# JavaScript Applications Lab: Social Network

This document defines lab exercises from the ["JavaScript Applications" Course @ Software University](https://softuni.bg/courses/javascript-applications). The goal of this lab is to practice the development of SPA (Single Page Application) from scratch, using libraries such as jQuery, sammyJS, Mustache, Q, Noty, etc.

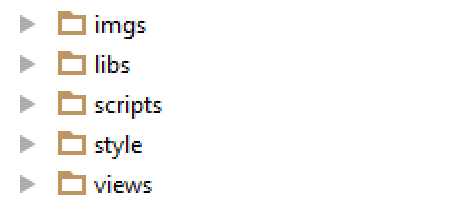
# Application Overview

You are given an **application description** and **screens of all possible views** attached to the archive. Read the description and study the requirements before proceeding.



# Project Structure

Let's create some folders:

* **imgs** - holds all image resources
* **libs** - holds all external libraries the application will be using
* **scripts** - contains all scripts written by you
* **style** - HTML stylesheets
* **views** - partial HTML templates

# Install the Necessary Libraries

Use a package manager such as **npm** (or download manually)and set up all the following libraries in the libs folder:

* **jQuery -** used for DOM manipulations
* **sammyJS -** used to asynchronously load HTML pages
* **Q -** used to defer asynchronous function calls and process them subsequently
* **Mustache -** templating enginefor injecting data into partial HTML views
* **Poppy -** remember that JS library you wrote back during the **Advanced** **JS** course? ☺ Well you're gonna used it here to display various popups for notifying the user that something happened.

*(In case you haven't written it, you can download it from here -* [*https://softuni.bg/downloads/svn/javascript-oop/March-2015/Advanced-JS-Sample-Exam-Poppy.zip*](https://softuni.bg/downloads/svn/javascript-oop/March-2015/Advanced-JS-Sample-Exam-Poppy.zip)*)*

# Application Services

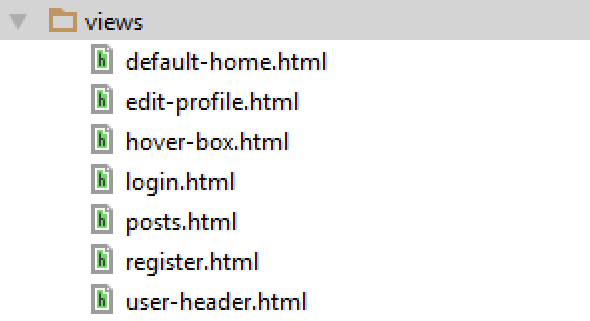
In order to build an app, we need some sorts of services to access data - without data, there is no application.

Go to kinvey.com and create a new app - **Social Network**. Set up 2 collections- **Users** and **Posts** and add the necessary columns as described in the **problem description** given in the archive.

# Split Views

Now that we have a **database** (the 2 collections in kinvey.com) and the **services** kinvey provides to access the data, let's begin writing our single-page front-end app!

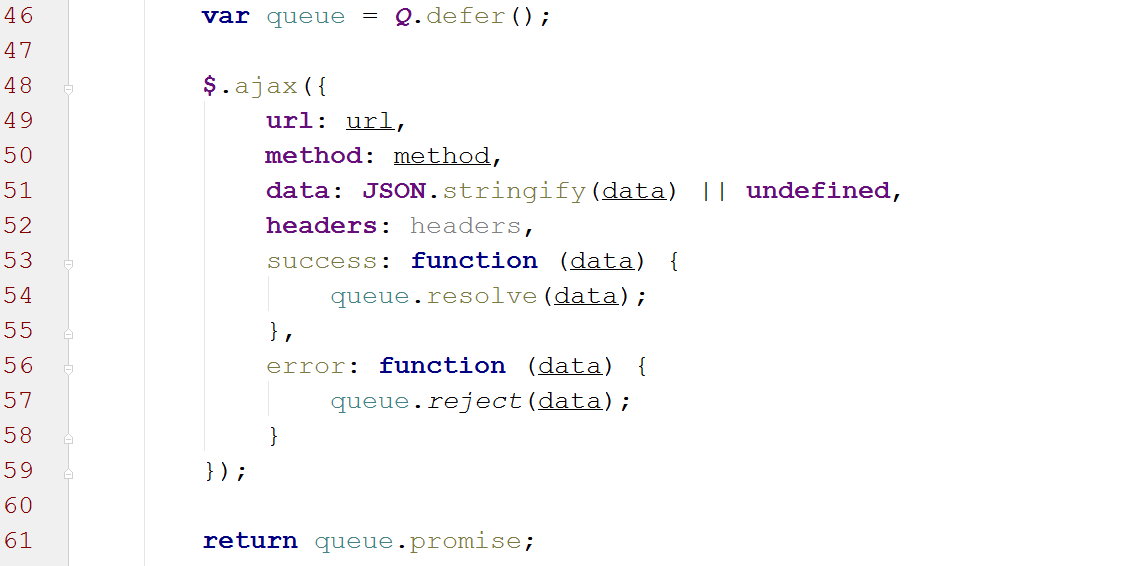
You are given a **307-line** long HTML file, holding all different views of our future application. Cut it down into several HTML partial views and put them into the **views** folder.



