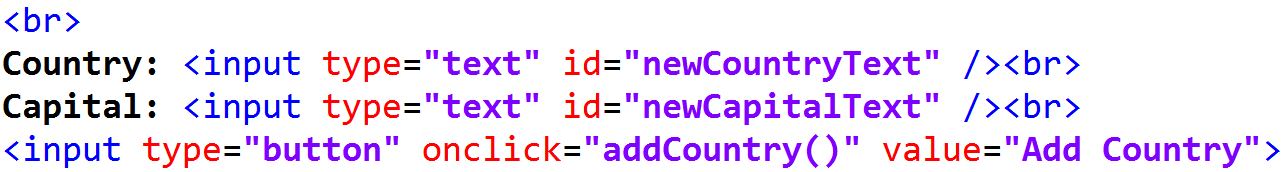


* **Test your code** to ensure it works (even when it is not yet finished):

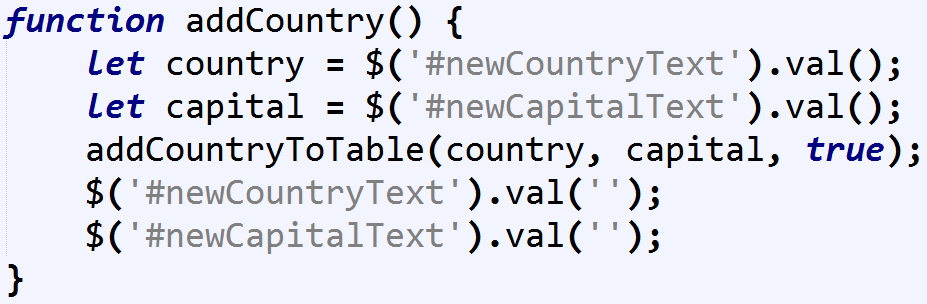


Still the action links will not work, but the countries should be shown in the table.

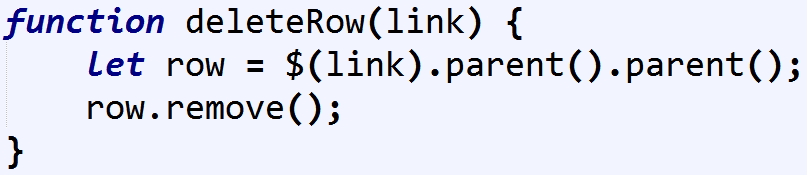
* Now, let’s **add the “create new country” functionality**. First add the input fields and **[Add Country]** button:



Then, write the **JS code** for adding a country in the table:

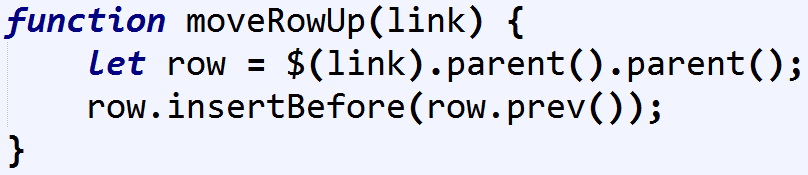


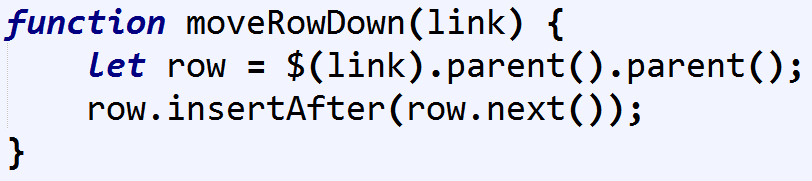
* Let’s implement the “**delete row**” functionality. Write the JS function to handle the **[Delete]** button click:



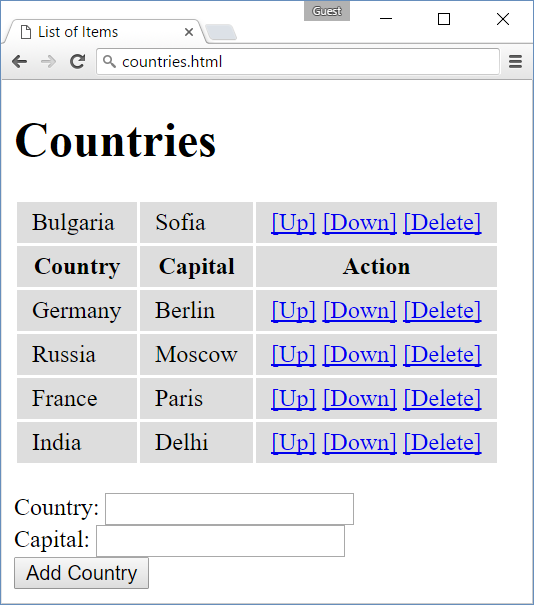
**How it works?** The function takes as input the DOM element holding the **[Delete]** hyperlink. Its parent is the table cell (<td>). Its parent is the table row for deleting (<tr>). It removes the row from the DOM tree.

* Next, let’s implement “**move row up**” and “**move row down**” functionalities:

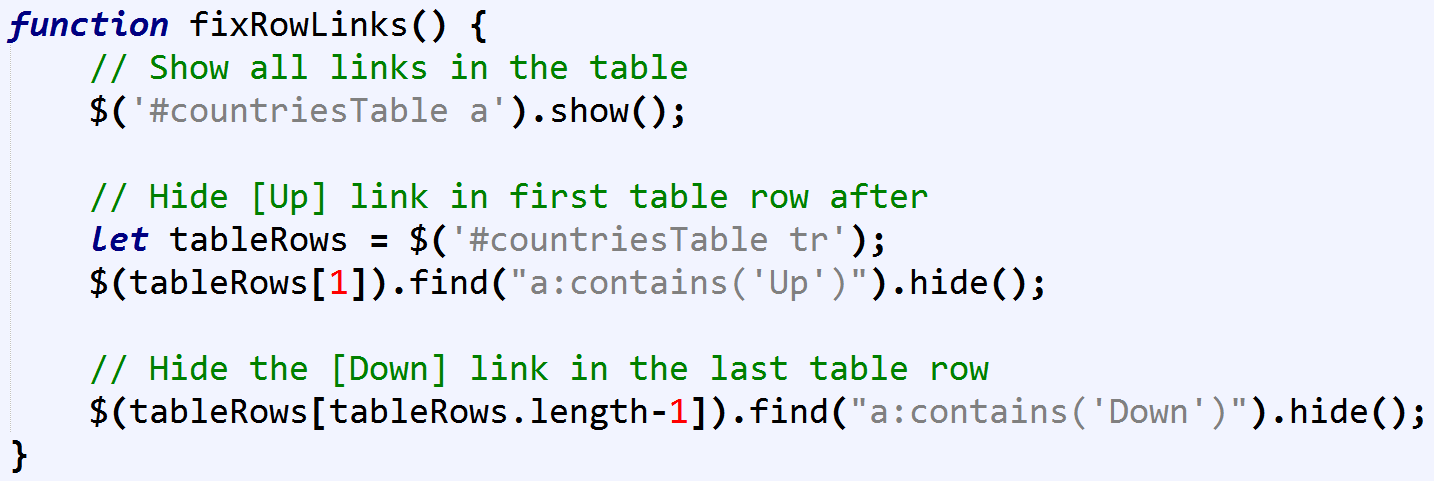




* Now **test your code** again. It is almost ready, but still needs some details to be improved. It does not hide the link **[Down]** at the last row and the link **[Up]** at the first row after the table header:



* Let’s implement **hiding** the **[Up]** and **[Down]** links when not needed. This JS function will do the job:

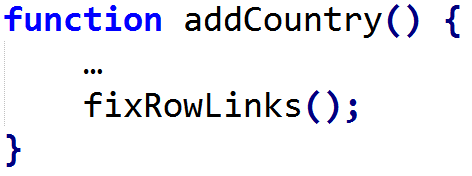
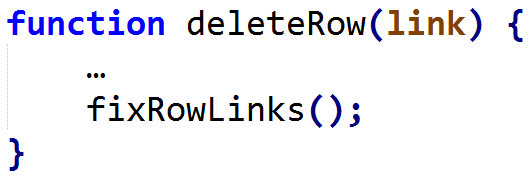


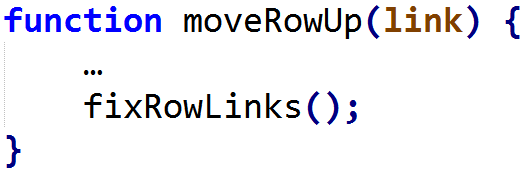
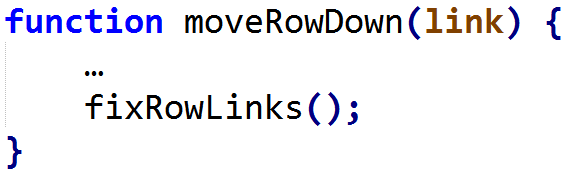
**How it works?** First, show all links in the table. Select them with jQuery selector and invoke show().

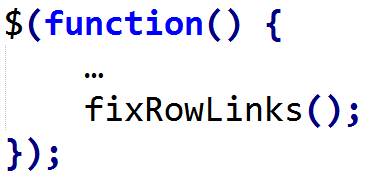
Later, in the first table row after the heading row (row 1), find all links holding the text “Up” and hide them.

Finally, in the last table row (length-1), find all links holding the text “Down” and hide them.

