# Exercises: Creating a Blog with HTML5, JavaScript and Kinvey

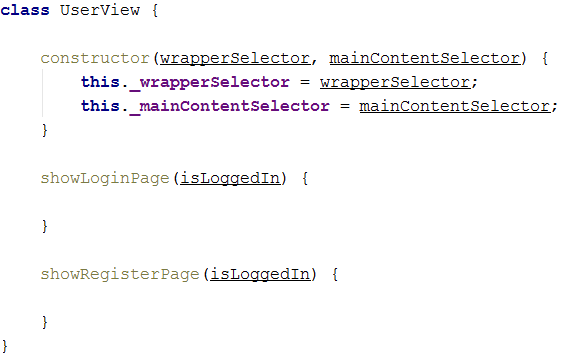
Problems for exercises and homework for the [“Software Technologies” course @ SoftUni](https://softuni.bg/courses/software-technologies).

If you follow the steps of this exercise, correctly, you will implement a simple blog SPA (Single-page Application) which can be of great use to you in the future.

Now that we have home view and controller, our home pages are basically ready. Now we can start implementing the user logic.

## Creating the User View class

The user logic for this particular application is very simplistic. We have login and register for users. That is the only logic directly connected to the users. The view class renders pages, as we’ve learnt so far. We have 2 pages – login and register.



Same as the other classes, we create a constructor, and the necessary functions. It is good for the consistency, the classes of particular groups to follow a particular pattern. For example every view’s constructor to take exactly 2 elements and those elements should be – wrapperSelector and mainContentSelector.

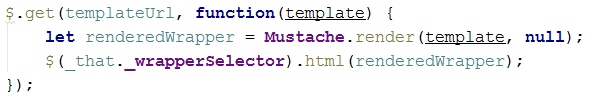
Now let’s see the functions, they are a bit more interesting. They don’t have data passes to them, that is because the templates for the forms do not render any objects, they just render HTML code. However they receive as a parameter a “**isLoggedIn**” Boolean parameter. This parameter indicates if the user is logged in. This is done because the view should know which sidebar to render along with the form – the guest or the user’s.

## Getting the login page

As every other page, before rendering, we need to get the template.



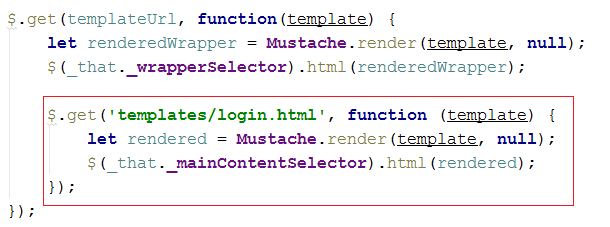
However, here we have a condition. If there is a logged in user, we render the logged in form, with the logged in sidebar, if there isn’t, we render the guest sidebar. The templateURL is the link to the template that we will get.



Same as the home pages, we first get the wrapper, we render it with null data and pass it to the HTML of the element corresponding to the wrapper selector.

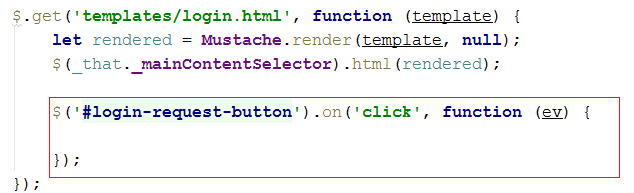
## Rendering the login form

Now that we have the wrapper, we should render the login form too. The form depends on the wrapper… We cannot render a form without a wrapper. That is why we nest the get request for the login form in the main get request.

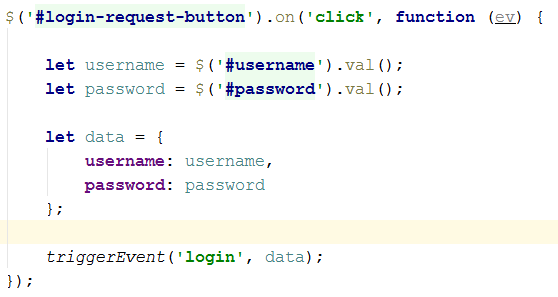


## Creating an event handler

Forms take as input data, and submit the input data to somewhere. Submitting data is done by clicking a button or something. Our current template has a button, but it has no event handler on click. So let’s make one.   
Why is the event handler created when the template is loaded? And why is it done in the view? When a template is loaded it is nothing but HTML code, so we must add handlers such as JavaScript functions and handlers, every time we load the template.