# AJAX Requester

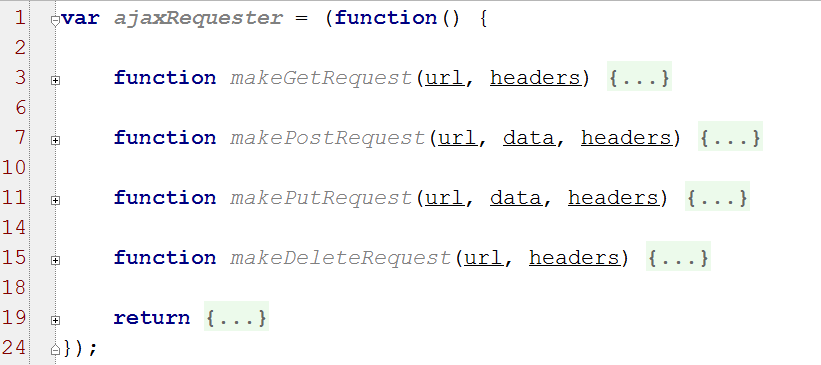
We have our HTML ready, but we still can't get any data from kinvey's services. Let's write an **AJAX Requester** for **requesting** and **retrieving** the data from our kinvey app in cloud.

1. Open the scripts folder and add a new script **ajax-requester.js**.
2. The requests we will be making are **GET**, **POST**, **PUT** and **DELETE**.
3. Making requests is done with jQuery's ajax function (**$.ajax({…})**). It receives as argument an object containing all parameters of the request.



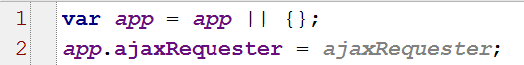
Each time we make a request we create a promise (queue) which will listen for the result of $.ajax.

Create requests for all four request types in 4 different functions and wrap them in an IIFE.



**Avoid duplicated code through reuse.**

1. Apply the **revealing module pattern** by revealing only what is necessary from the **ajaxRequester** module.
2. Create an **app** object to attach the ajax requester to it.

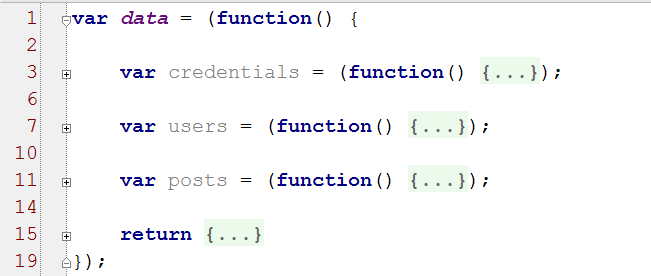


# Data

Now that we have a module for making standard AJAX requests, it's time we made a module for accessing the data for us.

1. Create a **data.js** script in the scripts folder.
2. Create a data module with **3 sub-modules**:

* **Credentials** - will hold information about the current user
* **Posts** - will offer functionality for getting/adding posts
* **Users** - will offer functionality for getting/registering/logging users



Let's start with **credentials** - they will hold info about the current user and allow him to authorize himself when he tries to access a given service. Create the following functions:

* **getHeaders(contentType, useSession)** - returns an **object** holding the following an **Authorization** key-value pair and **Content-Type** header. If **contentType** parameter is given to a **True** value–we add to the return object a **'Content-Type':'application/json'** header. If **useSession** parameter is given a **True** value – we add to the return object an **'Authorization'** header. The **authorization** header is **formed** by **converting** the **username** and **password** (or appid and appSecret) to **base64** string. To do that you need to firstly **append** the **username** and **password** (or the appid and appSecret) to each other **separated by colon ':'**, then you need to **convert** the whole string to **base64**. The authorization is **'Basic'** when using **app** **credentials** and **'Kinvey'** when using **session**.

Sample headers:

|  |
| --- |
| **'Content-Type'**: **'application/json'**, **'Authorization'**: **'Basic lkXy15TFdsMWdhMGc6NzQ3NjYxNGM2Y2Q5NGNkYWFlY'** |

The headers will be sent each time we make an AJAX request to kinvey.com.