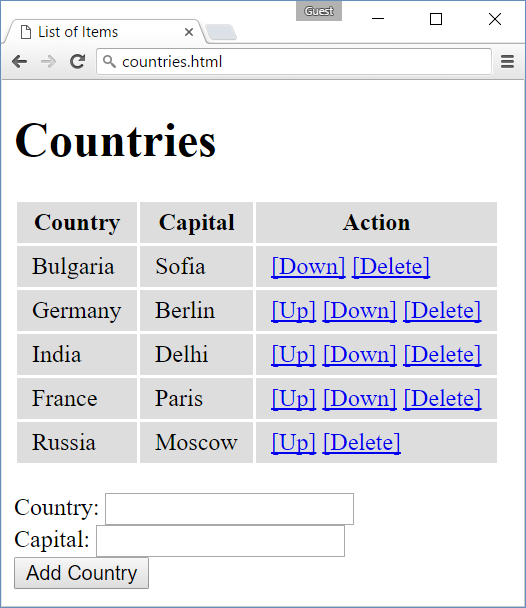
* Now we have the code that **hides** the unneeded **[Up]** and **[Down]** links. Put it at the end of all functions that **modify the table**:

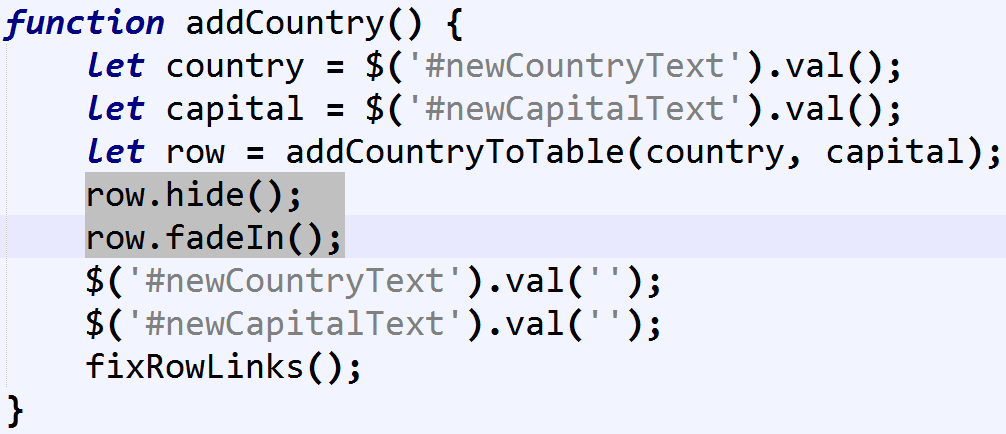
 

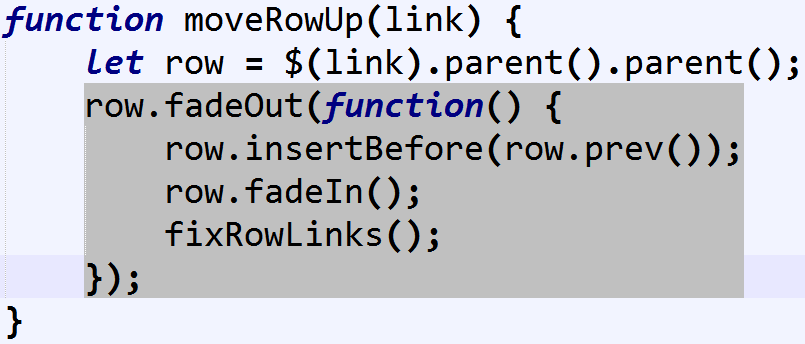


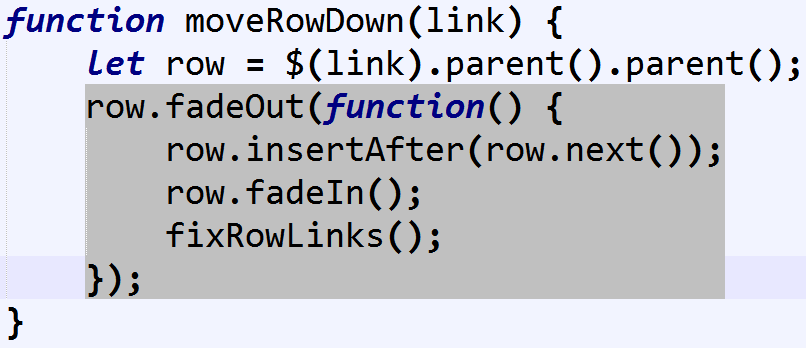
* **Test your code** again. It should work correctly. The first and last data rows should have no links for moving up and down respectively. This works correctly after using **[Delete]** and **[Add Country]**. Test is well.

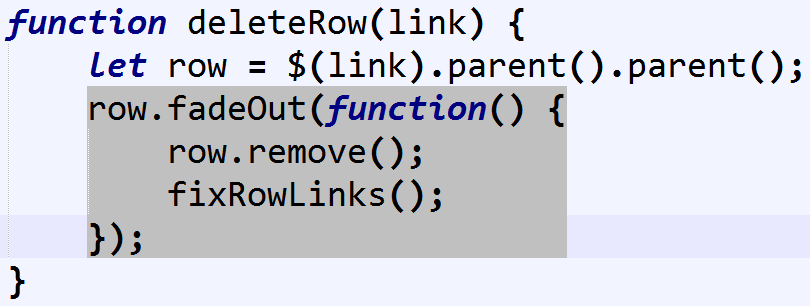


* Finally, let’s make things look and behave better. Add **transition effects (animation)** after **add** / **delete** / **move up** / **move down**. The **user experience** will be improved very much, try it! We shall use the jQuery fadeIn() and fadeOut() functions which run asynchronously and take a onComplete function callback to perform some action after the fade effect is completed:









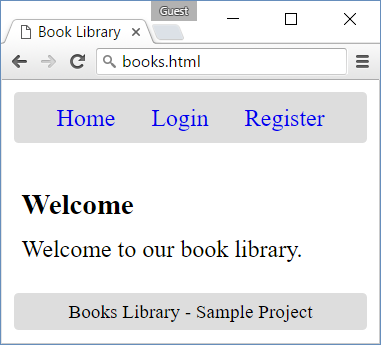
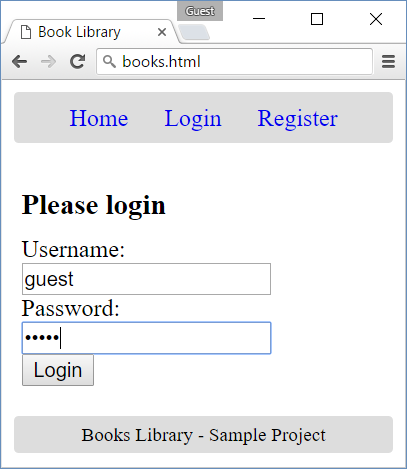
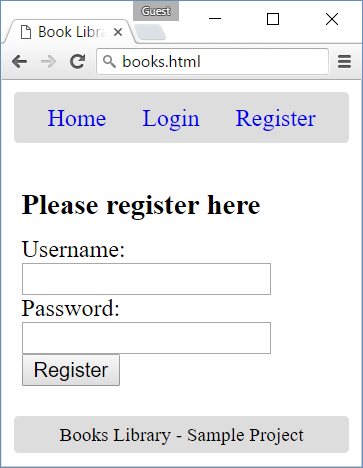
* Test you code and **enjoy the fading effects**!

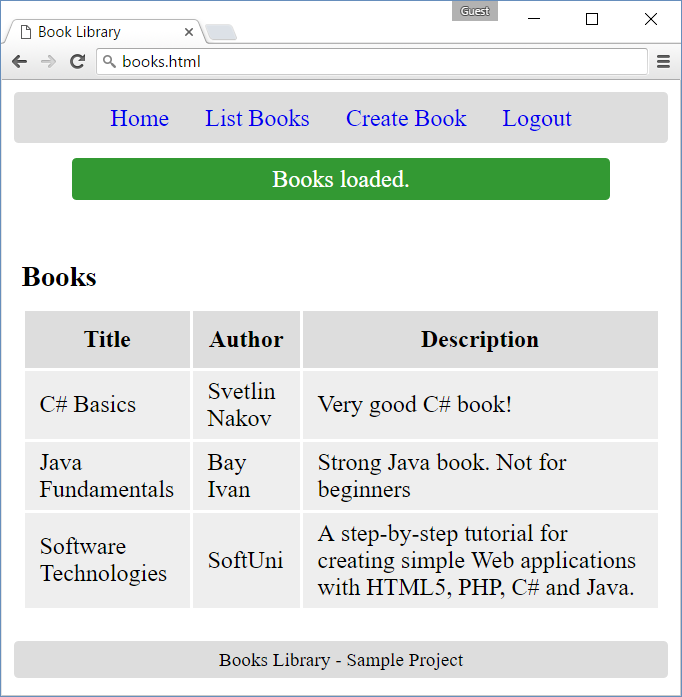
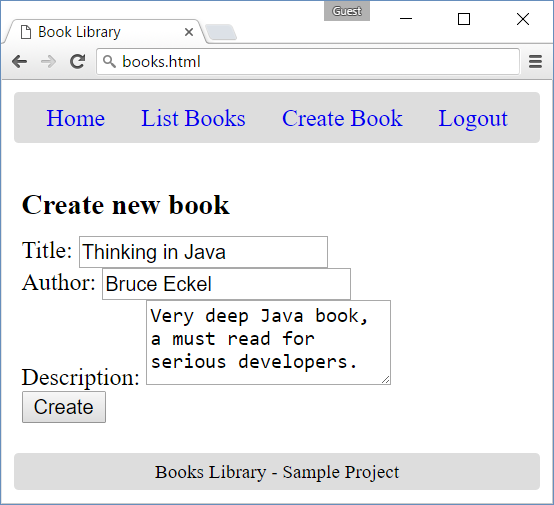
## Book Library

Design and implement a **“Book Library” front-end application** in HTML5 with REST back-end in Kinvey. Create a HTML page (JS single-page app) with several buttons (links) with the following functionality:

* **Login**
  + Login in Kinvey with existing username + password.
  + In case of success, a session key should be stored somewhere in the application.
* **Register**
  + Register a new user in Kinvey by username + password.
  + In case of success, a session key should be stored somewhere in the application.
* **Logout**
  + Logout from Kinvey and forget the session key from the application.
  + This button is available after successful login only.
* **List All Books**
  + Load the books from Kinvey and display them in the HTML page (in some predefined container).
  + This button is available after successful login only.
* **Create a New Book**
  + Shows a **“create new book” form**. When the form is submitted, a new book is added in Kinvey.
  + This button is available after successful login only.