This is an event handling function which will be invoked when the event “**click**” is triggered, on the button corresponding to the selector, specified in the JQuery function.



First we need to extract the data from the 2 input fields in the login form. We do that by taking them using their id’s as selectors and taking their value. Then we build an object **data** which holds username and password. And then we use the **triggerEvent** function which is provided to us by the framework. It triggers an event by given name and passes to the handler an object – in our case, the data.

## Creating the Register form

The register form is almost the same as the login form. The only differences are that the template is “**register.html**” and the data you need to extract is a bit more. Follow the steps above to implement the register form. Apart from username and password, you need to extract two more fields, the **confirm-password** field and the **full name** field.

Fill the data object with them and trigger an event, and pass the object with the event… But this time the event will not be “**login**”, but “**register**”.

When you create the **showRegisterPage** function you will be ready with the UserView.

## Creating the User Controller Class

Now that we have a view, let’s make a controller.



As we said, it is good for particular groups of class to follow a particular pattern. The controllers follow one too. Every controller has a view, a requester, a base URL and an app key.

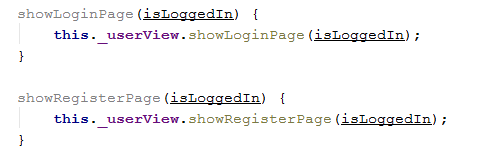
We have several functions. The first two are familiar – they work on the same concept as the home controller.

There are however different ones - the **login**, **register** and **logout** functions. Now here it becomes a bit complex.

1. The controller asks the view to render a page
2. The view renders the page and applies an event trigger to it
3. When the event is triggered the specified operation corresponding to the event, is called in the controller
4. That operation makes the request and finishes the process

## Implementing the showLoginPage and showRegisterPage

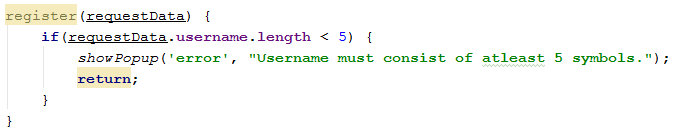
Let’s implement the **show-page** functions because they are more familiar.



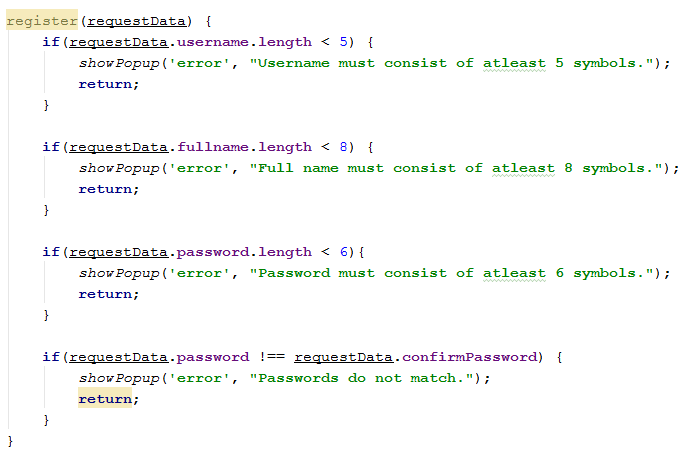
As we see the functions are nearly the same as in the Home Controller.

## Implementing the register core logic

Now this functionality will be a bit more interesting. We will have data validations. Let’s make a data validation.



The requestData contains the data passed to use by the view, when it triggered the event, it has username, password, full name and confirm password. Now if the username is less than 5 symbols long, we print an error popup notification, to notify the client that his request data is invalid, and we exit the function with the return statement. Let’s do some more validations.



We also check If the given passwords match.

