JavaScript Twitter exercise

In this exercise we'll use JavaScript, both on the server and client. We'll use Node.js to provide pages and authentication, as well as a RESTful API for providing Twitter API data.

Task 0.1: Ease into it...

We'll start of gently by just implementing a view using the template engine EJS. By using a middleware (layer between initializing loading of a page, and the presentation of that view), we are provided a session object located within the `request` object (i.e. `req.session.User`).

In EJS we can use inline JavaScript. So doing `<%= user.SOME_FIELD %>` will print the value of SOME_FIELD located on the user-object. In the same manner we can use IF statements:

```
<% if ( 'someLogicalExpression' === 'here' ) { %>
<strong>My content here</strong>
<% } %>
```

1. Try to login using a dummy Twitter account and see how the boilerplate looks.
2. Verify that a User object is sent into the index view in the Pages controller.
3. Find out what the User object contains.
4. Choose values you wish to print.
5. Implement the view located in `views/helpers/user.ejs`.
6. Verify that a User now is visible on the page.

Task 1: Implement timeline

Task 1.1: Providing Timeline!

Now let's implement some RESTful functionality. We'll start by providing an entry point for retrieving a timeline for the authenticated user. All of the REST is located in the API Controller.

The API Controller has access to a twitter object. To request data from the Twitter API, we can use the `twitter.twit` object.

The Twitter API documentation regarding timeline can be found here:
https://dev.twitter.com/docs/api/1.1/get/statuses/home_timeline

1. Try opening up the local API by running http://localhost:3000/timeline.json
2. Create a object literal defining what parameters to send into the Twitter API. You can access the incoming params by using `req.param('foo')`. So if you try to access "http://localhost:3000/timeline.json?foo=bar", `req.param('foo')` will be valued `bar`.
3. Use the `twitter.twit.get()` method to request data from Twitter. Signature looks like this: `twitter.twit.get(URL, DATA_OBJECT, CALLBACK_FUNCTION)`
4. Implement the callback function. twit.get() calls the callback with two arguments: `error` and `response_data`.
5. Verify that you get a timeline with 10 tweets by opening http://localhost:3000/timeline.json?count=10.

Task 1.2: Showing timeline!

Now we can implement some client side features and use our API to show some lovely data.

Be aware: This might get hairy…

Subtask: Implement fetch().

Fetch is a function which should be used to get the JSON data. It will run on the initiation for a module. It can be reused for several modules (timeline, search, favorites etc). To support this we have a module called "base.js". Here all reusable scripts can be stored. All modules extend from this object literal. So when using "this" in a method in "base" refers to the main plugin (e.g.

Timeline). So for instance you can access the timeline API URL by using `this.options.getURL`.

> ### jQuery AJAX information
>
> The jQuery AJAX method is located on the $ object ($.ajax()).
>
> It can take a object as an argument. For instance

```
{
    url: 'someValue',
    data: {'someData':'value'},
    dataType: 'what kind of data are you sending in?',
    contentType: 'what kind of data do you expect?'
}
```

> The `$.ajax` call returns a jqXHR object. You can use this as a sort of "promise" that you'll get notified when AJAX responds. Adding a success callback by doing `jqXHRObject.done(callback)` and a fail callback by doing `jqXHRObject.fail(callback)`. The done callback passes a response object as argument.
>
> **NB**: Even if the done() callback is triggered, we can still have an error. E.g. when twitter responds with an error message in a JSON format.

1. Do an AJAX request to the API and send in arguments passed in as a object literal as data to the AJAX call.
2. Use jQuery's $.extend() method to create a new object literal thats the merge between the plugin default ajaxOption and the user provided options through the parameter.
3. Implement a callback to when the AJAX call is successful. Print response (console.log)
4. Implement a callback to when the AJAX call fails. Print response
5. Run the project to verify that the API gives the correct response.

Remember: You should return the jqXHR object from the fetch-method. This way we can add more functionality on done() if we want to do so later on.

To notify the plugin that we have result we'll need to know some about publish/subscribe (publisher/observer) pattern.

> **Short introduction to pub/sub**
>
> Pub/sub is a pattern used to make code more testable and to avoid massive callback structures when using asynchronous languages.
>
> In jQuery you can trigger events and listen to events. Trigger events by doing `$("some-element").trigger('event-name', ['someData'])`.
>
> It is best practice to namespace your events. For instance having the event be named "pluginName.event".

1. Trigger the event **tweetsLoaded** on the correct namespace for when we have success data.
2. Trigger the event **error** when an error occurred.

The events must be triggered in the correct context. The plugin's main element makes sense here.

Subtask: Render our result!

Almost there. We'll just need to implement our rendering of the tweets. Let's just verify that we do in fact reach the render method.

1. Try logging the data in the render method.

Now we know the render method is triggered, and we have access to the Tweet-data. Let's first edit our view to print correct information.

> **Some information about mustache.js and ICanHaz.js**
>
> Mustache is a simple template engine for JS. We can print variables using

```
{{ someVariable }}
```

> and we can use

```
{{#someCondition}} <strong>foo</strong> {{/someCondition}}
```

> to do IF statements.
>
> ICanHaz.js is a layer onto of mustache, to be able to separate our templates and the JS. We have 5 templates saved: error, favorites, search, timeline and tweet_template. All of these are included into our DOM. By giving the templates names, we can access them with ICanHaz in the following manner:

```
var htmlObject = ich.myTemplateName(myObject);
```

> Where `myTemplateName` should be replaced with the ID for a given template. `myObject` is the object we want to fill the template with.