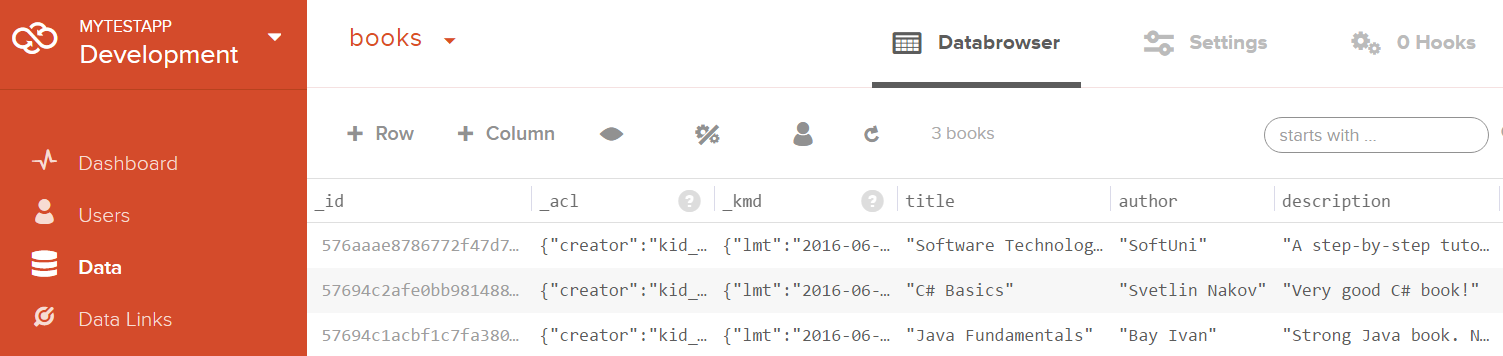
This is how your HTML page may look like:

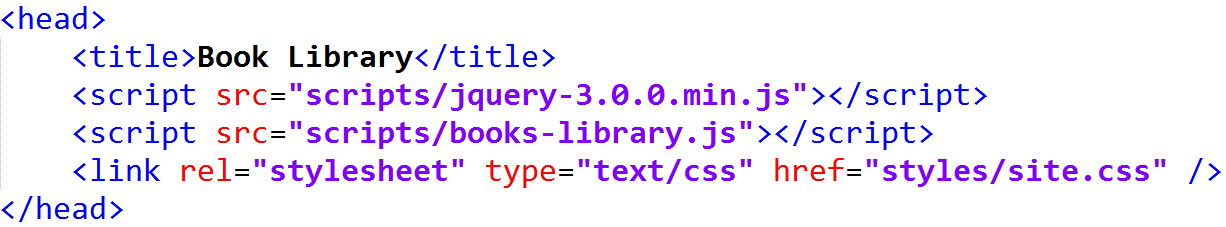
 

### Hints

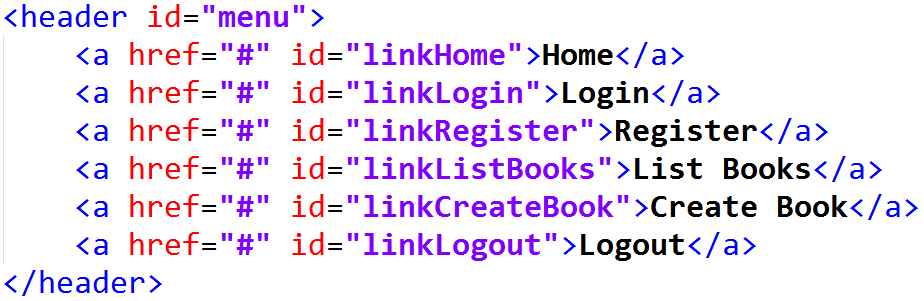
* Register in Kinvey and create a “BookLibrary” **app**.
* Create a **collection** “books” to hold book **title** + **author** + **description**.
* Create a few **sample books** in the “books” collection from the Kinvey management console.



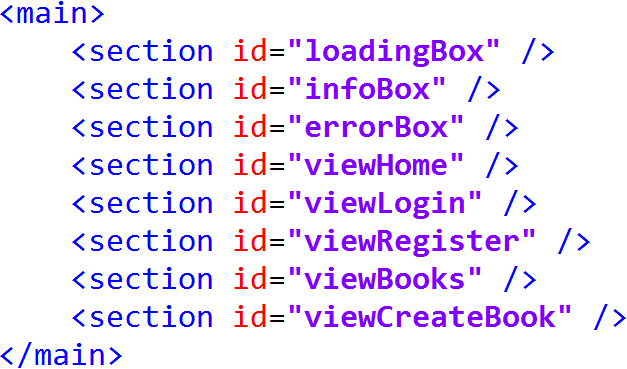
* Create an empty HTML page books.html.
  + The HTML page will hold the entire functionality of the application, structured in **views** (html <section> areas): one view for the login form, another view for the register form, another view for listing the books, etc.
* Create the CSS and JS files for the application:
  + Create an empty CSS file styles/site.css and link it from the HTML <head>.
  + Create an empty JavaScript file scripts/books-library.js and link it from the HTML <head>.
  + Download the **jQuery library** in scripts/jquery-3.0.0.min.js (ver. 3.0.0, compressed) and link it from the HTML <head>.



* The next few steps aim to create a <header>, <main> and <footer> sections. Let’s start with the **header**. It will hold the **navigation links** (some of them will be visible, some hidden):



* The <main> section will hold the **views** (sub-pages) of the application:



The loadingBox, infoBox and errorBox will show loading message / notification messages / error messages (when available). The home / login / register / books / create book views will hold the home text / login form / register form / list of books / create book form. These views will be created in the next steps.

* The **footer** is very simple:



* Next, **create the views**, one by one. Start with the loadingBox, infoBox and errorBox:



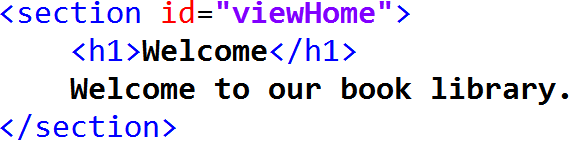




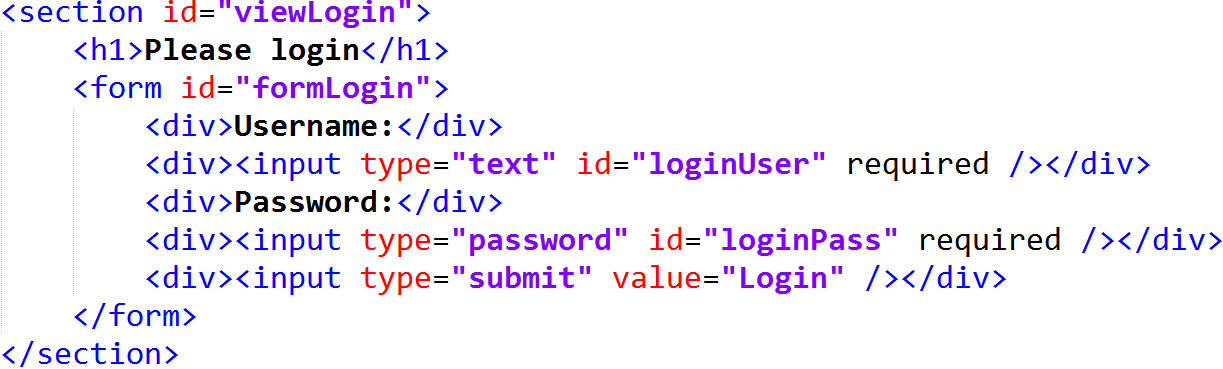
The **loading box** will show “**Loading …**” when the application waits for the server to load some data.

The **info box** and **error box** sections **hide slowly when clicked**. The idea is to have an easy way to show **loading progress notification**, **info** **notifications** and **errors**.

* The **home view** is trivial. It holds no logic, just says “welcome”:



* The **login view** holds the login form, that will be used for user login:

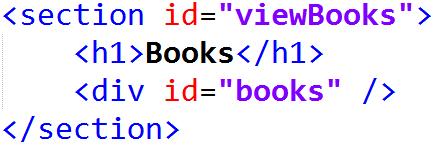


It encapsulates a simple **validation**, using the required attribute for the input boxes. The validation will be enforced when the form is **submitted**.