* **getSessionToken()/setSessionToken()** - gets/sets a session token in the local storage/session storage (up to you to choose).
* **getUserId()/setUserId()** - gets/sets the currently logged user's Id in the local storage/session storage (up to you to choose).
* **getUsername/setUsername()** - gets/sets the currently logged user's username in the local storage/session storage (up to use to choose).
* **clearStorage()** - clears all local information about the user (used when he logs out).

In short, the **credentials** module:

1. Provides functions for **retrieving**/**setting** information locally for the **current user**
2. Stores the **current headers**

Users:

* **login()**
  1. Send a **POST** request to **https://baas.kinvey.com/user/:appid/login** with username and password key-value pairs in **JSON string format.** Use the AJAX requester's **POST** function we wrote earlier using **app credentials**.
  2. Save the data from the response to the local/session storage for future use (**session** **token**, **username** and **userId**).
  3. Return the response data.
* **register()**
  1. Send a **POST** request to **https://baas.kinvey.com/user/:appid** with **username**, **password**, **name**, **about** **info**, **gender** and **picture** in the request body. Use the AJAX requester's **POST** function using **app credentials**.
  2. Return the response data.
* **editProfile()**

1. Send a PUT request to **https://baas.kinvey.com/user/:appid/:userid** with the new data you wish to change in the request body. Use the AJAX requester's **PUT** function using **user credentials**.
2. Return the response data.

* **getById()**
  1. Send a GET request to **https://baas.kinvey.com/user/:appid/:userid** to receive data about a user with the specified id using **user credentials**.
  2. Return the response data.
* **getCurrentUserData()**
  1. Return the **data** for the **currently logged user** from local/session storage.
* **logout()**
  1. Send a POST request to **https://baas.kinvey.com/user/:appid/\_logout** using **session credentials**.
  2. Clear the local/session storage of any saved data about the currently logged user.

Now do the same for posts. Provide functions for **getting all posts**, **getting post by id** and **adding a new post** in the **posts module**.

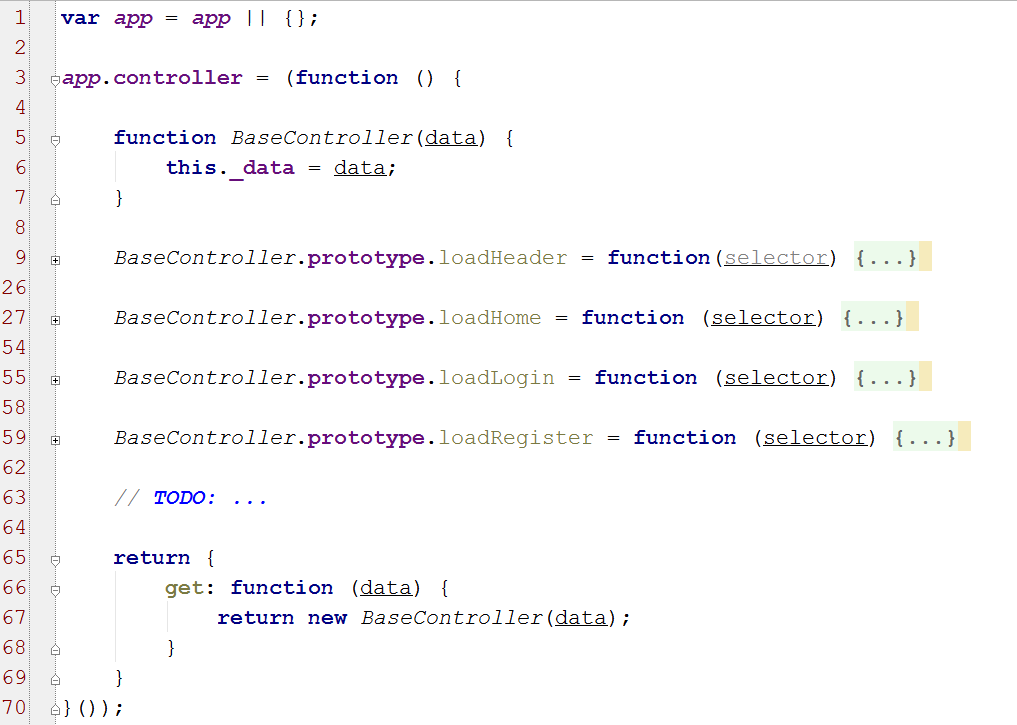
1. Reveal only the **users** and **posts** modules from **data**.
2. Attach the **data** module to the **app** object.

# Controllers

Now that we have a way to access remote data through the **data.js** script, it's time we consumed it.