1. Locate the Tweet Template file.
2. See example tweet data from the Twitter API Docs.
3. Fill out the template with the Twitter data.

Now we can use this finished template to populate our DOM with beautiful tweets.

1. Use the render method and iterate over all tweets given by data.
   1. Convert from tweet object to html template object.
   2. Add HTML to DOM. You can find what container to add data to by looking at the Timeline Plugin defaults settings and the init method.

Remember: `this` inside a method in `base` points to the Timeline Plugin object.

Subtask: Render errors

Sometimes we'll get errors. We need a good way to inform our user if something happened. Implement the renderError() method in the same way as our render method.

1. Locate the Error JS Template.
2. Fill out the proper elements.
3. Try provoking an error by sending in faulty parameters to the API (e.g. ?count=FOO)
4. Use the error template and add the error to the DOM. (Same as with

previous subtask.)

That's it! Now we should have a fully working timeline. But how about when new tweeters tweet? Let's implement a refresh method!

Task 1.3: Refreshing the timeline

We're now done with the `base.js` file. We have a working base object to use for the other modules (like search and favorites).

First we need to update the JSON API. Probably we haven't implemented all arguments to pass on to the Twitter API. For this task we need to pass in the `since_id` argument. This simply tells Twitter that we need all the tweets in the timeline feed posted after the tweet with the ID `since_id`.

1. Update the JSON API to send the URL parameter since_id into the `twitter.twit.get()` method.

We wrote (hopefully) the fetch() method in such a way that we can send in a object as a argument and we'll call the AJAX url with the argument we pass in as a part of the object. So for instance, if we run `module.fetch({since_id: 32132132})`, `fetch()` will pass this on to the JSON API.

1. Every time `fetch()` is executed, we need to save the ID of the newest tweet. (This can be saved directly on the plugin, i.e. this.since_id = SOMETHING).
2. Implement the refresh() method. (Remember the since_id value).

(Hint: Remember, we returned the jqXHR object in the fetch method.)

Task 1.4: Updating our status

Now we'll implement the possibility to tweet (i.e. update our status). First we need to expand our JSON API.

Subtask: Updating the JSON API

Look at the Twitter API documentation here:

https://dev.twitter.com/docs/api/1.1/post/statuses/update.

1. Locate the statusPost method.
2. Implement the method by taking parameters from the req.param()-method and passing it on the the `twitter.twit.post()` object.
3. The `twitter.twit.post` signature looks like this:

```
twitter.twit.post(TWITTER_PATH, INPUT_ARGUMENTS, CONTENT_TYPE
```

Subtask: Implementing AJAX Post

We need to be able to send a POST request to our JSON API. We'll use the `$.post` . Method for that. This method, as with the `$.ajax()` method, returns a jqXHR object.

The `$.post` method signature looks like this:

```
var jqXHR = $.post(URL, DATA_ARGUMENTS);
```

Remember to look in the plugin default options for any useful information

1. Implement the post method.
2. Trigger the error event if fail.

Subtask: Update status method

Now we can implement the logic for updating a status.

The updateStatus method will automatically be executed when the form is submitted. We'll need to implement functionality in this method.

1. Use the `this.$form` jQuery object (of the input form) to fetch the text area value.
2. Validate input and do nothing if invalid.
3. Construct a object literal to send to the post-method.
4. Execute post.
5. Refresh timeline if successful post.

Remember: You'll need to prevent default behavior of a form submit to avoid refreshing site.

Task 2: Searching the twitossphere

Now we have successfully implemented the entire timeline.

I hope this wasn't too difficult and poorly constructed, exercise wise. The good thing is the worst is behind us. Now lets do some easy extensions for our Twitter Client. First we'll implement search!

Task 2.1: Extending the JSON API.

We got this now.

1. Locate the search API method.
2. Look at the Twitter API docs for what arguments to send.
3. Construct an argument object literal to pass on to the Twitter API.
4. Use the `twitter.twit.get` method.
5. Print the result.
6. Verify that the search works by visiting http://localhost:3000/search.json?q=realDonaldTrump

Task 2.2: Navigate through pages.

We need to have an easy way to swap to pages. We see in the `base.js` file that we have a method called `showPage`. We'll use this to navigate between pages.

Each page has it's own HTML container in the Index template. By default only the Timeline container is visible. If we want to show the Search page, we need to hide all other pages and show only the Search page.

1. Locate the `showPage` method.
2. Show the current page (we always have access to the page's container element by doing this.$el)
3. Hide all other pages.

All pages have the CSS class `page-role`.

When implementing this method we can navigate through the different sites.

Task 2.3: Implement search

When submitting the search field, the search method will be triggered.

1. Locate the search method in the Search client controller (plugin)
2. Find the search input field by using the Search page container element and the formSelector in default options.
3. Generate the object literal of arguments to send to the Twitter API.
4. Get the results via AJAX.

That's it for the entire search module! Easy!

Task 3: Favorites

As a last task, we'll implement a list of the authenticated users favorite tweets.

Task 3.1: Extending the JSON API

1. Locate the favorites API method.
2. Look at the Twitter API Docs to see what arguments to send.
3. Construct an argument object literal to pass on to the Twitter API.
4. Use the `twitter.twit.get` method.
5. Print the result.
6. Verify the results by visiting http://localhost:3000/favorites.json?count=50

Task 3.2: Go at it..

Implement the rest of the favorites.