# Lab 1

# Client-side Development using HTML, Javascript and CSS

Spring 2020

TDDD97 Web Programming

http://www.ida.liu.se/~TDDD97/

Department of Computer and Information Science (IDA)

Linköping University

Sweden

# 1. Introduction

In this first lab, you will use the components of HTML5, that is HTML, CSS and JavaScript, to implement the client-side of Twidder. You have also been provided a server-stub, which works as your not yet written server-side code. This helps you have a fully functional application by the end of lab 1 and be able to see the output at an early stage. Later on, when you have your own server-side code implemented, the server-stub will be removed from the application and you will use your own server-side code. You will read more about the provided server-stub in section 2.

Each lab has been divided into several continuous steps which each adds a new piece to the final application and make the product more complete. You are required to develop step by step and implement each step according to the instructions. Once you are finished with each lab, you will present your work to your responsible lab assistant. For more specific information about the presentation and evaluation process of lab 1, please check section 7: Presentation and Evaluation. For more information about the examination process, please check the course page.

### Requirements

By the end of lab 1, the following functional requirements shall be met.

#### Functional

- The user shall be able to sign up, sign in and sign out.
- The user shall be able to view his/her own personal information provided during sign-up, everything excluding the password, once signed in.
- The user shall own a message wall which other users and himself/herself can post messages on it.
- The user shall be able to refresh his/her own wall to check for any newly posted messages, without refreshing the rest of content. In other words, the browser's refresh button shall not be used.
- The user shall be able to view another user's personal information, everything excluding their password, and message wall by providing his/her email address.
- The user shall be able to post a message on another user's message wall.
- The user shall be able to refresh another user's message wall to see the newly posted messages, without refreshing the rest of the content.
- The user shall be able to change his/her password while being signed-in.

#### Non-Functional

- Once the application is opened for the first time, it will not require to refresh itself during its lifetime. Such applications are called Single Page Applications or SPAs. For more information please check out section 4: Application Structure.
- Using "window.alert()" or similar types of window based inputs/outputs is not allowed.
- The HTML code shall validate in https://validator.w3.org/ . **The HTML code in the script elements which represent different views shall be validated separately by copying the code and pasting in the validator.**
- The CSS code shall validate in https://jigsaw.w3.org/css-validator/
- It is not allowed to use JQuery.

# 2. The Project Folder

The project folder is an ordinary folder that contains all the required files building your application. By the end of lab 1, your project folder will contain the following files:

- client.html //created
- client.css //created
- client.js //created
- serverstub.js //provided

## client.html

The client.html file will contain all of the HTML code implemented by you which mainly shapes the UI. It also uses all other files mentioned in the list above. As you proceed, you will add more HTML code to this file, moving towards a fully functional web application.

**Figure 1 - Initial content of client.html.**

```
<html>
    <head>
       <link href="client.css" type="text/css" rel="stylesheet">
       <script src="client.js" type="text/javascript"></script>
       <script src="serverstub.js" type="text/javascript"></script>
    </head>
    <body>
    </body>
</html>
```

## client.js

The client.js file will contain all the functionality you implement. As you proceed, you will add more Javascript code to this file, moving towards a fully functional web application. Make sure that you separate different functionality in different methods from the beginning, making different pieces of the code easy to use and re-use.

**Figure 2 - Sample code in client.js.**

```
displayView = function(){
   // the code required to display a view

};
window.onload = function(){
   //code that is executed as the page is loaded.
   //You shall put your own custom code here.
   //window.alert() is not allowed to be used in your implementation.
    window.alert("Hello TDDD97!");

};
```

## client.css

This file will contain all the styling you implement. You will implement this file completely from the scratch as you proceed step by step.

## serverstub.js

This file simulates the server-side code for Twidder application in Javascript at client-side. There are three reasons behind providing "serverstub.js":

1. Without knowing what the server-side functionalities are, what each takes as input and what each returns as output, implementing the client-side cannot be completely done.
2. As every web application is the combination of the client-side and the server-side working together, by having the server-side in hand and implementing the client-side by the end of lab 1 you can have a fully functional application in an early stage.
3. As "serverstub.js" is in Javascript language and is located at client-side, there is no need to be concerned about client/server communication at current stage and you can simply use the provided javascript methods simulating different server-side functionalities.