* The **register view** is very similar:



* Next comes the **books view**, which will list the books from the back-end service after an AJAX call:

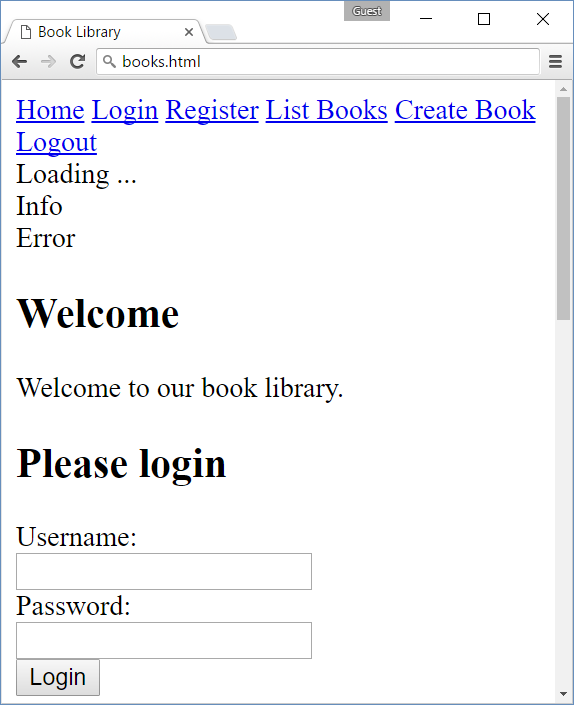
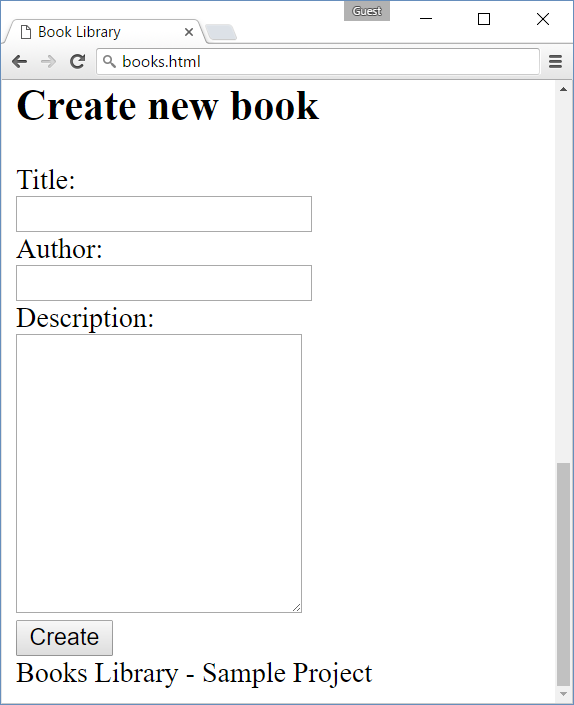


The **books** <div> inside the view will hold the books in the form of a table.

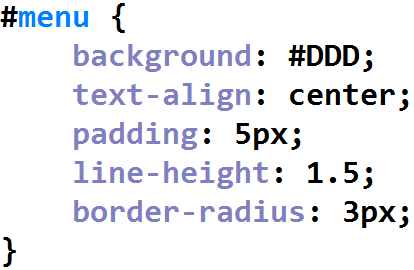
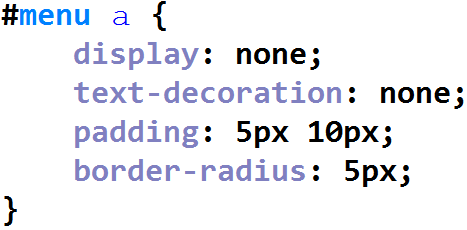
* Finally, define the “**create book**” **view**, which holds the create new book form:

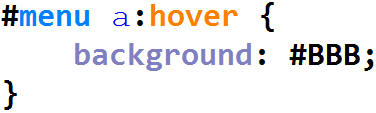


* Let’s **test the HTML page** of the application, just to see how it looks like:

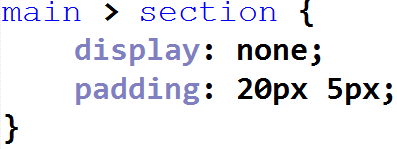
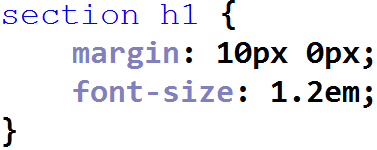
 

* Definitely, it needs some **CSS** to look better, right? Let’s add some CSS styles. First **style the header** and the navigation links in it:

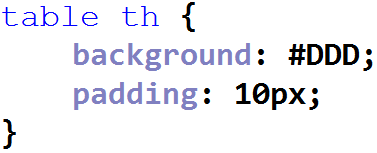
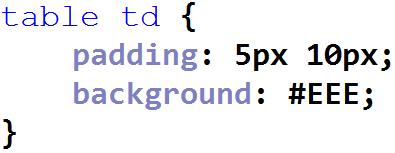
 



* Next, **style the main container** and the views in it:

* Add some simple **styling for tables** (for the header cells and row cells). It will be needed when listing books:

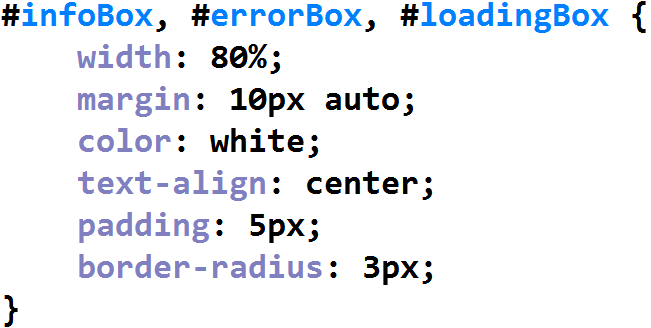
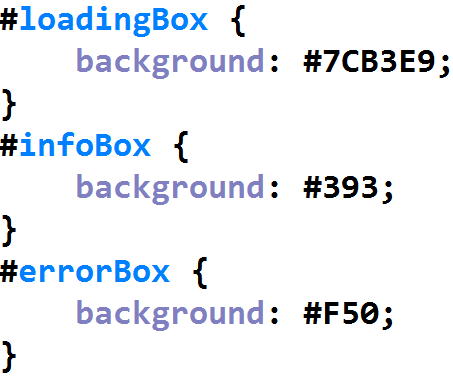
* Add styling for the **blue** **loading progress box**, which will show “Loading …”. Add styling for the **green** **info box**, which will hold notifications for successfully performed operations. Also add styling for the **red** **error box**, which will hold error notifications. The goal is to have the following look for these notifications:



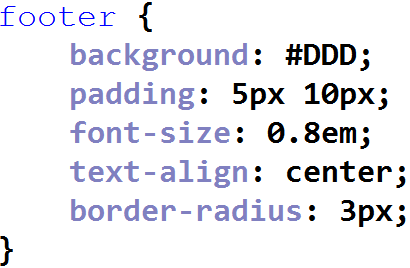




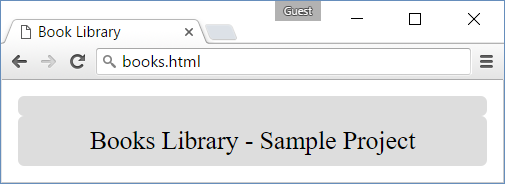
The **CSS styles** might look like these:

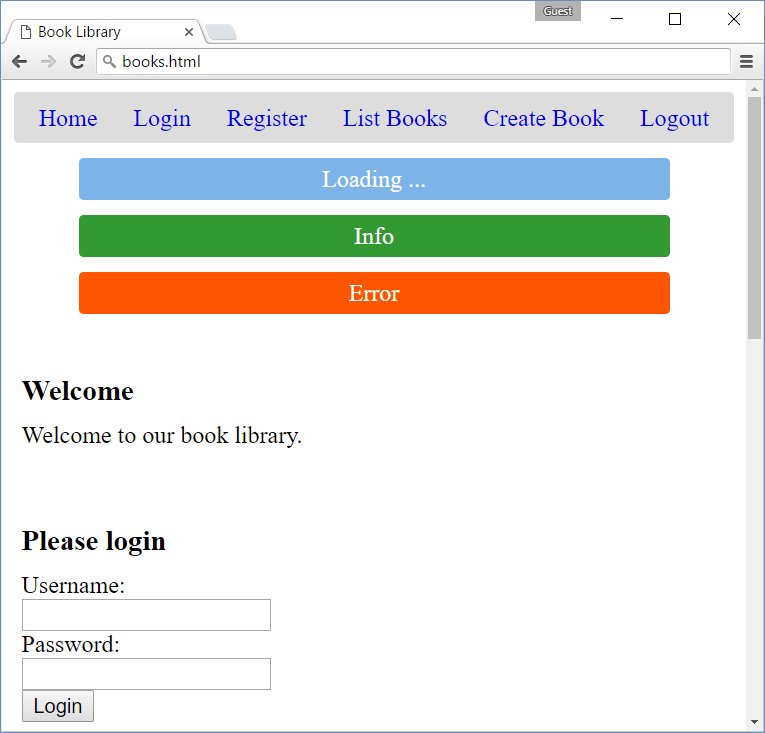
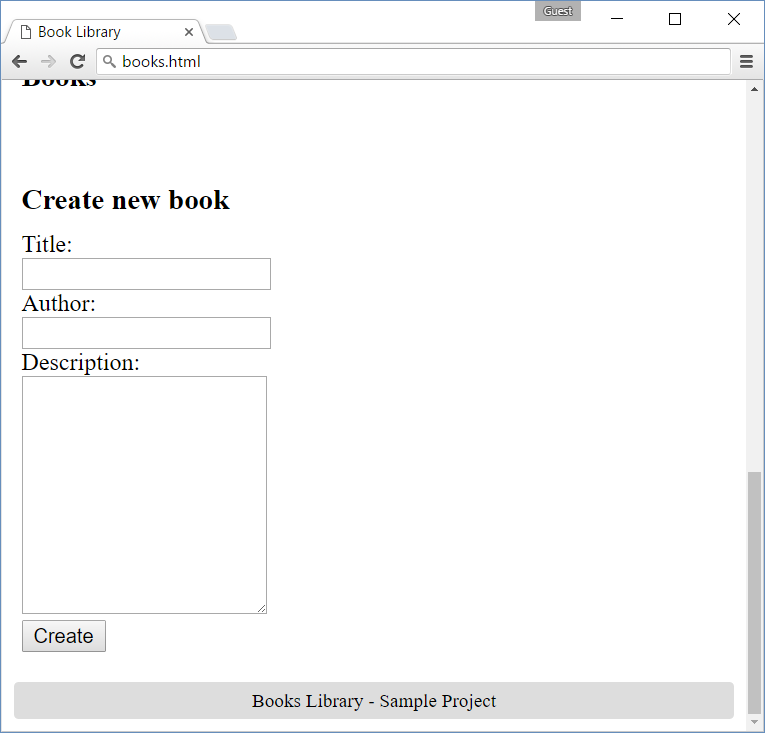
* Finally, style the page **footer**:



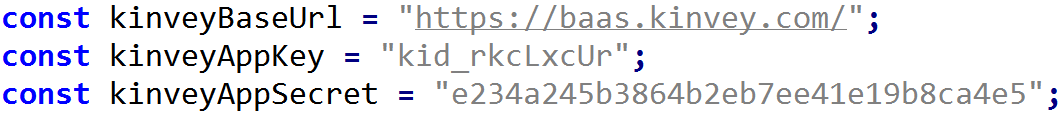
* Now **test again** the HTML page with the CSS styles. It should look better:



No, it looks broken. What’s wrong? Look inside the CSS. Some elements are intentionally hidden from the CSS (display: none). They will be shown later by the JS code. To test the HTML with the CSS styles, comment all occurrences of “display: none” in the CSS. **Now test again**. The page should look better:

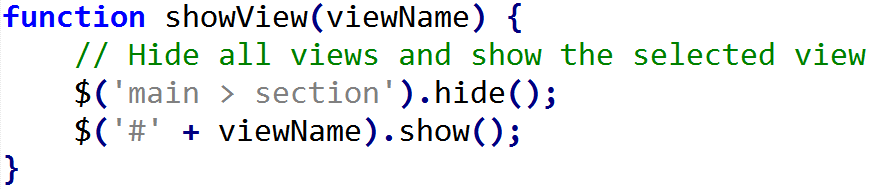
 

* It’s time to **write some JavaScript**. All these views should be driven by the JS code. Start with some settings. Define the Kinvey service URL and authentication settings as JS constants:



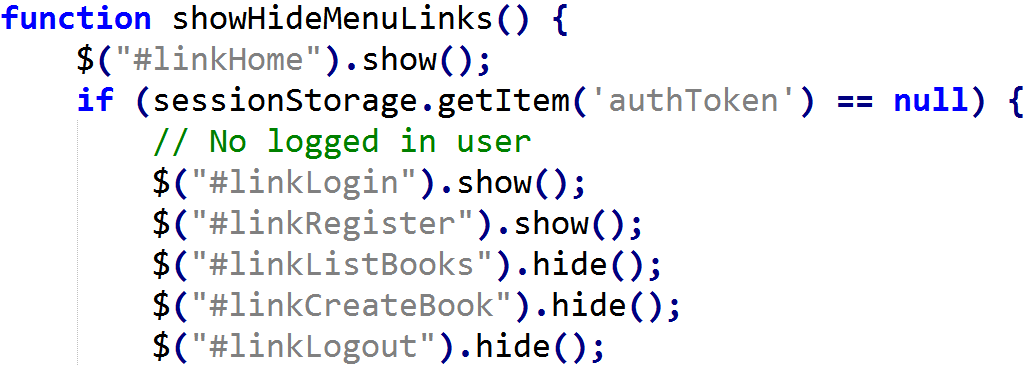
Please **change these credentials with yours**! Just login in your Kinvey app and copy / paste the **app key** and the **app secret** from the app dashboard in the Kinvey Management Console.

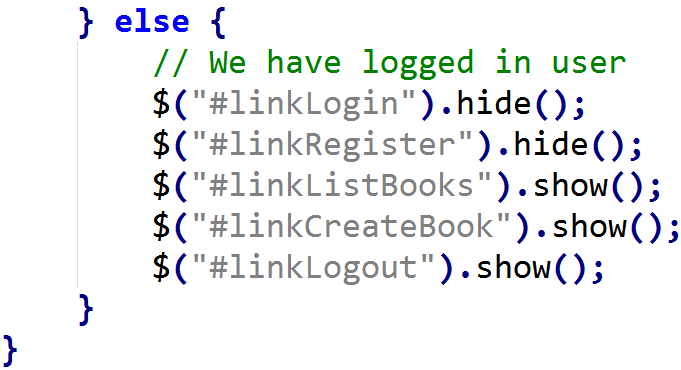
* Create a function to **change the current view**. It hides all views except the specified view. The idea is how have only one view visible in the page:



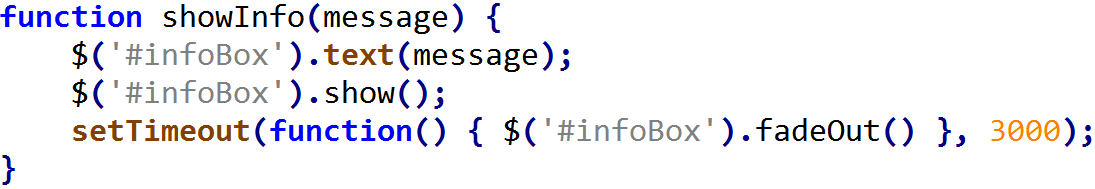
* Create also a function to **hide all unneeded navigation links**. When the user is logged in, hide the “Login” and “Register” links and show “List Books”, “Create Book” and “Logout” links. When there is a logged in user, do the opposite: show “Login” and “Register” and hide “List Books”, “Create Book” and “Logout”.

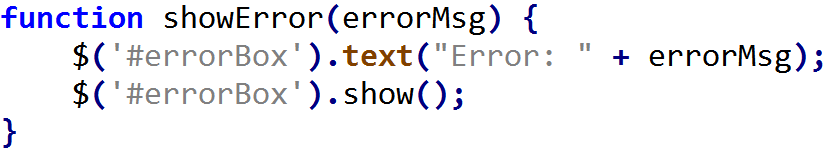
Assume we keep the authentication token for the currently logged-in user in the **session storage**:





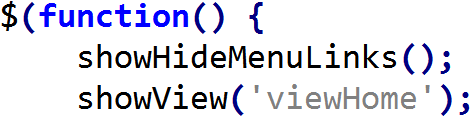
* Define functions to **show info and error messages**. These functions will be needed to notify the user in case of success or error after AJAX call:



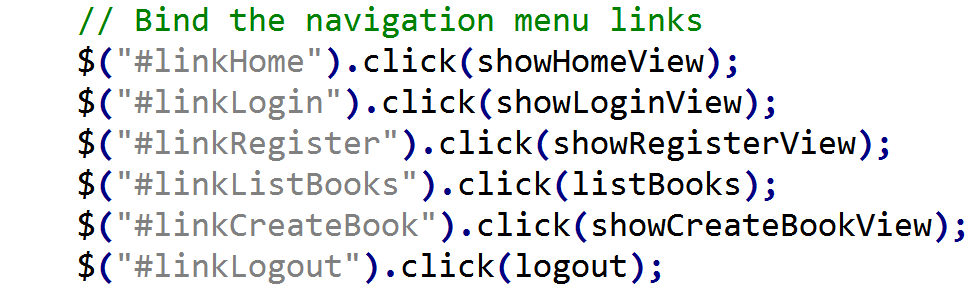


The info messages hide automatically after 3 seconds. The error messages hide only manually, on click.

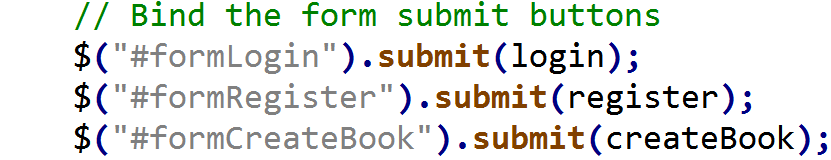
* At **application start**, show the home view and hide all other views. This will be done by handling the jQuery “on document ready” callback $(function { … }), which is called just **after the HTML is loaded** in the DOM tree and is ready to attach event handlers:



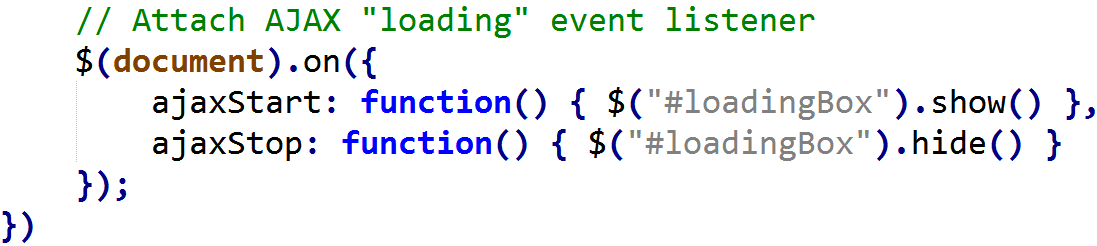
Also **bind the navigation links** to JS functions that will be defined later:



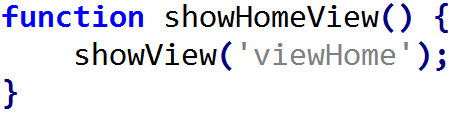
Also bind the **form submit actions** for all forms to JS functions that will be defined later:



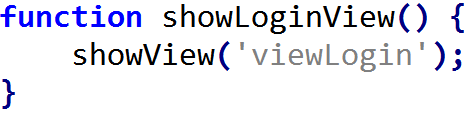
Bind the global **AJAX loading events** to display “Loading …” message when the service calls are in progress:



* Write the **action** that shows the **home view**:

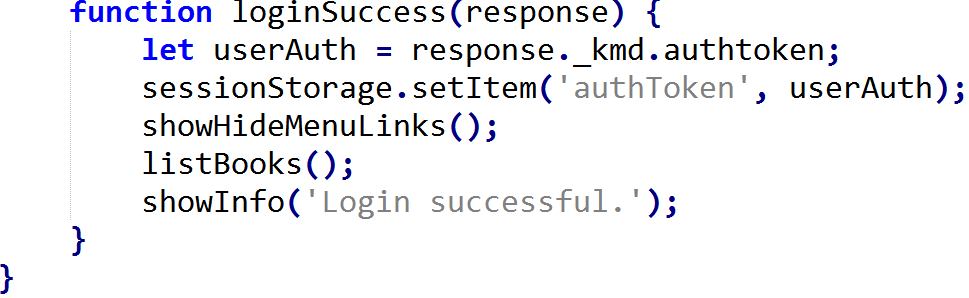


* Write the **action** that shows the **login view**:



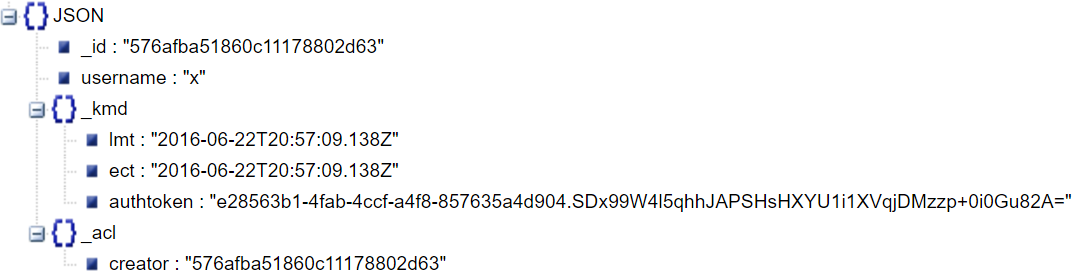
* Now it’s time to make the first **AJAX call**. It will be the **“Login” action**. When the user fills the login form and submits it, the login() function is called. It tries to login in Kinvey. See its code:





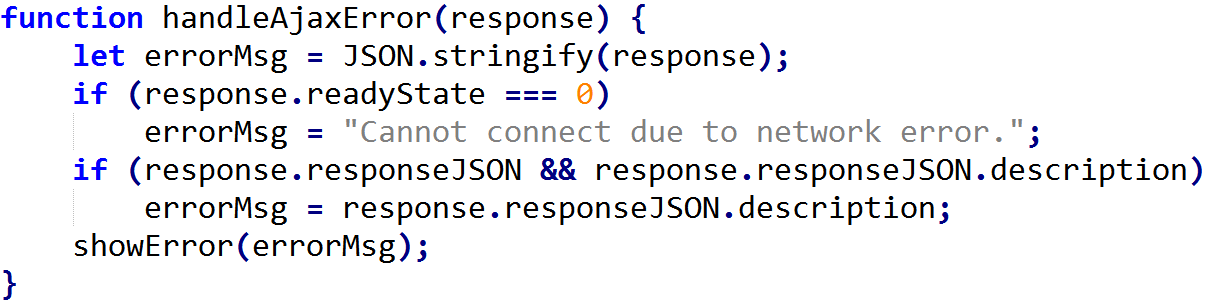
**How it works?**

The above code makes an **AJAX** **POST** request to [https://baas.kinvey.com/user/{app\_id}/login](https://baas.kinvey.com/user/%7bapp_id%7d/login). It posts the **username** and **password** from the login form in order to login in Kinvey. The service call is authenticated with the Kinvey app credentials (app\_id + app\_secret). The Kinvey back-end is expected to return a **JSON object** holding an **authentication ticket** (session auth token) like this:



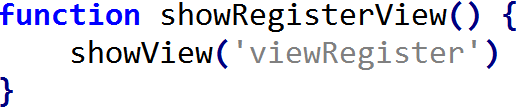
If the AJAX request is **successful**, the loginSuccess() function is called (asynchronously). It saves the **Kinvey authentication token** returned from the REST service (\_kmd.authToken) in the browser **session storage**, then redirects to the “**list books**” action, which should load and display the books from the library.

In case of error, the handleAjaxError function is called to show and error message:



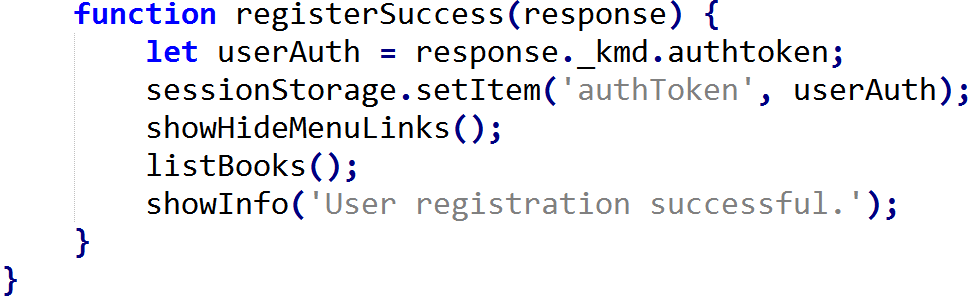
It checks for some **standard errors** (like “network error”) and tries to extract the **message text** from the AJAX response JSON object (if it is available). If message text is available it is shown, otherwise, the entire AJAX call response is shown as JSON text. This function will be used to **handle the errors for all AJAX requests** in the application, not just in the login.

* The “**show** **registration view**” action is very similar:

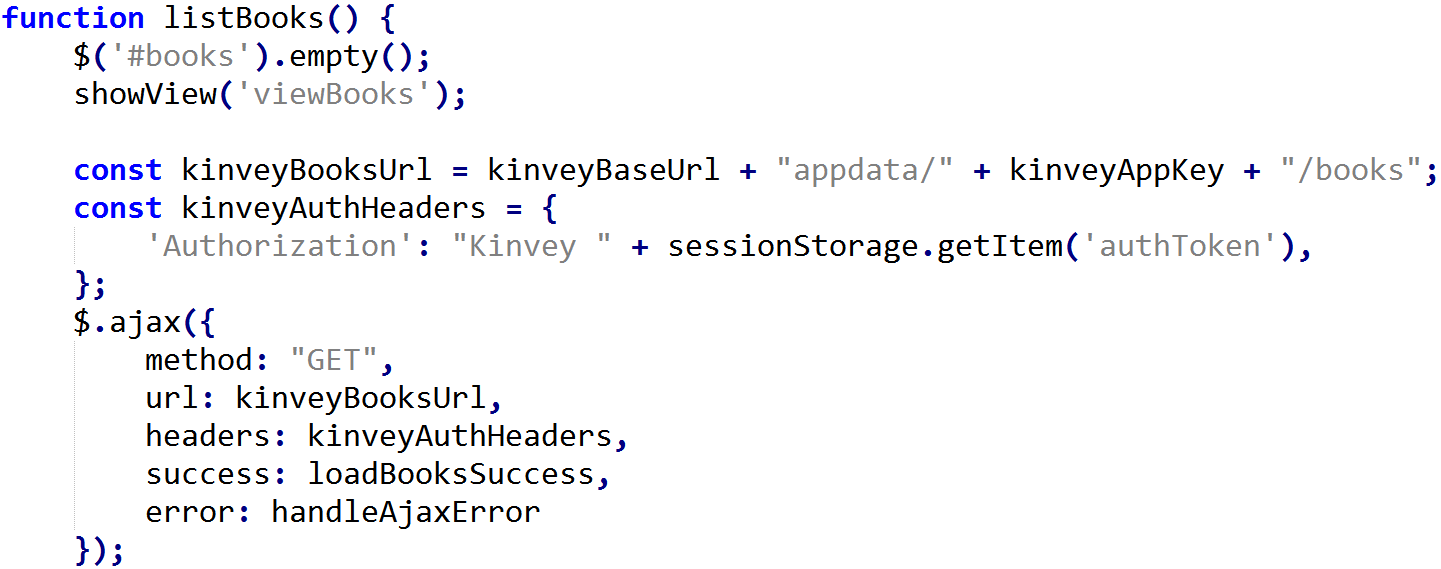


* The register() action is very similar to the login() action:

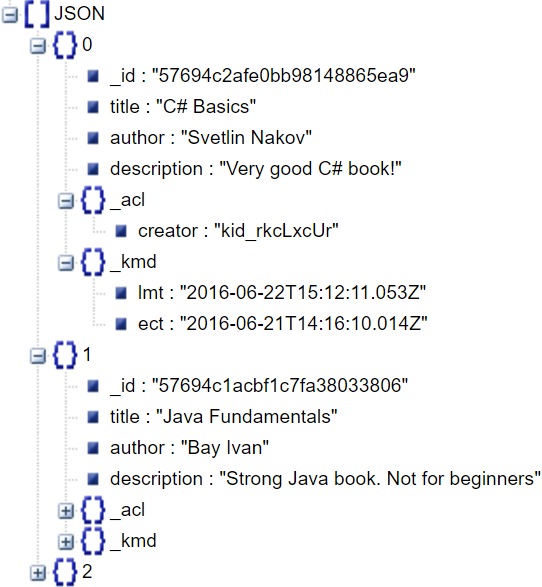




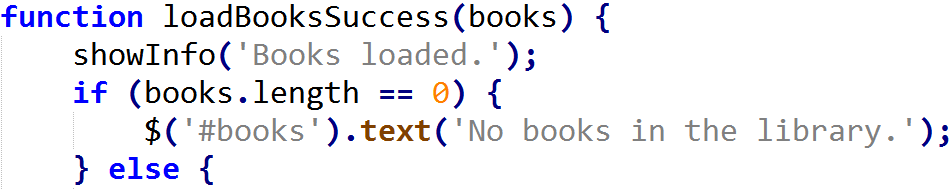
* Now, let’s implement the “**list books**” action to load and display the books from the Kinvey back-end:



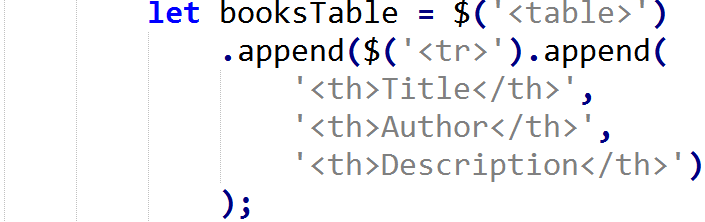
The above code sends **AJAX GET** request to [https://baas.kinvey.com/appdata/{app\_id}/books](https://baas.kinvey.com/appdata/%7bapp_id%7d/books) and processes the returned JSON object, which should look like this:



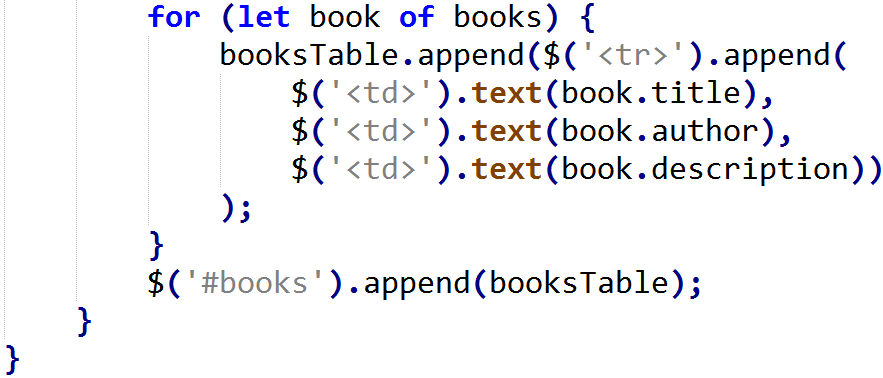
If the AJAX call is completed successfully, the returned **JSON array of books** is processed and displayed as a **HTML table** in the “books” <div> by the following JS function:



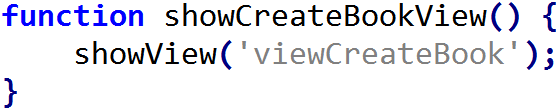
It first displays a **success notification message**, then creates checks for **empty list of books** and creates a **HTML table** and its heading row:



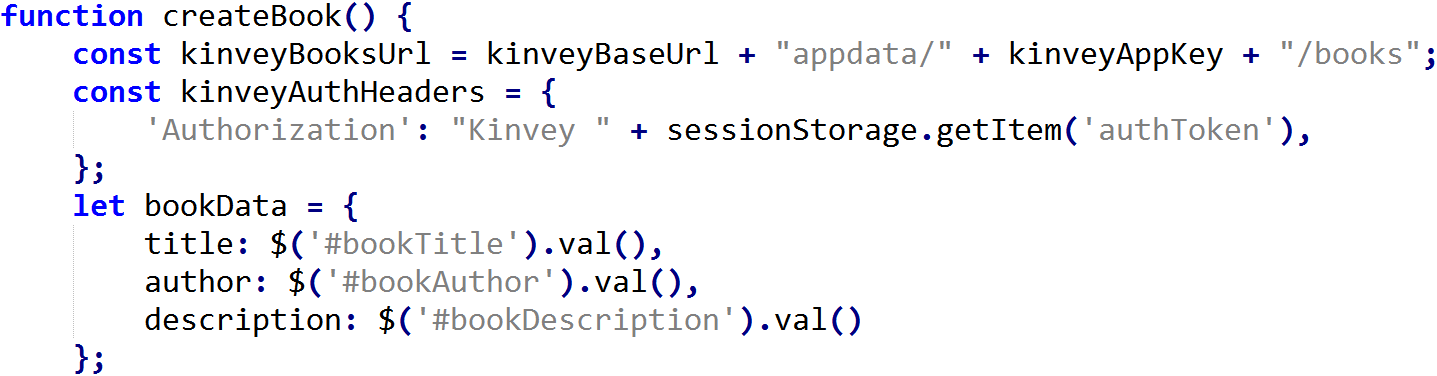
Finally, it appends **a table row for each book object** returned form the Kinvey back-end and appends the generated books table to the DOM tree in the “books” <div>:

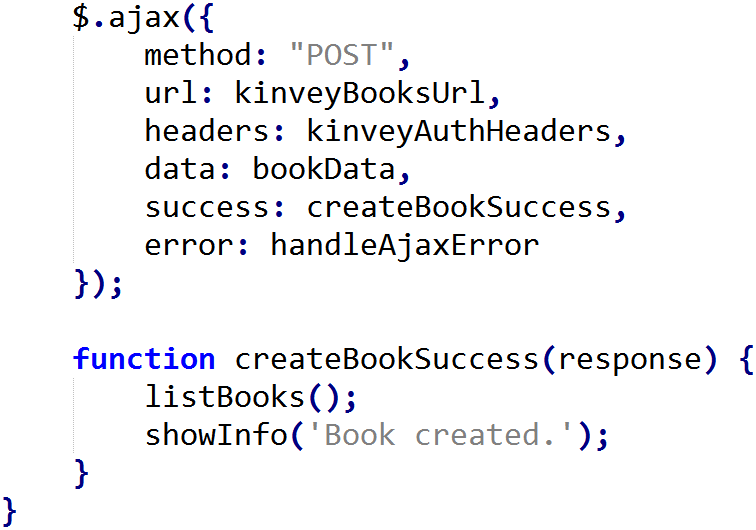


* The next view to implement is the **“create book” view**. It should show the “create new book” form when the “**Create Book**” link is clicked from the navigation menu:



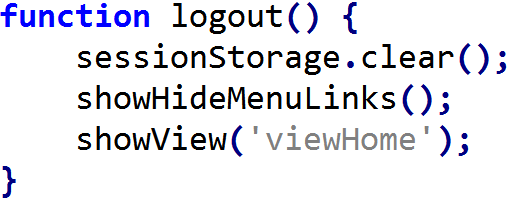
* Now, implement the “**create book**” JS function which is called when the “create new book” form is submitted:



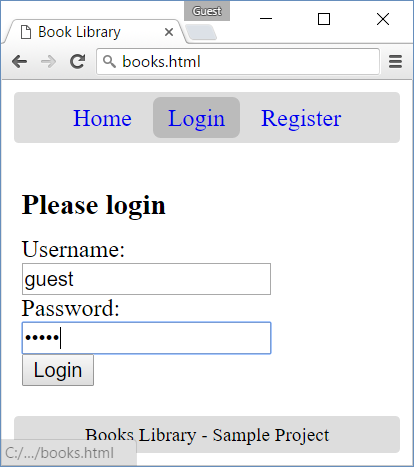
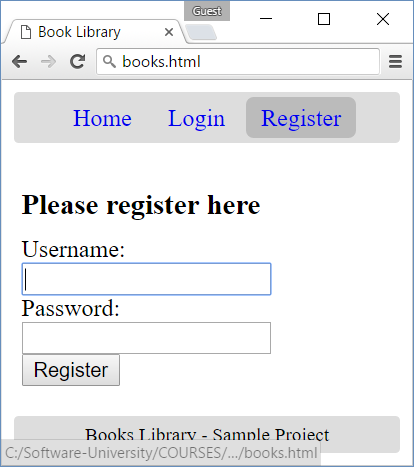


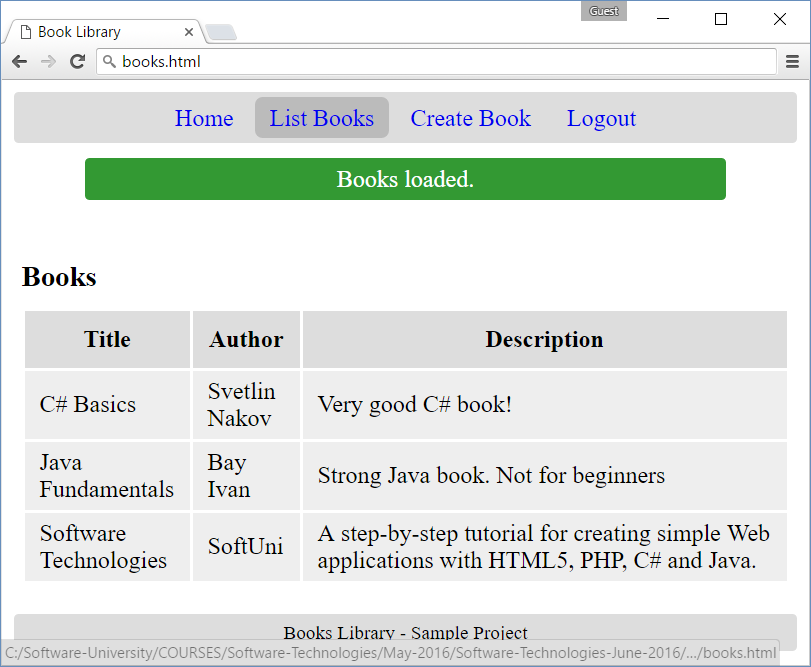
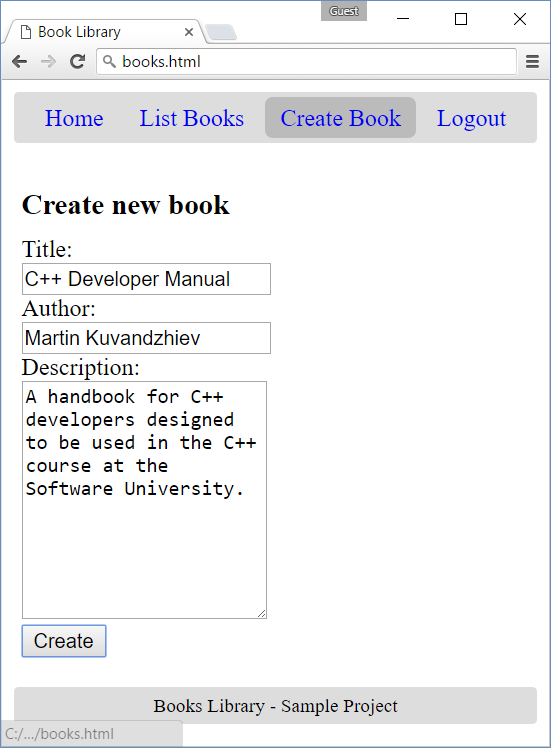
The above function makes an **AJAX POST** to [https://baas.kinvey.com/appdata/{app\_id}/books](https://baas.kinvey.com/appdata/%7bapp_id%7d/books) and sends the book title, author and description to it. As a result, the Kinvey back-end REST service will return the created JSON object. When the AJAX call is **completed successfully**, the application is redirected to the “**list books**” view. In case of error, it is shown by the handleAjaxError function, like in the all other AJAX calls.

