Through this first mandatory individual assignment, you will get hands-on practice with basic front-end web programming and the use of application programming interfaces (APIs). We will start off with a small web page using HTML and CSS before we move on to JavaScript and Ajax.

Most of the fundamental knowledge will be or has been covered in the lectures, but some assignments require you to explore additional resources on the web. Here, links are provided. Also, just as with all types of programming, Google might be a good companion.

**Delivery**
In Devilry before Friday 14th of September 23:59. Upload one zip-file named after your UiO username, containing one folder for each assignment. Each folder should contain all files necessary to run your solution to the assignment in the browser. As assignment 3.1 builds on the code of 2.1, you can deliver these as one solution in the same folder/file.

As the group teachers will be using Google Chrome for testing, make sure your solutions work well in this browser. Otherwise, there are no requirements for compatibility with older browsers.

## Part 1: HTML and CSS

### 1.1 Personal web page

In this initial assignment, we will be working with HTML and CSS to create a responsive web page that fits all screen sizes. If you want, you can put it on the folk.uio.no server, as a presentation of yourself.

If you have limited experience with HTML and CSS, we recommend reading the HTML5 Style Guide and Coding Conventions before you proceed.

**Step 1: HTML structure**

Create a .html document with the standard HTML-structure (<html>, <head>, <body> etc.). Set up a basic structure following HTML5 semantics that includes a header-element to contain a menu with links to relevant pages and a picture of you. Further, the page should include a section that present projects that you have been part of in a list, and/or similar sections with work experience,
and courses and educational institutions attended. One section presenting one of the examples above is sufficient.

**Step 2: CSS**

Create a .css file and link to it in your HTML-document.

Add styles to the different components of the page, so that its design resembles (something like) the layout illustrated in the figure below.

![Diagram](image)

**Step 3: Responsive design**

Read the following articles:

- [https://www.w3schools.com/html/html_responsive.asp](https://www.w3schools.com/html/html_responsive.asp)
- [https://developers.google.com/web/fundamentals/design-and-ux/responsive/](https://developers.google.com/web/fundamentals/design-and-ux/responsive/)

Make sure that your design automatically fits all screen sizes. For instance, the number of projects displayed horizontally should automatically adjust to the width of the screen. This can be tested in the Chrome browser with *Device Mode*.

**Part 2: DOM, JavaScript, and jQuery**

2.1 Working with the DOM: Interactive list

**Step 1: Add**

Create a new HTML-document with an unordered list element, a text-box, and a button that says “Add.”
Add JavaScript (and/or jQuery) that appends a new `<li>` element to the unordered list when the button is clicked. The text of the new li-element should correspond to the text entered by the user in the text-box. Make sure that the content of the text-box is cleared when the button is clicked to be ready for new input from the user.

![Image of unordered list with items: Norway, Denmark, Sweden, Germany]

**Step 2: Delete**

Add functionality to allow the user to delete a selected list item. When the “delete” button is clicked, the item should be removed from the list.

![Image of unordered list with 'x' symbols: Norway, Denmark, Sweden, Germany]

If you want to find some suitable icons to use as labels check out [Font Awesome](https://fontawesome.com).

**Step 3: Offline storage**

The list will be deleted if we refresh the page, as the data is only stored in our temporary DOM. Traditionally, this is solved by storing the information in a text-file locally, or in a database on the server. Modern browsers do however also enable us to store information locally in a built-in database in the web browser.

By using the HTML5 offline browser storage, make sure that the list is stored in the offline storage. When an item is added, add it to the local storage. When it is deleted; remove it. When the page loads, retrieve the items that exist in the offline storage to populate the list.

Implement the necessary functionality. Read this for a tutorial of the HTML5 web and offline storage:

- [https://www.w3schools.com/html/html5_webstorage.asp](https://www.w3schools.com/html/html5_webstorage.asp)
2.2 Working with functions: List search

In JavaScript, modular code with small, reusable functions is common best practice. In this assignment, we shall create some small JavaScript functions, and then use these to create a searchable list. To make your code structured and readable, make sure you name your functions to reflect their behavior.