For example, the method `serverstub.signUp(formData)` does sign up a user based on the provided information. then you can use `serverstub.signIn(username, password)` to sign in the same registered user. In lab 2, you will have the chance to reimplement the "serverstub.js" in python language at server-side. By the end of lab 3, you will remove "serverstub.js" and will connect your client-side and server-side code, implemented in labs 1 and 2, together making a fully functional application.

All the server-stub methods will return an object containing the fields "success", "message" and "data". The "success" field contains a boolean telling if the request has been successful or not and the "message" field contains an error message if something has gone wrong or a success message if the request has been completed successfully. The "data" field contains the returned data from the server-stub if the method is expected to return any other data, otherwise it is undefined. You can find the documentation for the methods provided by ServerstubJS at the end of this document.

# 3. Development Tools

To complete this lab you will need a text/code editor and a web browser. For both cases you are free to use any tool of your choice as long as:

1. The text/code editor of your choice does not provide any visual web designing capability.

2. The web browser of your choice supports HTML5. We recommend Google Chrome and Mozilla Firefox because of their good support for web development and the tools they provide. It is highly recommended to use the “Web Console” and “Element Inspector” tools provided by both of the browsers during the development.

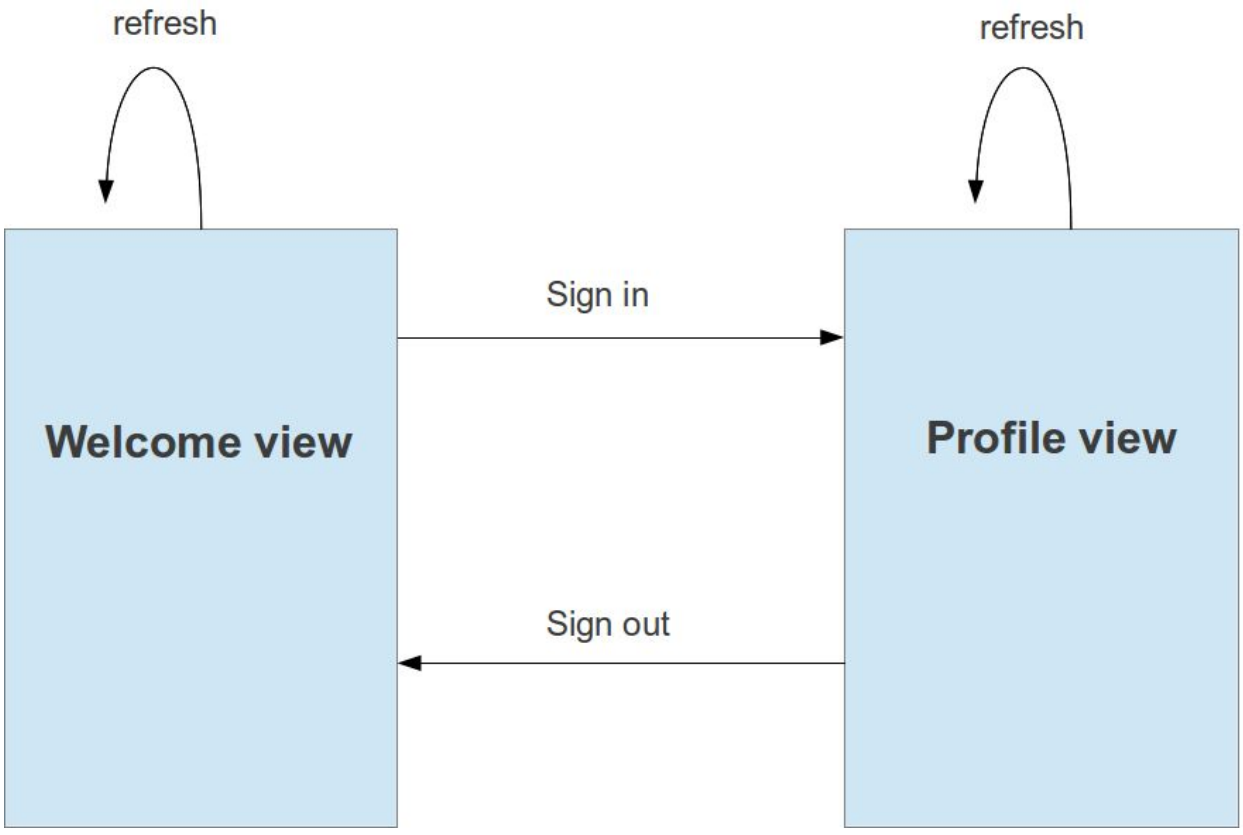
**Note: using visual web designers is not allowed and is considered as cheating.**

