Dr Tomasz Kurowski
t.j.kurowski@cranfield.ac.uk

Introduction to JavaScript tutorial

JavaScript (JS) is a core language and technology for Web pages and applications. It makes it possible to create Web pages which are interactive rather than simply being static HTML pages. Every major browser includes a JavaScript engine which can execute JavaScript code embedded or included in HTML pages, and that code is then able to react to events (such as users pressing a button) manipulate the page structure through the DOM (Document Object Model) interface, which represents the document as a tree of elements (nodes) corresponding to nested HTML tags.

Thus, together with HTML (which provides page structure) and CSS (which provides page styling), JS (as the provider of interactivity) is one of the three most basic technologies for front-end (client-side, i.e. running in a user's browser) Web development.

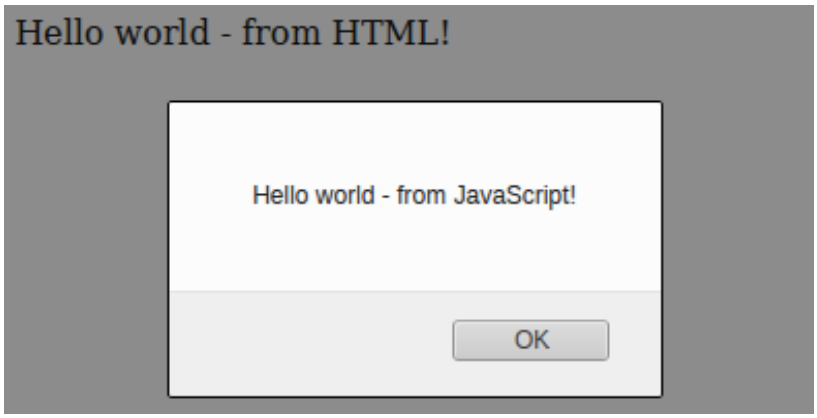
In recent years JavaScript has also seen popular use as a general-purpose programming language, outside of the browser. Using an environment called **Node.js**, it is possible to run JavaScript code without a browser, similar to scripting languages like Python. It is therefore possible to write both the back-end of an application (running on a server via Node.js) and its front-end (running on a user's browser) in the same language.

This practical will provide an overview of basic JavaScript features on both the front- and the back-end, and direct you to more advanced resources such as Web frameworks.

JavaScript in the browser

In the most basic scenario, JavaScript can be directly embedded in an HTML page by using the `<script>` tag. Open your preferred text editor and create a simple HTML page called (you can call it **index.html)** as in the example below:

```
<!doctype html>
<html>
<head>
     <meta charset="utf-8" />
</head>
<body>
    <p>Hello world - from HTML!<p>
    <script>
        alert('Hello world - from JavaScript!');
    </script>
    <p>Bye bye - from HTML!</p>
    <script>
        alert('Bye bye - from JavaScript!');
    </script>
</body>
</html>
```



You should be able to open your HTML page in a browser simply by double-clicking the document (normally you would get it from a Web server instead - we will get to that!). Take note of the order of execution - the first embedded script interrupts the rendering of the page and shows a message window (through the use of the `alert` function), and the rest of the page is not rendered until this window is closed. You can re-run your code by refreshing the page.

Directly embedding JavaScript in HTML pages quickly becomes very unwieldy due to the mixing of languages and project structure. Most of the time it makes more sense to have the JavaScript code in a separate file which is then included in the HTML page. The <script> tag then looks as below:

```
<script src="path/script.js"></script>
```

Move your JavaScript greeting(s) code to a separate script file and include it as above. We will primarily be working inside `.js` files rather than `.html` files today.