Now that we have validated data, we can send it to Kinvey with a **POST** request and register the user in the database. However there is one thing we must do before that. Kinvey will automatically hide the password field from the naked eye, but it will not hide any additional fields the client has imported into the request data. If we post in the database the confirm-password field, it will be visible to anyone with access to the database, which is not good. That is why we will delete that property. This is done very simply, by using the **delete** statement.



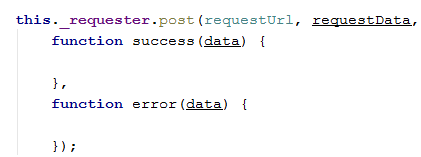
Now that we have clear data let’s make the request itself.

This time we are making a **POST** request. Every REST request, aside from the **GET** request, has request body, with data. That is why we are giving one more parameter to the request.

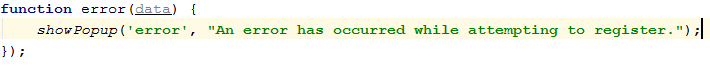
We initialize the request URL



And then the request:



Let’s implement the error callback first:



Then the success callback function:



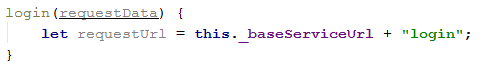
As usual we notify the client that his request has been successful.

After we’ve registered, we redirect the client to the login page route. That is done by using the **redirectUrl** function, provided by the framework.

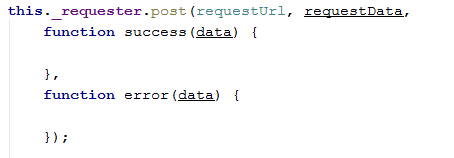
With this we are ready with the register functionality. We have posted in the Kinvey database the user with his name, password and full name.

## Implementing the login core logic

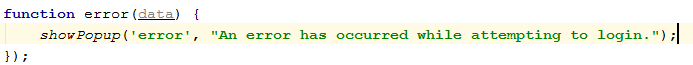
First let’s initialize the request URL.



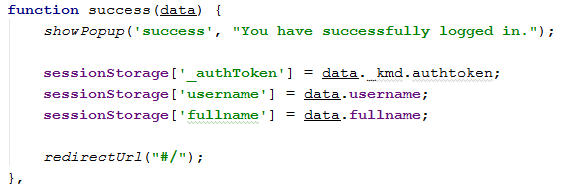
Then let’s make the request itself.



As usual let’s implement the error callback function first.



Then let’s implement the success callback function



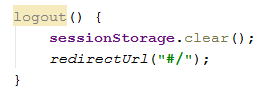
First we show up a success popup notification to notify the client that his request was successfully completed. The data object contains several fields, mainly corresponding to the information we have stored in Kinvey. Basically what we need from that is the authorization token, the username, and the full name. When a client is logging in, that is called user session. To save a user session, you use the sessionStorage. Store the data exactly as it is described above.

After we’ve logged in and saved the current user session, we redirect the client to the home page route. **“#/”** is the home route in an application.

With this we have completed the **login** functionality.

## Implementing the logout logic

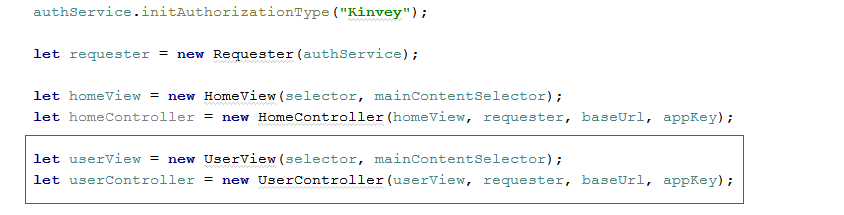
The logout logic is very simple. Because the user session is basically what identifies that a person has logged in, and is completely stored in the session storage, clearing the session storage would basically mean that there is no longer a logger in person. So…



After we’ve logout we redirect the client to the home route.

## Revision

If you’ve followed the steps above exactly as specified, you should have completely functional user-based logic. Test it by instantiating a user view and a user controller in the app.js. Then use the **showLoginPage** and **showRegisterPage** functions.



Then try the two pages. If you’ve done everything correctly, you should see the following.



