# 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.

## Before Starting

The goal of this lab is to create a **Single Page Application (SPA)** with JavaScript and HTML5 – **Simple Blog System**. The blog should provide the following functionality: **list posts**, **register**, **login**, **logout**, **create new post**.

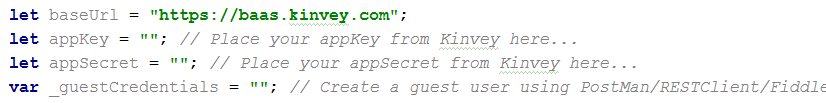
TODO: Add Screenshots

You are given a **project skeleton** (project template) holding a mini **framework** to be used as foundation of your project. You will use the **project structure**, Web design, project architecture and ready-to-use classes to build the Blog. Your goals are to implement the requested functionality using the project framework.

## Initialize the Main Resources

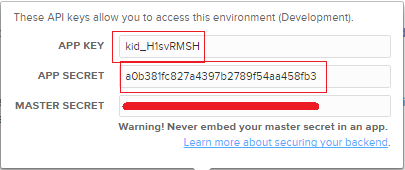
The **skeleton of the project** provides you with some already implemented functionality. However, the skeleton holds placeholders, which you must fill in order to make the application work.

Let’s first check the TODOs the developers left us. Open up app.js – the main file that runs the application.

****

## Fill the Placeholders

Extract your **app key** and **app secret** from your Kinvey application and fill them in app.js:

****

**Tip:** never give your master secret, to anyone!



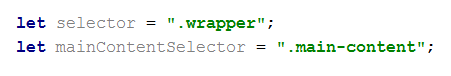
## Guest Credentials

Now we will initialize the **guest credentials**. The guest credentials are needed because Kinvey does not allow data retrieval before the request has even achieved success state.

The guest credentials are basically the credentials of a dummy user, used mainly because Kinvey does not allow data retrieval without login state. We pre-initialize them so we have them ready in order to take the posts you need to render on the home page.

## Initializing the Selectors

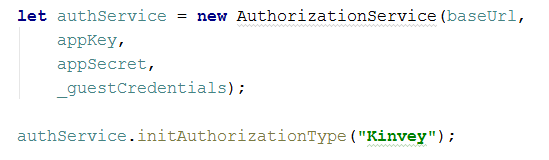
After storing the app key, secret and user credentials, we are left with only the selectors as preparatory actions.



## Creating the Requester and Authorization Service



The application requires a Requester, to handle the HTTP requests, and an Authorization Service, to handle the authorization and generating of request headers. The framework we received provides implementation for both classes. Use them. Let’s first create the authorization service:

****

As we see, first we instantiate an object from the class AuthorizationService, and we pass the needed arguments to the constructor. After that we are calling the initAuthorizationType function, why? Every API has some authorization type in its session authorization header. The Kinvey API uses “Kinvey”, that is why this line of code is necessary, to specify exactly what API we are working with.

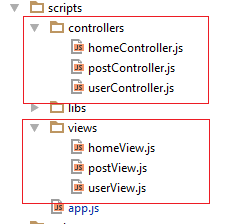
Next comes the Requester, its constructor accepts an AuthorizationService as a parameter, because the requester needs one, in order to generate its request’s headers.



Now that we have some of the base things needed for the functionality of the application, we can start implementing the core elements – the controllers and views.

## Creating the Controllers and Views

As the comment in the app.js file specifies, we need several controllers, and several views. The controllers depend on the views, so the views must be initialized first. Let us create the needed files, for now.



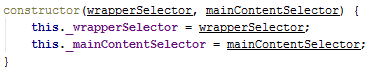
## First lets create the Home View class

Open homeView.js and create a class HomeView.



## Create constructor for the Home View class

Every class has a special method / function, called **constructor**. It is used to initialize the member data for the class in order to make it functional. Let’s create one for the Home View class.



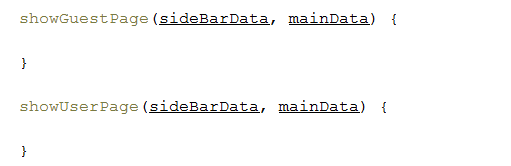
The constructor takes 2 parameters, because the class has 2 members. A wrapper selector and a main content selector, which correspond to the selectors we initialized a while before in the app.js. The “**this”** keyword is used as a reference to the class.”**this.\_wrapperSelector**”, is a declaration of a class member. Both class members are being assigned values which were passed as parameters to the constructor.

## Home View functionality

Now that we have a class with a constructor, we can make an instance of it. But what good could that give us – an empty class. Let us implement the main functionality of the Home View class.

The Home View class is used to render the home pages. We have basically 2 home pages – the Guest home page and the User home page. The Guest home page is rendered when no user is logged in, and the User home page when there is a logged in user. Logged in users have different rights, and functionality, which is why the pages are different and visualize different things.

Let’s create two functions – “**showGuestPage**” and “**showUserPage**“.

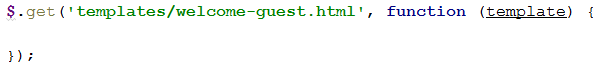


Pages are rendered trough templates of HTML code. Those templates are being filled with specific data, which is passed to the functions that render them. Mustache.js is used to render the template with the given data, but we’ll reach that point too.