# 4. Application Structure

Twidder is a **Single Page Application,** which is composed of two views(states) named Welcome and Profile. Depending on the user is signed-in or not, a view is selected and displayed to the user. Only one of the two views can be displayed at a time. As the user opens the “client.html” file for the first time, the Welcome view is displayed to the user which enables him/her to sign up and/or sign in to the system. Once the user is signed-in, the Profile view is displayed, which enables the user to use the provided application functionalities. The user has the possibility to sign out and return to the Welcome view/page.

As the user refreshes the page, the current view is maintained and the user shall be able to continue his/her work. At the same time, if the user closes the “client.html” page and opens it again, the previously loaded view shall be loaded again and the user can continue his/her work. For example, if the user signs in and closes the page before signing out, the Profile view shall be loaded once the user reopens the “client.html”.

**Figure 3 - The state diagram of the Twidder application.**




Note: By the end of lab 1, your final product is expected to behave accordingly.

# 5. Lab Instructions

Lab 1 is divided into several steps to help you get a good workflow. Each step adds a new part to the application until you have a fully functional application by the end of lab 1.

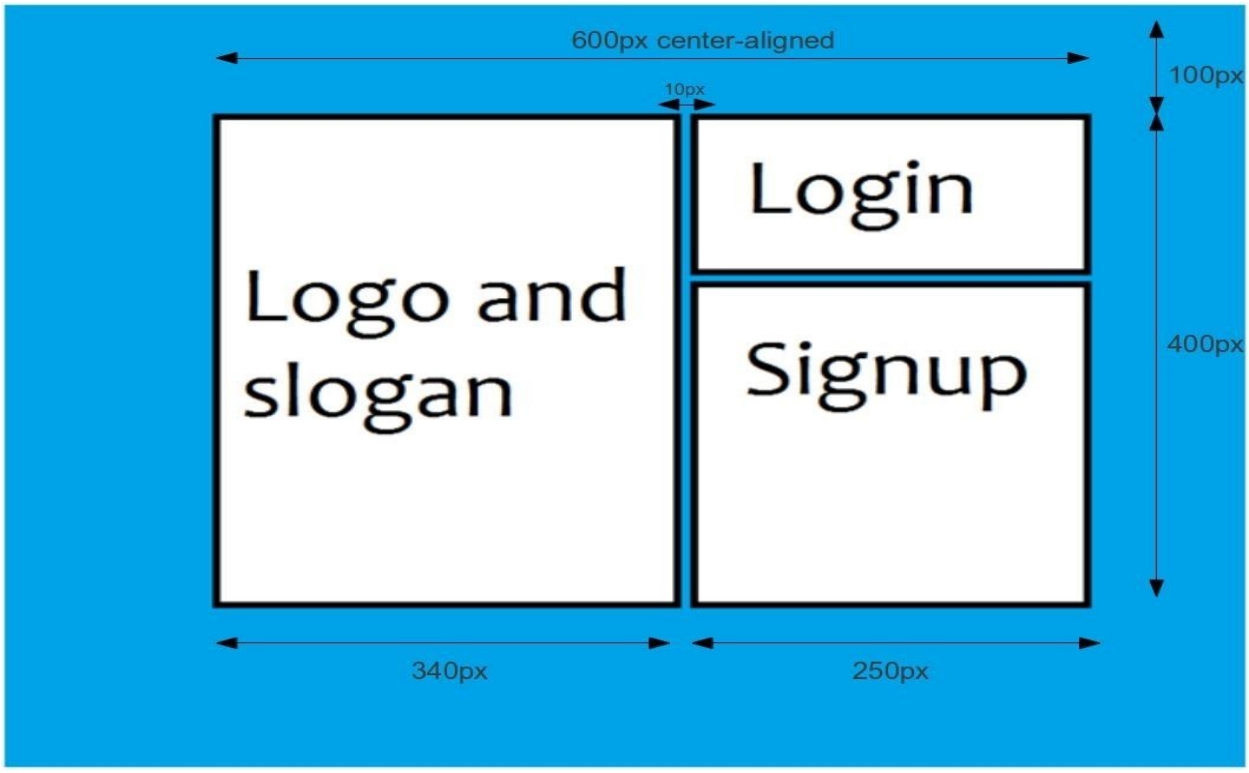