Step 1: HTML-document

First, create an HTML-document with an unordered list element in the body (<ul>). This will be our list in which we will add some list elements.

Step 2: List populator

A
Create a function that takes a variable as argument. The function shall create a new list element (<li>) containing the value of the variable, and append it to the unordered list in your HTML-document. For example, if the function is called with the argument “Euler”, a new list-element with the text Euler should be appended to your unordered list.

B
Create a function that takes the following arguments: 1) an array (for example of strings) and, 2) a function. The function should be applied to all elements in the list. For example, if the function is called with the following arguments: ("Euler", “Fortran”, “Lisp”, “Java”, “Simula"), myFunction), your function should iterate through the array of strings, and call “myFunction” with the current string as an argument.

C
Create an array containing names of (some) rooms at IFI. Combine your two functions to populate your unordered list with these rooms. Your result should look something like the figure below.

```
- Euler
- Java
- C
- Shell
- Lisp
- Scheme
- Simula
- Smalltalk
```

Step 3: List Search

Add an input-element to your HTML-document, with the placeholder “Search”. This textbox shall be used to search for different rooms by name. The displayed list of room-names should update
according to the input provided by the user. For example, if the user types E, only rooms starting with “E” should be shown in the list. We will do this by creating two additional functions:

**A**

Create a function that takes in two variables as arguments, called `element` and `searchWord`. If the value of `element` starts with the value of `searchWord`, return true, if not, return false. For example, if the function is called with the following arguments: (“Fortran”, “For”), the function should return true, since `element` (“Fortran”) starts with `searchWord` (“For”). If the arguments are (“Fortran”, “Java”), it should return false.

To make life easier, the following function can be useful:

- [https://www.w3schools.com/jsref/jsref_startswith.asp](https://www.w3schools.com/jsref/jsref_startswith.asp)

**B**

Now, create a function that takes an array (named `list`), and a variable (named `searchWord`), as arguments. The function should iterate the array and call the function created in step 3A with the current list element and `searchWord` as arguments. For instance, a call could look like this: `example(list[i], searchWord)`. All elements where the function evaluates to true should be added to a list which is returned when the array has been iterated.

In the following example, we call the function with a list of names and the search-word (“Li”). Our function returns a list of all elements in the array that starts with the search word.

```javascript
functionName(["Euler", "Fortran", "Lisp", "Limbo"], "Li");
```

→ ["Lisp", "Limbo"];

**Step 4: Full implementation**

Use the functions created in part 2 and 3 to populate the unordered list based on the text provided by the user through the search box. The list should update every time the text in the input-element changes (see figure below). Make sure that the search is not case sensitive, that is, it should ignore whether the user used upper or lower-case letters.
Part 3: Objects, JSON, and Ajax

3.1 Population lookup

You will now build further on your first list-application from assignment 2.1. We want to create an application that allows the user to enter the name of a country, which will be added to the list. In addition to the name of the country, we also want to display the current population figure.

Step 1: Display figures

Find an API that provides you with the population figure of a given country. api.population.io can, for example, provide this through the URL http://api.population.io/1.0/population/Norway/today-and-tomorrow

When the user has entered the name of a country and press add, your application should retrieve the population data from the API, and add the name of the country and its population figure to the list.

If the user enters something that is not recognized as a country by the API, it should not be added to the list, and an appropriate message be shown to the user. You can display this message by using a simple alert box, or by creating a more advanced, but user-friendly Modal Box.

Step 2: Counter

Make the population figures for all countries in your list count upwards/downwards every second to attempt to reflect the current updated figure.

Hint: To find the count rate you can use the population.io-API to get the estimated figure for today and tomorrow and divide the difference between these two on the number of seconds within 24 hours.

Here, you also need to know something about timing events in JavaScript. You can read about this here:

❖ https://www.w3schools.com/js/js_timing.asp

Good luck!