JavaScript is a full-fledged language with many of the features you are already familiar with, such as variables, loops, conditionals, as well as dynamic typing similar to Python. There are two basic ways to declare variables:

```
let some_number = 50;
const some_string = 'Hello world!';
```

You can request input from the user by using the `prompt` function. Try something like the code below:

```
const greeting = 'Hello world!';
const greeting_limit = 20;
let greet_count = prompt('How many greetings do you want?', 10);
// 10 is the (optional) default used if the user provides no value

if (greet_count > greeting_limit) {
    greet_count = greeting_limit;
}

for (let i = 0; i < greet_count; i++) {
    alert(greeting);
}
```

Single and double quotes are generally interchangeable in JavaScript. You can also use *backticks* (`), which allow you to format strings through the inclusion of variables inside ${ } blocks, as in the example that follows. You might recall a similar notation from bash!

```
const user_name = prompt('What is your name?');
const greeting = `Hello ${user_name}!`;
alert(greeting);
```

You definitely have some experience using Web applications, so you probably know that the sort of pop-up windows we have been using are not the typical way people interact with Web pages. Instead they type text into text boxes, press buttons, and click links, all directly on the page itself. JavaScript can interact with those HTML objects using the DOM. Add an empty paragraph with a unique id to your HTML page:

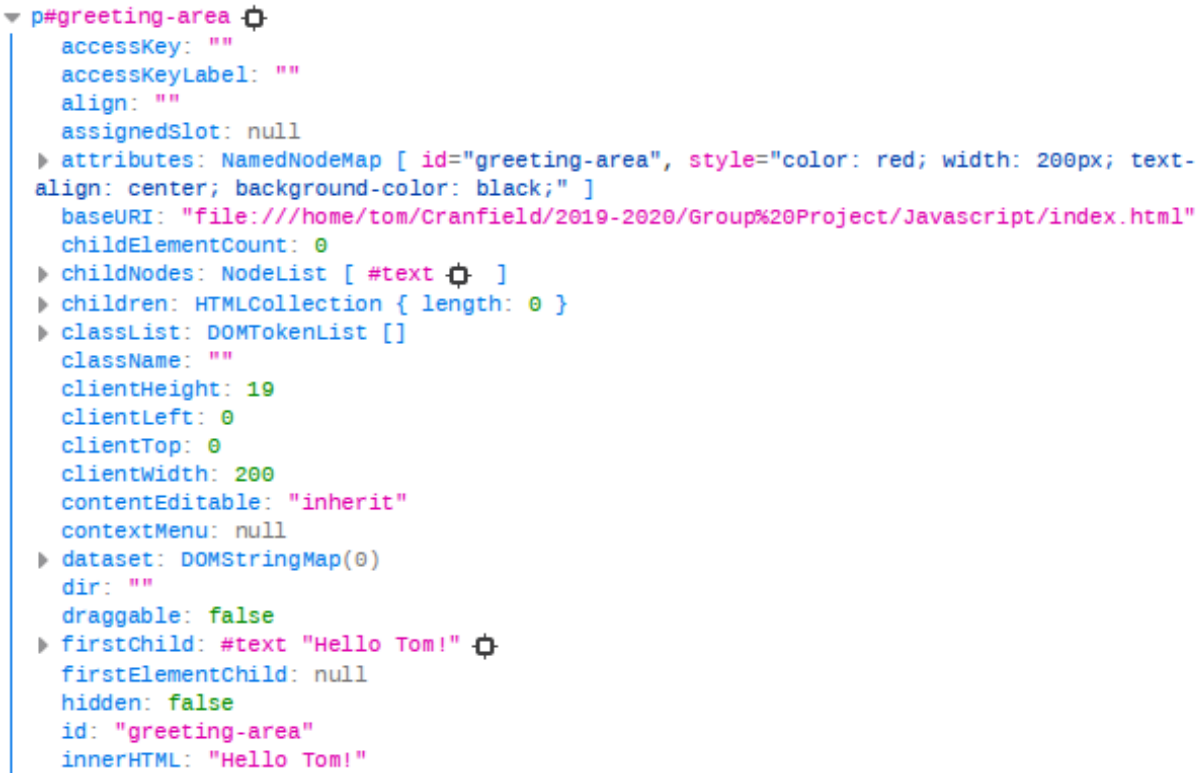
```
<p id="greeting-area"></p>
```

You can then retrieve a reference to the DOM element corresponding to the paragraph and manipulate it, for example by changing its contents or CSS styling:

```
const elem = document.getElementById('greeting-area');
elem.innerHTML = greeting;
elem.style.color = 'red';
elem.style.width = '200px';
elem.style['text-align'] = 'center';
elem.style['background-color'] = 'black';
```