## Step 0: Creating the project folder

In this step, you are required to create a project folder based on the information available in Section 2. As a start, you need to create a folder with an appropriate name on the file system which will accommodate all the files building up the Twidder application. As mentioned in Section 2, you need to create three files from scratch in the project folder and add one provided file named "serverstub.js" available on the course page. "client.html" needs to have enough code to use all other files inside of the project folder. "client.css" and "client.js" are the only empty files for now. Figure 1 demonstrates the initial content of "client.html".

## Step 1: Designing the welcome view

In this step, you will create the sign-in and sign-up forms using HTML and CSS in a temporary HTML file named welcome.html, which is using a temporary CSS file named welcome.css. The sign-in and the sign-up forms shall be separate, and will not be providing any functionality at the current stage. Your page shall be laid out as demonstrated in Figure 4. You shall use CSS for styling the layout and **using HTML tables is not allowed.**

**Figure 4 - The Welcome view layout.**



**The sign-in form shall require the following information:**

1. Email as username
2. Password

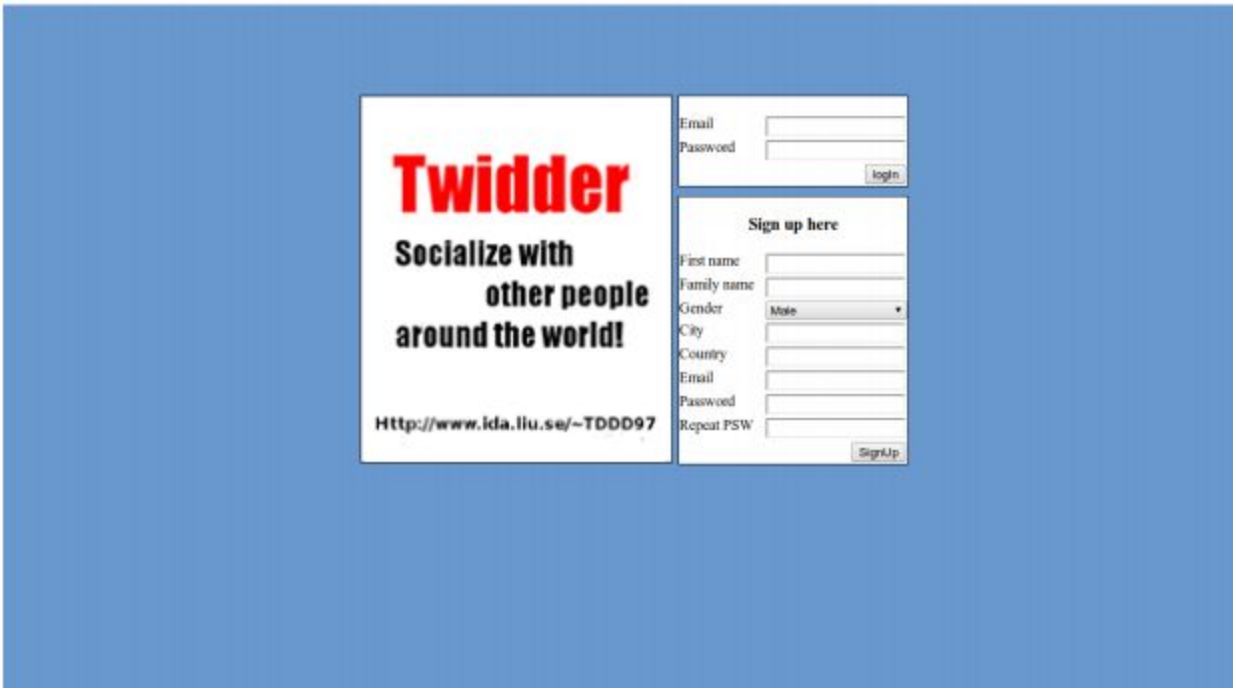
**The sign-up form shall require the following information:**

1. Email as username
2. Password
3. Repeat password
4. First name
5. Family name
6. Gender
7. City
8. Country

Each piece of information in both forms like username or gender is requested as a single row including a label and an input element. Each form also contains a submit button which appears at the last row.