* Finally, implement the “**logout**” functionality:



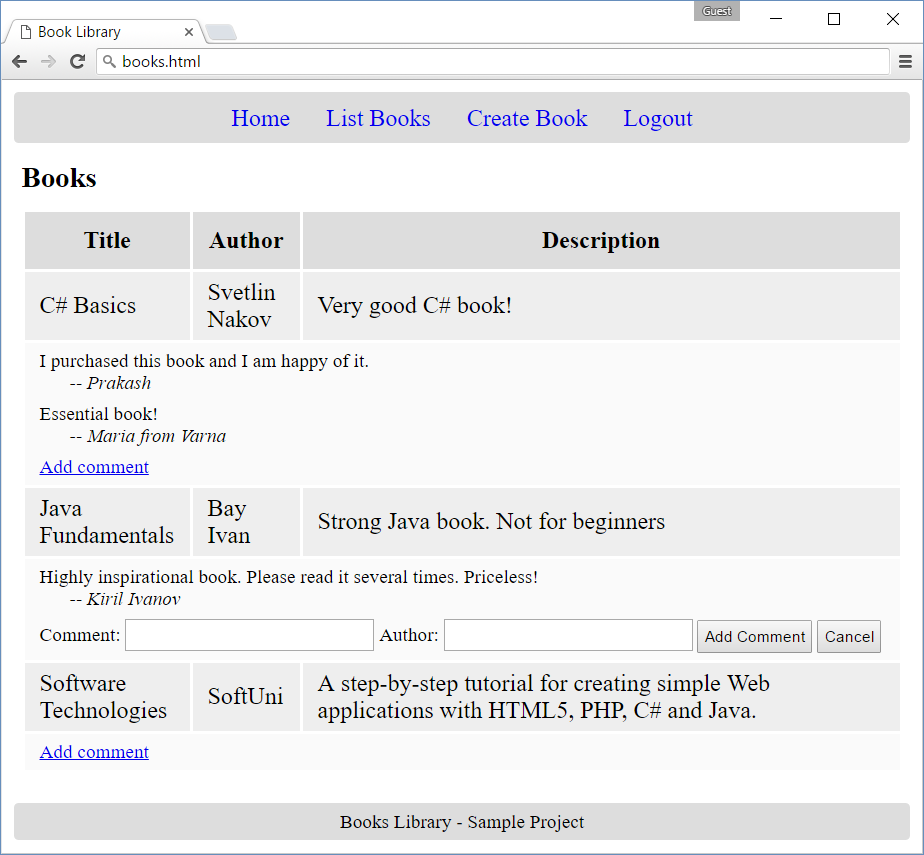
* Now the entire application is ready. **Test it** to ensure everything works correctly:

## Book Comments

Add functionality to **add comments** for each book. You may add **additional table row** to display the comments:

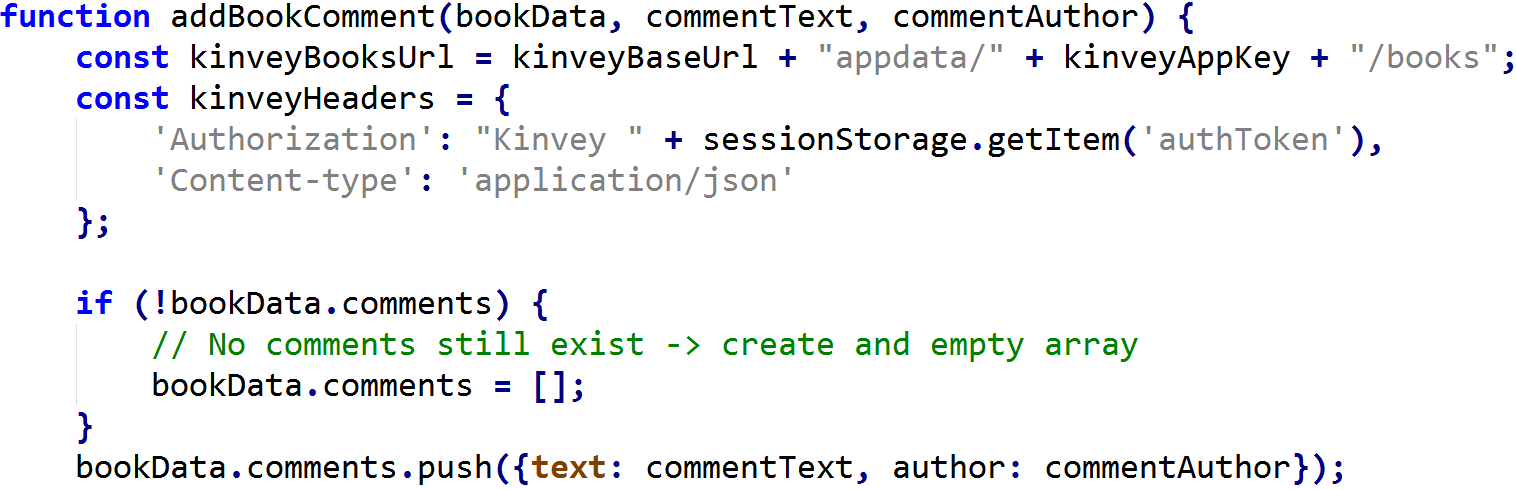


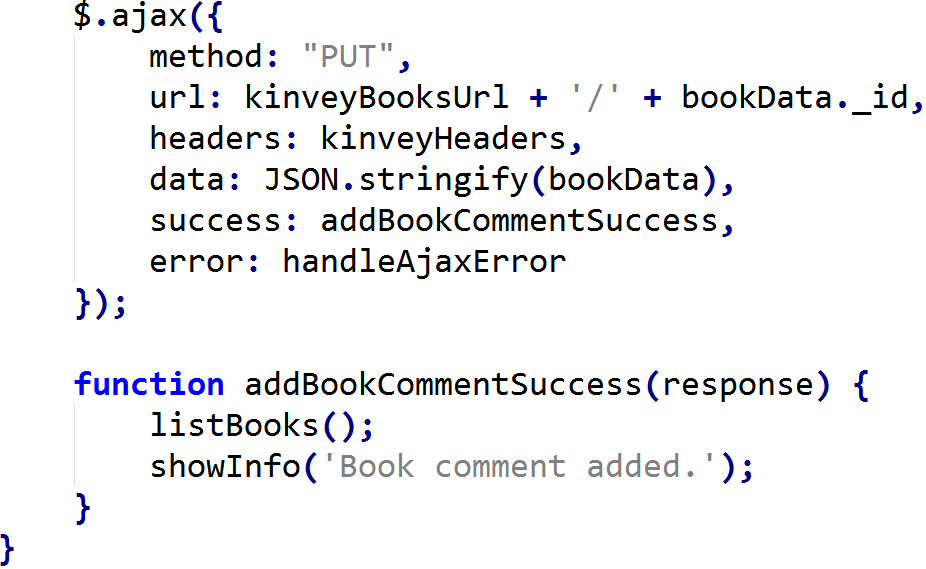
### Hint

* Keep the **comments** for each book inside the **book object** in the Kinvey back-end as **array of objects**, each holding **text** and **author**. The modified book object could look like this:



* Render the **comments for each question** (if any) in the loadBooksSuccess(books) function.
  + Check for comments for each question returned from the AJAX call and if comments are more than 0, display them in a new table row <tr> in a single table cell <td colspan="3">…</td>.
  + For each comment use two <div> elements (for the **title** and for the **author**).
  + Add another <div> holding a hyperlink **[Add comment]**, which shows the “add comment” form.
  + Add a **hidden “add comment” form**, holding **title** and **author** text fields + **submit** and **cancel** buttons.
  + You may use “**required**” attribute to enforce non-empty comments.
  + Bind the **“submit” event** to a JavaScript function that makes AJAX call to create a new comment.
  + Keep the original **question object as JSON string** in an attribute “data-question” in the table row holding the comments. When a new comment needs to be added, you can add it to the question object and update the Kinvey back-end with AJAX call.
* To **add a new comment** for certain question in the Kinvey back-end, use **AJAX PUT** request to [https://baas.kinvey.com/appdata/{app\_id}/books/{book\_id}](https://baas.kinvey.com/appdata/%7bapp_id%7d/books/%7bbook_id%7d) to **update the entire book object** holding the new comment. Kinvey does not support partial updates (**HTTP PATCH**), so the entire book object should be updated. You can use the following JS code to **add comments** to existing book in Kinvey:





* Note that the above described approach (to hold comments as array inside the book object) is **simple**, but **not recommended**, because it has some significant **drawbacks**:
  + Users will be able to **add comments to their own books only**. You could easily test this scenario. This behavior can be changed from the Kinvey Management Console 🡪 Data 🡪 books 🡪 Settings 🡪 Permissions 🡪 Public.
  + If two users **add comments in the concurrently**, the second user could overwrite the first user’s comments. You could easily test this scenario.
* A better approach would be to have two collections “books” and “comments” and to reference the “book” entity from “comments” entity, as it is described in the Kinvey documentation: <http://devcenter.kinvey.com/rest/guides/datastore#RelationalData>.