Part 0 – General info & Fundamentals of Web Apps

# A. General info

## Course material

* Meant to be read one part at a time and in order
* The material contains exercises, which builds on the previous parts
* Exercises can and should be done as encountered in the material, but it can be beneficial to read all material first
* It is best to follow the code along while making small modifications independently
* The code of the example applications for each step can be found on GitHub

## Taking the course

* 14 parts, the first is part0
* One part corresponds loosely to 15-20h on average
* Parts 1-4 you are expected to do *at least* all of the exercises that are not marked with an asterisk \*
* Asterisks count toward the final grade, but skipping them does not prevent you from doing the compulsory exercises in the next parts.
* Part 5 does not have asterisk marked exercises since there is no similar dependency on previous parts
* Exercise completion time statistics can be found via the [submission system.](https://studies.cs.helsinki.fi/stats/courses/fullstackopen)
* Parts 0-5 form the core course – **Full Stack Web Development (5cr, CSM141081)**
* Credits and grades are based on the total number of submitted exercises for parts 0-7 as below

**A picture containing text, black, screen

Description automatically generated**

* Once you have completed enough exercises for a passing grade, you can download the course certificate from the [submission system.](https://studies.cs.helsinki.fi/stats/courses/fullstackopen)
* To receive university credits you must complete the course exam
  + This does not count to the final grade, but you must pass it. More info [here](https://fullstackopen.com/en/part0/general_info#the-course-exam).
* More info about extra xredits from part 6, 7, 8, 9, 10, 11, 12 and 13 can be found in part0 general info

### Course plan

* I would like to complete the course in the timespan of Week 4 – Week 17.
* That would give me 14 weeks
  + - If one part takes 2 weeks, I would get at least 7 parts completed
    - If one part takes 3 weeks I would get almost 5 parts done
    - Therefore a good goal is to complete one part every 3 weeks, not counting part 0

## Git Instructions

* The course exercises are submitted to GitHub, so Git must be installed and you should know how to use it.
* For instructions, see [Git and GitHub tutorial for beginners.](https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners)
* Vscode is recommended, dont code with Nano, Notepad or Gedit or NetBeans.
* Install Node.js
  + The material is done with 16.13.2 so don’t install anything older than that
  + See [Node.js installation instructions](https://nodejs.org/en/download/package-manager/)
    - I installed it here: C:\Program Files\nodejs\
    - With the basic installer from the webpage
  + Node package manager npm will be automatically installed with Node.js. We will be actively using npm throughout the course.
  + Node also comes with npx, which we'll need a few times.

# B. Fundamentals of Web Apps

* Let’s go through some principles of web development by examining an example application at <https://studies.cs.helsinki.fi/exampleapp>
* The first rule of Web Development: Always keep the Developer Console open in the web browser.
  + On MacOS F12 or option-cmd-i
  + On windows or linunx F12 or ctr-shift-

## HTTP GET

* The server and the web browser communicate with eachother using the HTTP protocol.
* The *Network* tab shows how the browser and the server communicate
* When eg. Reloading page with F5 the *Network tab* will show that two events (three in this case) have happened
  + The browser fetched the contents of the page
  + The browser downloaded the image *kuva.png*
  + Clicking an event shows more information, clicking headers shows
  + The general segment shows that the browser requested the address: *Request URL:* [*https://studies.cs.helsinki.fi/exampleapp/*](https://studies.cs.helsinki.fi/exampleapp/)
  + Through the [GET](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET) method, and that the request was successful, because the server response had the [Status code 200](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)
* There are several [headers](https://en.wikipedia.org/wiki/List_of_HTTP_header_fields) to each request and server response
  + The *Response headers* on top tell us eg. The size of the response in bytes and the exact time of the response
  + Another important field of this header *Content-type* tells us that the response is a text file in utf-8 format and the contents of which have been formatted with HTML
  + This way the browser knows the response to be a regular HTML page and to render it to the browser ”like a web page”
  + The response tab shows the response data, a regular HTML page. The *body* section determines the structure of the page rendered to the screen
  + The page contains a div element, which in turn contains a heading, a link to the page notes and an img tag, and displays the number of notes created
  + Because of the img tag, the browser does a second HTTP request to fetch the image kuva.png from the server.

## Traditional web apps

* This homepage works like a traditional web application. When entering the page the browser fetches the HTML document detailing the structure and the textual content of the page from the server.
* The server has formed this document somehow. The document can be a static text file saved into the servers directory. The server can also form the HTML documents dynamically according to the application code, using, eg. Data from a database.
* The HTML code of the example application has been formed dynamically because it contains information on the number of created notes.
* The HTML code of the homepage on the server end is as follows:

*Text

Description automatically generated*

* Oldschool PHP programmers would write HTML amid the code, which today is not smart practice.
* In traditional web apps the browser is “dumb”. It only fetches HTML data from the server, and all application logic is on the server.
* A server can be created using Java Spring, Python Flask or Ruby on Rails to name just a few examples.
* The example uses Express library with the Node.js. This course will use Node.js and Express to create web servers.