**Note: An image containing a logo and slogan is provided to be embedded inside of the welcome view in the specified area. You can also use a custom image to be displayed.**

**Figure 5 - A sample output of the Welcome view.**



## Step 2: Displaying a certain view

The purpose of this step is to build one HTML file for the whole application that is capable to display different views of the application depending on whether the user is signed-in or not.

Add the "welcome.html" content, only the HTML code which resides inside of the `body` tag, to "client.html" and place it inside a `<script type="text/view"></script>` tag with an appropriate `id` (e.g., "welcomeview"). All `script` tags should be located in the head of the HTML file. Then copy all your CSS code from "welcome.css" to "client.css". From this point we do not need the "welcome.html" and "welcome.css" files anymore. You will also need to create another `script` tag, which will be completed in the following steps. It should have an easily accessible `id` (e.g., "profileview") and only contain an arbitrary line of text. Each of the above `script` tags shall contain sufficient HTML code to represent a view.

Once the "client.html" document is loaded, we need to decide which view should be displayed depending on whether the user is signed-in or not. Because there is no login mechanism implemented yet, for now you need to display the Welcome view once "client.html" is opened and loaded.

### How it works?

Once a piece of HTML code is placed inside of `<script type="text/view"></script>` tag, the browser skips rendering it because it does not recognize the type, "`text/view`" is a type chosen by us and is not known to the browser, and only adds the `script` tag to the DOM tree containing some unrecognized script. Depending on which state the application is in, signed-in or not, this piece of script representing a view can be read and inserted to the page once the page is loaded.

### How to know the page is loaded?

Once the page is loaded completely by the browser and the DOM tree is built, the method `onload` belonging to the `window` object is called automatically by the browser. By overloading the `onload` method, a custom piece of code, in this case the code for reading and inserting HTML code, can be executed every time the page is loaded or reloaded.

### How to read and insert HTML code?

Because the `script` tags are part of the DOM tree and given that they have an appropriate `id`, they can be accessed and their content can be read. For inserting the read content, which is a piece of HTML code, a tag located in the `body` is required, which will accommodate the read content. A `div` tag with an appropriate `id` is the most preferable one because it has been designed for providing sections. Once the HTML code is inserted, the DOM tree is updated and the added content is displayed automatically. In other words, the HTML code is copied from a `script` tag and pasted inside of a `div` tag.

## Step 3: Implementing user input validation for sign-in and sign-up forms

While performing the sign-in and sign-up operations, we prefer to minimize the load and traffic on/to the server. One way of doing that is to make sure we do not send unnecessary login and signup requests that will be rejected by the server because of incorrect or inadequate information. In this step, you will make

sure the sign-in and sign-up forms hold valid information before a request is sent to the server. This means unless the following criteria are met, this can of course be different application to application, no request shall be made when the submit button is clicked:

1. No field shall be blank.
2. The email field shall contain a well formatted email address
3. For signup, both password fields must contain the same string.
4. The password is at least X characters long in both forms.

Note: The same validation needs to be done at the server-side in lab 2.

To implement the validation mechanism you need to use HTML and Javascript both and implement a method inside of "client.js" for each form which handles the validation whenever the user clicks on the submit button.

### How it works?

Thanks to HTML5 part of validation, items 1 and 2, can be achieved by using a certain attribute, **required**, and choosing a correct input type. For example the type "email", make the form do the validation automatically and check if the content is a well formatted email address or not. You can also use the attribute "required", to make a field mandatory or not.

Implementing items 3 and 4 takes more effort and you need to use Javascript for performing the validation. You will use the form's "onsubmit" attribute or the element's "oninput" event attribute to call your custom validator method before the actual submission. If any field contains invalid data then the user needs to be notified. You can use a custom-made feedback area inside of the page, no alert allowed, or use HTML5 validity errors.

