# 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 #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 by 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 #2: 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 #3: 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 "promise". 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. Implement a callback to when the AJAX call is successful. Print response (console.log)
3. Implement a callback to when the AJAX call fails. Print response
4. Run the project to verify that the API gives the correct response.

To notify the plugin that we have result we'll need to know some about publish/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 practise 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.