The `document` object represents the root node of the DOM tree. Note that there is a whole family of functions other than `getElementById` which allow you to search for and traverse DOM tree nodes, such as `getElementsByClassName`, `getElementsByTagName` or `querySelector`. Some of those (as you can see by the plural "Elements" in the function names) may return arrays of matching elements rather than a single element. You can also use those on individual elements rather than `document` to find child nodes (i.e. nested tags), so for example `elem.getElementsByClassName('post')` will return an array of elements with the CSS class named "post" nested inside the `elem` node.

Especially for debugging purposes, it is often useful to have a "print" function more subtle than the `alert` we have been using so far. You can use the `console.log` function to print to a Web console, which can be viewed in most browsers through some type of "developer tools" option. For example in Firefox, the console can be viewed by pressing Ctrl+Shift+K, and on Chrome it can be viewed by pressing Ctrl+Shift+I. Take a moment to print (`console.log(elem)`) and investigate the `elem` object which we have been using:



You can do the same with any object we use to examine its structure. Make sure you always know what you are working with!

Instead of using `prompt` to get input from the user, we will now use a standard HTML input text box to get user input and a standard HTML button to trigger a

JavaScript function which will update our greeting. Add the following text to your HTML page:

```
Enter your name:
<input type="text" id="user-name-input" />
<input type="submit" value="Send" onclick="updateGreeting()" />
```

Inside your script file, you should then define the following JavaScript function, which is referred to by the HTML above. Note that the function has no parameters and returns no value, it merely interacts with the DOM, which should automatically update the rendered page.

```
function updateGreeting() {
    const input_elem = document.getElementById('user-name-input');
    const user_name = input_elem.value;
    const greeting = `Hello ${user_name}!`;
    const elem = document.getElementById('greeting-area');
    elem.innerHTML = greeting;
}
```

Try out your code!

The `onclick` attribute adds a *listener* to the button element. It is executed every time the button is pressed. This is one of the most popular ways event-driven programming is approached. There are other events you can listen for - try to add an `oninput` listener to the input textbox to trigger the greeting update function every time a user types in text! You can do this in HTML previously, or programmatically within your JS script:

```
const input_elem = document.getElementById('user-name-input');
input_elem.addEventListener('input', updateGreeting);
```

or

```
const input_elem = document.getElementById('user-name-input');
input_elem.oninput = updateGreeting;
```

Note that in both of these examples we provide the **function itself** rather than its result (i.e. `updateGreeting` instead of `updateGreeting()`). Functions are *first-class citizens* in JavaScript, meaning they can be assigned to variables, passed as arguments, and generally be treated the same way as other objects. You could even provide the function body directly to addEventListener without ever defining a named function like `updateGreeting`:

```
input_elem.addEventListener('input', function () {
    const input_elem = document.getElementById('user-name-input');
    const user_name = input_elem.value;
    const greeting = `Hello ${user_name}!`;
    const elem = document.getElementById('greeting-area');
    elem.innerHTML = greeting;
});
```

This is called an *anonymous function*, since it doesn't have a name.

Simple front-end message app

We will now try to build a simple “guestbook” application. Our Web page will display a list of stored messages and allow a user to add new ones.

Create a new project (or simply a new pair of HTML and JS files) and prepare an HTML container for our messages:

```
<div id="message-container"></div>
```

Then define the following array with two elements (both of which are objects enclosed by { curly braces }) in the JavaScript file:

```
const messages = [
    {
        author: 'Tom',
        message: 'This is the first message in our app!'
    },
    {
        author: 'Tom',
        message: 'This is another message!'
    }
];
```

As you can see the way objects (JavaScript's version of *dicts* or *HashMaps*) are defined is very similar to the JSON format mentioned in the lecture – the name stands for "JavaScript Object Notation" after all! An array is simply defined as a list of elements between square brackets. You can add extra elements to such an array using `push`:

```
messages.push({
    author: 'Fady',
    message: 'This is the third message!'
});
```

Note that we can add elements to messages even though it is `const`. This is because `const` merely stops us from *redefining* the variable (array or object), not from modifying its elements.