## Step 4: Adding the sign-up mechanism

In this step you are required to add the sign-up functionality by calling `serverstub.signUp(dataObject)` after the validation process is succeeded. As mentioned in section 2, the data passed to `serverstub.signUp(dataObject)` is in form of a Javascript object with the following fields:

1. `email`
2. `password`
3. `firstname`
4. `familyname`
5. `gender`
6. `city`
7. `country`

If the user tries to register with an email address that is already registered, this is to be shown in the same way as faulty fields during the validation, this time the error message returned from the server-stub shall

be displayed.

If the sign-up succeeds, the message returned by `serverstub.signUp(dataObject)` shall be displayed or the user shall be directly directed, logged in, to his/her page.

## Step 5: Adding the sign-in mechanism

In this step you are required to add the sign-in functionality by calling `serverstub.signIn(username, password)` after the validation process is succeeded. As mentioned in section 2, the data passed to the `serverstub.signIn(username, password)` function is in form of two separate string arguments:

1. Email as username
2. Password

To keep track of the signed-in user, the server-stub uses a token-system. Once the `serverstub.signIn(username, password)` method is called and sign-in process is completed successfully, the server-stub creates a randomized token and returns it to the client stored in the field "data" of the returned object. This token has to be stored at client-side to be used further for sending requests to the server-stub. This shall be done using web storage facility as part of HTML5. In other words, if the user is signed-in then a token exists at client-side representing the signed-in user otherwise there is no such token stored at client-side.

As you remember, in step 2 you were required to display the Welcome view as the page was loaded because at that time no sign-in mechanism was in place. now that part needs to be changed and decide which view needs to be displayed based on if there is a token at client-side or not. This means, once the page is loaded/reloaded the right view shall be displayed. The only issue is that for transiting from Welcome view to Profile view after signing in, the user needs to refresh the page. To solve the issue, the deciding code should be called manually after the token is stored at client-side as it is called automatically after each page load/reload.

If the username or password is incorrect the returned error message by the server shall be displayed to the user and no token shall be generated.

## Step 6: Implementing tabs

Once the user signs in, the Profile view is displayed to the user. Currently, this view contains nothing but a single line of text added earlier. In this step you shall implement a row of tabs which each is associated to a panel. There shall be three tabs with the following names:

1. Home //selected by default
2. Browse
3. Account

The user can only choose one tab at a time and the associated panel with its content is displayed consequently.

### How it works?

To implement such tabular data presentation, you can use CSS `display` property. once a tab is clicked, the `display` property of the selected panel is set to `block` as others' are set to `none`. It is a good idea to implement each panel as a separate `div` tag. You can also use any other technique of your choice.

**IMPORTANT**

- The selected tab shall be marked.
- Once the user leaves a tab for a while and comes back to it, the data in the tab shall be preserved without asking the server stub again.

## Step 7: Implementing the Account tab

It is considered a good practice to change password every now and then. To give the user the possibility to change his/her password, you are to implement password change functionality inside of the panel associated to the Account tab. To do this, you need to use an appropriate method provided by "serverstub.js".

**IMPORTANT**

- The user shall see a feedback for any problem regarding changing password. For example, wrong old password. The message coming from the server-stub shall be shown to the user.
- There shall be two text fields for entering the new password. If the passwords don't match then an error message shall be shown to the user.

Last but not least, the user shall be able to sign out from the system when he or she is done with his/her session. In this step you are also required to implement this functionality inside of the panel associated to the Account tab by using an appropriate method provided by "serverstub.js". After the user is signed out, he/she needs to be redirected to the Welcome view automatically.

## Step 8: Implementing the Home tab

In this step you are to implement the Home tab. No matter what styling it has, it needs to include the followings:

1. The signed-in user's personal information provided at sign-up process except the password.
2. A text area and a post button which the user can use to post messages to his/her own wall.
3. A list of all posted messages forming a wall.
4. A button to reload only the wall in order to see the newly posted messages by other users.

To get it done, you also need to use the methods provided by "serverstub.js".

## Step 9: Implementing the Browse tab

In the previous step you implemented the Home tab. In this step you are to implement the Browse tab which enables the signed-in user to view other users' Home page. The Browse tab shall enable the signed-in user to enter the e-mail address of another user via a text field and a button. When submitted, the found user's Home page shall be displayed. The view of another user's Home page shall be functionally identical to your own Home page. In other words, the signed-in user can view the personal information and wall of other users and at the same time post messages on their walls and reload their wall to see whether there is any new posted messages on their walls.