Text

Description automatically generated

## Running application logic in the browser

* The mechanism of invoking event handlers is very common in JS. Event handler functions are called callback functions.
* The application code does not invoke the functions itself,. But the runtime environment – the browser, invokes the function at an appropriate time when the event has occurred
* Document Object Model or DOM
  + We can think of HTML pages as implicit tree structures
  + The DOM is an Application Programming Interface (API) that enables programmatic modification of the element trees corresponding to web pages
  + The JS code introduced used the DOM-API to add a list of notes to the page
  + The topmost node of the DOM tree of an HTML document is called the document object.
  + We can perform various operations on a webpage using the DOM-API.
  + You can access the document object by typing document into the Console tab

## Submitting a form request 🡪 asking the browser to do something to the database/server

* Submitting a form request via the box creates a form submit event.
* This event is a HTTP POST request to the server address new\_note
* The server responds with HTTP status code 302, which is a URL redirect, with which the server asks the browser to do a new HTTP GET request to the address defined in the headers *Location –* the address *notes*
* So the browser reloads the Notes page
* The reload then causes three more HTTP requests: fetching the style sheet (main.css), the JS code (main.js) and the raw data of the notes (data.json)
* The network tab also shows the data submitted with the form
  + On newer chrome I should see a Form Data header… but I cannot see it for some reason
  + It seems like this is now in the “payload” tab exactly
* The form tag has the attributes action and method in the elements tab which define that submitting the form is done as an HTTP POST request to the address *new\_note*
* The code on the server responsible for the POST request is quite simple (NB: This code is on the server, and not on the JS code fetched by the browser)

**Graphical user interface, text

Description automatically generated**

* The server does not save new notes to a database, so new notes disappear when the server is restarted

## AJAX

* The notes page of the application follows an earl-nineties style of web dev and uses “Ajax”. As such its on the crest of the wave of early 2000s web technology.
* AJAX (Asynchronous JavaScript And XML) is a term introduced in February 2005 on the back of advancements in browser technology to describe a new revolutionary approach that enabled the fetching of content to web pages using JavaScript included within the HTML, without the need to rerender the page
* Before the AJAX era all web pages worked like the traditional web application we saw earlier in this chapter. All of the data shown on the page was fetched with the HTML code generated by the server.
* The Notes page uses AJAX to fetch the notes data. Submitting the form still uses the traditional mechanism of submitting web forms.
* The application URLs reflect the old, carefree times. JSON data is fetched from the URL …/data.json and new notes are sent to the URL …/new\_note
* Nowadays URLs like these would not be considered acceptable, as they don’t follow the generally acknowledged conventions of [RESTful](https://en.wikipedia.org/wiki/Representational_state_transfer#Applied_to_web_services) APIs.
* The thing termed AJAX is now so commonplace that it’s taken for granted. The term has faded into oblivion and the new generation has not even heard of it

## Single page app

* In our example app the home page works like a traditional webpage:
  + All of the logic is on the serverside
  + The browser only renders the HTML as instructed
* The notes page gives some of the responsibility - generating the HTML code for existing notes - to the browser.
* The browser tackles this task by executing the HTML code for existing notes, to the browser.
* The code fetches the notes from the server as JSON data and adds HTML elements for displaying the notes to the page using the DOM-API
* In recent years the [Single-page application](https://en.wikipedia.org/wiki/Single-page_application) (SPA) style of creating web applications has emerged.
* SPA style websites don’t fetch all of their pages separately from the server like our soample application does, but instead comprise only one HTML page fetched from the server
  + The contents of which are manipulated with JS that executes in the browser
* The Notes page of our application bears some resemblance to SPA style apps, but it’s not quite there yet.
  + Even though the logic for rendering notes is run on the browser, the page still uses the traditional way of adding new notes.
  + The data is sent to the server via the forms submit, and the server instructs the browser to reload the Notes page with a *redirect*
* A SPA version of the example application can be found at: <https://studies.cs.helsinki.fi/exampleapp/spa>
  + At first glacnce the application looks exactly the same
  + The HTML code is almost identical, but the JS file is different (spa.js) and there is a small change in how the form-tag is defined
  + The form has no action or method attributes to define how and where to send the input data.
  + When creating a new note and sending it and then viewing the network tab we can see that only one request is sent by the browser to the server. The whole page is also not reloaded
  + The POST request to the address *new note spa* contains the new note as JSON data containing both the content of the note (*content)* and the timestamp (*date*).
  + The content-type field in the Response headers tells the server that the included data is represented in JSON format (application/JSON)
  + Without this header, the server would not know how to correctly parse the data
  + The server responds with the status code 201 created. This time the server does not ask for a redirect, the browser stays on the same page and it sends no further HTTP requests
* The SPA version of the app does not traditionally send the form data, but instead uses the JS code it fetched from the server.