We can create new DOM elements by using the `document.createElement` function and add new children to a specific DOM node by using that node's `appendChild` function. By doing these within a loop iterating over our message array, we can populate our container:

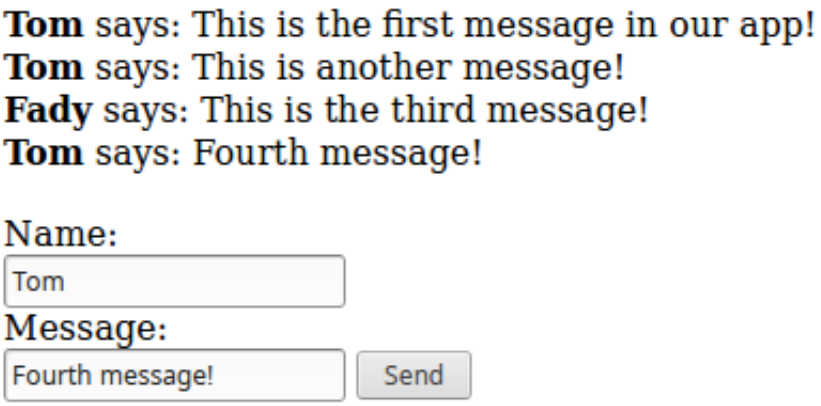
```
function displayMessages() {
    const container = document.getElementById('message-container');
    container.innerHTML = ''; // Clearing contents
    for (let i = 0; i < messages.length; i++) {
        const elem = document.createElement('DIV');
        const msg = `<b>${messages[i].author}</b> says:
${messages[i].message}`
        elem.innerHTML = msg;
        container.appendChild(elem);
    }
}
displayMessages();
```

Tom says: This is the first message in our app!
Tom says: This is another message!
Fady says: This is the third message!

We have wrapped our message display code into a function (and then executed it) so that we can call it again later to update the message display.

Now that we are able to display our message array, we need some way to add new messages. Add the following simple "form" (note that we are not actually using a HTML form, which could refresh the page and contact some external resource) to your HTML page:

```
<div>Name:</div>
<input type="text" id="user-name-input" />
<div>Message:</div>
<input type="text" id="message-input" />
<input type="submit" value="Send" onclick="sendMessage()" />
```



Try to implement the `sendMessage` function on your own! It should retrieve the user name and message from the input elements, add a new message to the message array, then update the container (we already have a function for the last part!).

Note that the messages are not kept across different browsers or even a page refresh - everything happens on the front-end, in the browser, without any storage method such as a database. In a proper application the messages would be fetched from a server and new messages would be sent to the server, for example through a REST interface. The server application would then store and retrieve the messages from a file or database. Don't worry – we will get there!

Extra task: Make your app more like a Facebook or Twitter feed by displaying only several of the latest messages, with the newest on top!

JavaScript on the server - Node.js

As mentioned in the introduction, these days JavaScript is not restricted to the browser. You can execute a JavaScript script from the command line using the **Node.js** environment, as well as create and manage Node.js projects, as well as install dependencies, by using the **npm** package manager. Both are available on Cranfield machines accessible via VMware Horizon, but they can also easily be installed on your own computer by following instructions from the following link: https://nodejs.org/en/download/.