To get it done, you also need to use the methods provided by "serverstub.js".

**IMPORTANT**

- The user shall receive a feedback if the searched user does not exist.

# 6. Questions for consideration

1. Why do we validate data before sending it to the server at client-side, as opposed to just letting the server validate data before using it? What we get and what we lose by it?

2. How secure is the system using an access token to authenticate? Is there any way to circumvent it and get access to private data?

3. What would happen if a user were to post a message containing JavaScript-code? Would the code be executed? How can it oppose a threat to the system? What would the counter measure?

4. What happens when we use the back/forward buttons while working with Twidder? Is this the expected behaviour? Why are we getting this behaviour? What would be the solution?

5. What happens when the user refreshes the page while working with Twidder? Is this the expected behaviour? Why are we getting this behaviour?

6. Is it a good idea to read views from the server instead of embedding them inside of the "client.html"? What are the advantages and disadvantages of it comparing to the current approach?

7. Is it a good idea to return true or false to state if an operation has gone wrong at the server-side or not? How can it be improved?

8. Is it reliable to perform data validation at client-side? If so please explain how and if not what would be the solution to improve it?

9. Why isn't it a good idea to use `tables` for layout purposes? What would be the replacement?

10. How do you think Single Page Applications can contribute to the future of the web? What is their advantages and disadvantages from usage and development point of views?

# 7. Presentation and Evaluation

Once you are finished with lab 1, you will present your work to your responsible lab assistant during a scheduled lab session. You may be asked about the details of your implementation individually.

# 8. Documentation for serverstub.js

Here is the documentation for Serverstub.js which helps you get started with its services, functions, without any need for going through their implementation.

All services, functions, return a javascript object containing 2 fields:

**success**: True if the operation is successful and False if the operation failed.
**message**: The message returned from the function after completing or failing the operation.

As mentioned in the following, responses returned from some services contain a third field, named **data**. This field contains the actual data returned by the service.

Example:
signin() can return {"success": true, "message": "Successfully signed in.", "data": token}

**Public methods provided by "serverstub.js"**

`serverstub.signIn(email, password)`

**Description:** Authenticates the username by the provided password.
**Input:** two string values as username and password.
**Returned data:** A string value containing a randomly generated access token if the authentication is successful.

`serverstub.signUp(dataobject)`

**Description:** Registers a user in the database.
**Input:** An object containing the following fields:
email, password, firstname, familyname, gender, city and country.
**Returned data:** -

`serverstub.signOut(token)`

**Description:** Signs out a user from the system.
**Input:** A string containing the access token of the user requesting to sign out.
**Returned data:** -

`serverstub.changePassword(token, oldPassword, newPassword)`

**Description:** Changes the password of the current user to a new one.
**Input:**

- token: A string containing the access token of the current user
- oldPassword: The old password of the current user
- newPassword: The new password

**Returned data:** -

`serverstub.getUserDataByToken(token)`

**Description:** Retrieves the stored data for the user whom the passed token is issued for. The currently signed in user can use this method to retrieve all its own information from the server-stub.
**Input**: A string containing the access token of the current user.
**Returned data:** A user object containing the following fields:
email, firstname, familyname, gender, city and country.

`serverstub.getUserDataByEmail(token,email)`

**Description:** Retrieves the stored data for the user specified by the passed email address.
**Input:**

- token: A string containing the access token of the current user
- email: The email address of the user to retrieve data for

**Returned data:** A user object containing the following fields:
email, firstname, familyname, gender, city and country.

`serverstub.getUserMessagesByToken(token)`

**Description:** Retrieves the stored messages for the user whom the passed token is issued for. The currently signed in user can use this method to retrieve all its own messages from the server-stub.
**Input:** A string containing the access token of the current user.
**Returned data:** An array containing all messages sent to the user.

`serverstub.getUserMessagesByEmail(token,email)`

**Description:** Retrieves the stored messages for the user specified by the passed email address.
**Input:**

- token: A string containing the access token of the current user
- email: The email address of the user to retrieve messages for

**Returned data:** An array containing all messages sent to the user.

`serverstub.postMessage(token, message, email)`

**Description:** Tries to post a message to the wall of the user specified by the email address.
**Input:**

- token: A string containing the access token of the current user
- message: The message to post
- email: The email address of the recipient

**Returned data:** -