## Getting the Guest home page template

Let’s render the Guest home page.

First we need to get the template, which is done, using JQuery, like this



We are using the template “**welcome-guest.html**” which was given to us in the skeleton. The **get** function requests a file on the local host, and passes it as a parameter to the function given as callback to that request.

## Rendering the page content

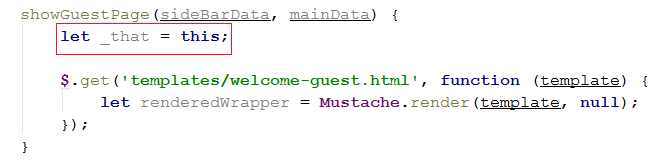
Now we have a template and data, let’s render the content for the home page using Mustache.



Now, why do we pass to Mustache null as data? In the current page we render several things – the sidebar, the recent posts links, on the sidebar and the main content (the posts which are presented on the home page). The sidebar basically needs **no** data, because the recent posts are loaded trough a separate **get** request. That is why we just have to pass to the wrapper selector the currently rendered wrapper.

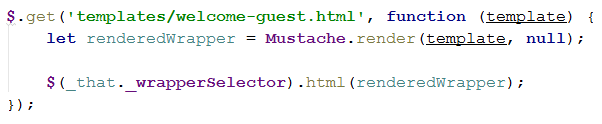
Now if we want to take the wrapper selector of the current class, we cannot just say “**this.\_wrapperSelector**”. In JavaScript everything is an object, including the function, and we are currently operating in the **get** function handler. If we call **this** it will create a reference to the function object, and the function has no wrapperSelector member.

That is why we will do the following



By doing this, we’ve just created a reference to the main class, out of the function handler. That way we will be able to use it inside the function.

Now that we have that, let’s render the rendered wrapper on the web browser like this

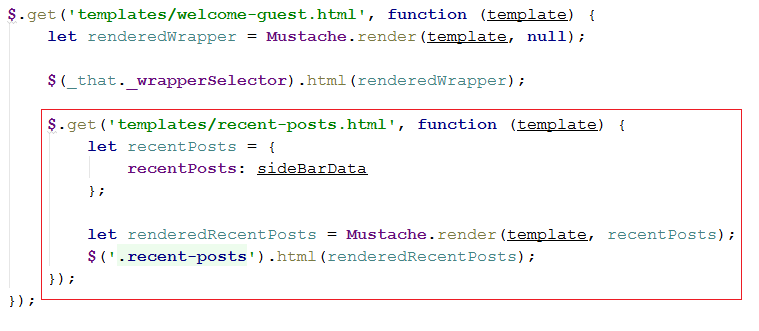


Using the reference word that we created externally – “**\_that**” we call the \_wrapperSelector member and we use it as a reference to the HTML element. Then using JQuery, we are replacing that particular element’s HTML with the currently rendered content.

## Rendering the recent posts in the sidebar

In JavaScript requests are asynchronous, that is why when 1 request is depending on the other, you must nest it inside the other, because that way you are assuring yourself that the requests will be completed in the order you gave them. This is done because, when we have multiple asynchronous requests, one might be completed before the other, despite being called after it.

Now, previously we rendered the sidebar, this time let’s render the sidebar’s recent posts. The recent posts depend on the sidebar, that is why we nest it’s rendering in the sidebar’s.

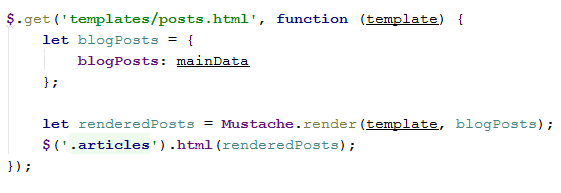


As we see it is the same, simple **get** request, except this time we get the “**recent-post.html**” template. Then we create an object **recentPosts**, and in it we create a **recentPosts** member which is being initialized with the value of the **sideBarData** parameter passed to the **showGuestPage** function. Here Mustache actually uses some data to render the current template. That is why we pass to it the recentPosts object we created. Mustache needs object with the same name as it was referenced in the template. You can go check the template to see how it was reference there.

After that, instead of using any of the selectors of the class, we use a one-time only selector, that is why it isn’t in a variable, because it is used one time only. We render the rendered recent posts to the current selector’s HTML.

## Rendering the main content

Now that we are finished with the sidebar, we should render the main content of the home page, which are the blog posts. That is done by doing the same thing as the recent posts, however this time we are passing to it the **mainData** parameter of the **showGuestPage** function.



This is also done in the **get** request of the sidebar, right after the **recent-posts** rendering function. As you see this time we initialize the object as **blogPosts**, because the template uses that word to create a reference for Mustache. After we’ve rendered the posts, we pass them to the **articles** element, again using the “\_that” reference.

## Rendering the User Home Page

Following the steps above, do the same for the User home page. The only difference between the two functions is that the user home page gets initially the “**welcome-user.html**” template instead of “**welcome-guest.html**”.

## Revision

Initialize a **homeView** variable in the **app.js** and instantiate it as an object of the **HomeView** class. Then call the **showGuestPage** function just to check how it looks. After that we’ll delete it. Like this:



If you’ve done everything correctly you should see the following