To work with Node.js interactively (useful for testing small snippets of code), you can simply launch it from the command line like so:

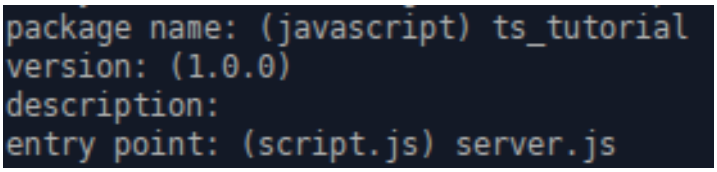
```
node
```

To run a pre-written script, which will be our main approach, you can do the following:

```
node script.js
```

Try that with some simple scripts, for example one printing "Hello World". Note that, as the code is not running in a browser, you will not have access to any browser-specific functionalities, such as the DOM or functions like `prompt` or `alert`. Without any libraries you are largely limited to `console.log` if you want to print text out.

In this way JavaScript can be used as a general-purpose scripting language similar to Python. It also has a package manager called **npm**, analogous to the PIP package manager which you have used with Python.



You can create a new npm project by typing in `npm init` in your project directory. This will create a `package.json` file for you, which will keep track of your dependencies. While setting up your project, set `server.js` (we will create this file in a moment) as our project entry point.

In the previous section of the tutorial we have been somewhat limited by needing to open our Web page in the browser manually, directly from the local hard drive. In actual practice, Web pages are served through some sort of Web server, which is accessed through an URL. We will now set up a simple HTTP server to serve our application using the **Express.js** framework, which is popularly used to, among other things, implement REST interfaces and servers.

Install Express by typing:

```
npm install express --save
```

Note that dependencies are added to package.json and installed into a new directory called `node_modules`. If you use git and create a git repository for your project, made sure `node_modules` is in `.gitignore`! You don't want to include all the dependencies in your project.

Create a `server.js` file and type the following:

```
const express = require('express');
const app = express();

const port = 3000;
const path_to_project = "/home/tom/Javascript_practical/dist";
app.use(express.static(path_to_project));

app.listen(port, function () {
    console.log(`Application deployed on port ${port}`);
});
```

Make sure to replace `path_to_project` with the actual path to your `index.html` and associated JS script files. The script loads the Express.js dependency, creates an Express server object called `app`, then sets it up to provide static content from a specified directory. Note that we do not need to specify the actual file names - "`index.html`" serves as the default entry point for Web pages, and the `src` attribute we use for our JS scripts has a relative path.

We then start an HTTP server by making it listen for requests on port 3000. The anonymous function we provide as the second argument to `app.listen` gets executed once the server is ready.

Start up your script and server by typing:

```
node server.js
```

If everything went well, you should now be able to access your app by opening your Web browser and typing in http://localhost:3000/ in the address bar!

Server-side persistence

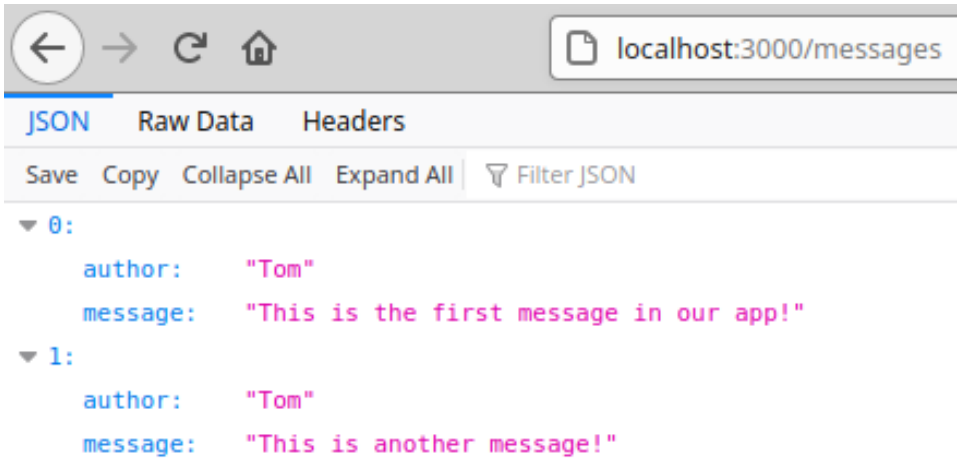
Now that we have a separate server application, we could try storing the messages on the server instead of in the front-end application. This will allow all users connecting to the application to share the same message feed. We will need to reorganize our project to make it possible – feel free to create a copy and work on it as a separate project.