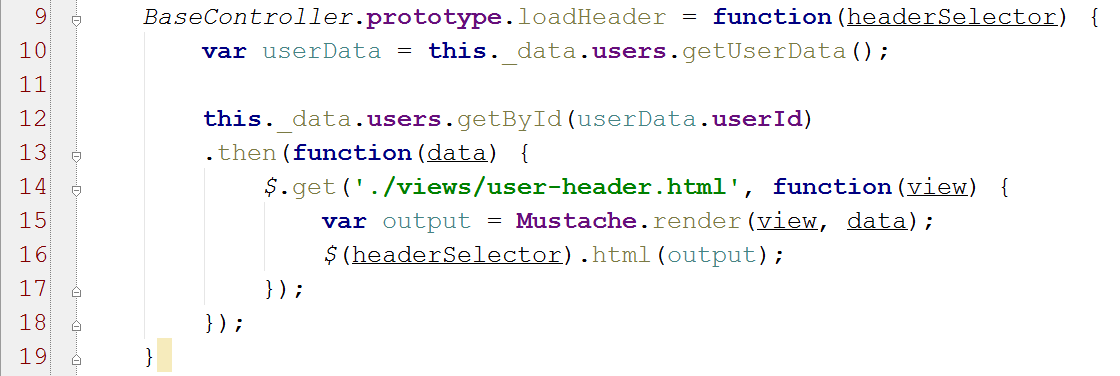
Controllers offer functionality for something to happen. For the sake of simplicity, we will create only one controller in the **controller.js** script.

1. Create a controller module that receives data and encapsulates functionality for loading different parts of our single page application.



Through dependency injection the controller receives the **data** it will use. The different functions will be attached to the prototype and will be called from outside whenever a certain piece of HTML needs to be loaded.

For example, the header page will be loaded as such:



In other words, we:

* + - 1. Get data about the currently logged user
      2. Load the specific partial view from the server
      3. Tell Mustache to render the view with the given data
      4. Append partial view to selector

The view should define **placeholders** (keywords wrapped with **{{key}}**) where we want the data to be injected by Mustache, where **key** is the respective key from the provided object:

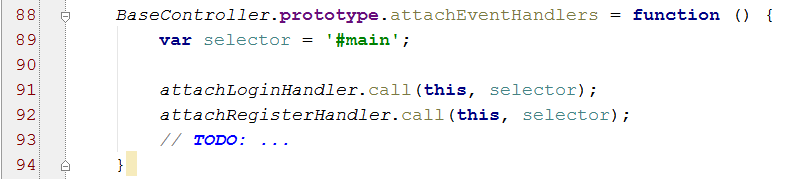
|  |  |
| --- | --- |
| <**div class="image-container"**>  <**img class="thumbnail" src="{{picture}}"**> </**div**> <**div class="summary-container"**>  <**h2**>Hello, {{name}}!</**h2**> </**div**>  … | data = {  name: "Pesho",  picture: "data..",  ...  } |

1. Validate that user is logged in. In case he/she hasn't, do **not** display the header view.

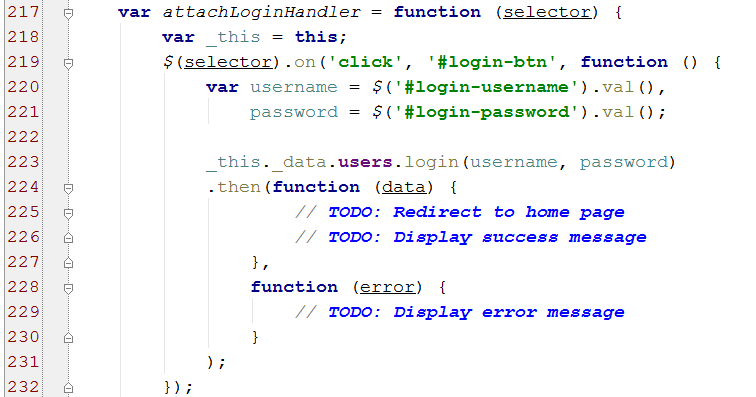
Do the same for each load function in controller by following the 3 steps: **validate user**, **load data** and **inject data in partial view** and **append the result HTML to DOM**.

### Event Listeners

Add a function to the controller **attachEventHandlers()** that adds **event handlers** on several **components** on the DOM.



For example, it attaches an event listener for the submit button on the login page.



Define functions for all possible events on our SPA we would wish to be handled.

# App

App will be the starting point of our application. There we define our **requester**, **data** and **controller**.

The data receives the **requester** it will use to make requests and the **base URL** of the target service. The controller itself receives the **data** and we tell it to attach all **event listeners** to DOM.

Below that, we define a **router** using **SammyJS**. The router is used to listen for URL changes in our app. Whenever it detects such, it invokes specific controller actions. For example, whenever the URL is **'#/'**, the **loadHeader()** and **loadHome()** functions from the controller are invoked.