Move the definition of our `messages` array from the front-end `script.js` file to our `server.js` file. Then, add the following GET request end-point to your Express `app` object:

```
app.get('/messages', function (request, response) {
```

```
    response.json(messages);
});
```

Re-start your server (i.e. interrupt and start `node server.js` again). It should now respond to GET requests at `/messages` by sending the messages array as a JSON response. You can test this out by typing http://localhost:3000/messages into your browser.



In order to make use of this on the front-end, our script has to make a GET request to the server. This can be done using the `XMLHttpRequest` class. Create the following function:

```
function updateMessages() {
    const http_req = new XMLHttpRequest();
    const url = 'http://localhost:3000/messages';
    http_req.open("GET", url);
    http_req.onload = function() {
        const messages = JSON.parse(http_req.responseText);
        displayMessages(messages);
    }
    http_req.send();
}
```

We will use this `updateMessages` function in place of the `displayMessages` function we used previously. The function makes GET request, processes the response in an anonymous callback function, and calls the `displayMessages` function, providing it with the retrieved messages. Modify your

`displayMessages` function to handle a `messages` array given as an argument instead of a locally stored variable, then try your Web application out – it should function as previously and display the messages stored on the server, although it will not allow you to post new ones.

To send messages from out front-end application to the server, we will need to add a POST request end-point to `server.js`:

```
app.post('/sendmessage', function (request, response) {
    messages.push(request.body);
    response.send(request.body); // echo the request
});
```

This will push the JSON body of a POST request to our messages array, and send the same request back as a response (the latter step is optional – we could send back something else, like a confirmation or error message). In order to correctly parse incoming JSON bodies, we also need to install the body-parser package:

```
npm install body-parser --save
```

And add it as middleware to our Express app:

```
const bodyParser = require('body-parser');
app.use(bodyParser.json());
```

We can now send new messages to the server from our front-end application. Modify your `sendMessage` function to include the code below:

```
const message = { author: user, message: msg };
const http_req = new XMLHttpRequest();
const url = 'http://localhost:3000/sendmessage';
http_req.open("POST", url);
const content_type = 'application/json;charset=UTF-8';
http_req.setRequestHeader('Content-Type', content_type);
http_req.send(JSON.stringify(message));

http_req.onload = function() {
     // Update message display once a response is received
```

```
        updateMessages();
    }
```

Congratulations! Your front-end application now sends and receives data from a server using a POST and GET request, respectively. The message array should now be shared between different users of the app – test this out by opening it in different browser tabs (or different browsers entirely).

Note that while the messages will be stored as long as the server is running, restarting it will remove them. To achieve true persistence, you would have to store them on a file or in a database. We will be using databases for this later in the week, but if you have **extra** time you could try modifying our server to make it save the `messages` array into a file whenever a new message is received, and retrieve it whenever the server is launched. This could be achieved using the **fs** (file system) module (https://nodejs.org/api/fs.html).

Web frameworks

We have been using "vanilla", or plain JavaScript throughout this introductory tutorial to become familiar with the basics of the language. In practice however, JavaScript front-end applications are usually developed using some sort of *framework*. Frameworks are essentially large libraries or sets of libraries, each with their own approach to solving common front-end problems, organizing code, binding values between JavaScript objects and HTML pages, and so on.

Each of the major front-end frameworks differs significantly in terms of project organization and design philosophy. Three of the most popular ones today are **React** (https://reactjs.org/), **Angular** (https://angular.io/), and **Vue** (https://vuejs.org/). Feel free to try them out when you have the time and pick the one you like for your own projects!

From your perspective as Applied Bioinformatics students: **Vue** is the simplest, **React** is the most popular and has the best community support, and **Angular** is the most similar to the enterprise, object-oriented approach you have learned in your Java work – quite complicated, but good for large projects.

We will not be delving deeper into front-end development frameworks in the Data Integration module, as our interest lies more with the back-end, but there will be an opportunity for you to learn a bit more about the frameworks at the start of the Group Project.