What is the course?

* An intro to modern web application development with JavaScript, through building single page applications with ReactJS, using REST APIs built with Node.JS (and GraphQL, modern alternative to REST APIs).
* The course also covers testing, configuration and environment management, using MongoDB for storing application data
* Content is the same as the Full stack course at the Department of Computer Science at the University of Helsinki
* Pre-requisites:
  + Good programming skills
  + Basic knowledge of web programming & databases
  + Basic Git version-control system
  + Perseverance and ability for independent problem solving / information seeking

Content:

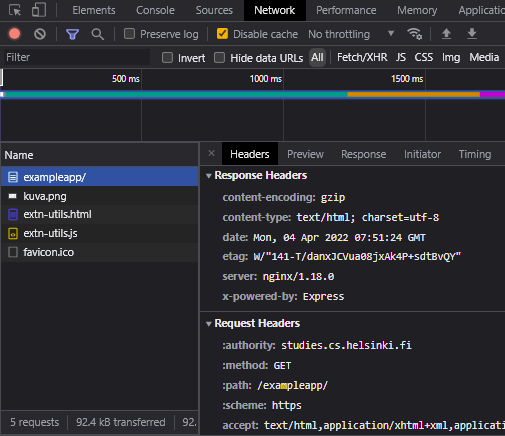
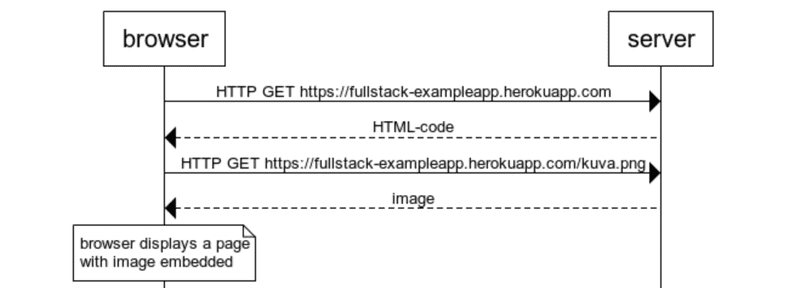
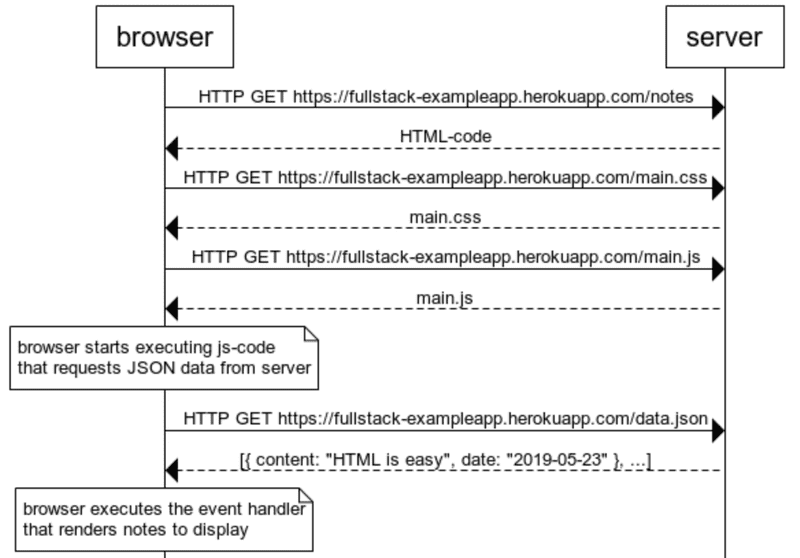
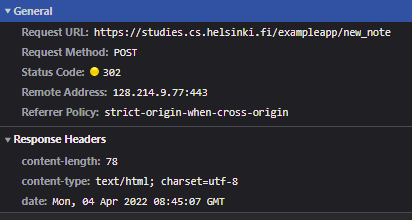
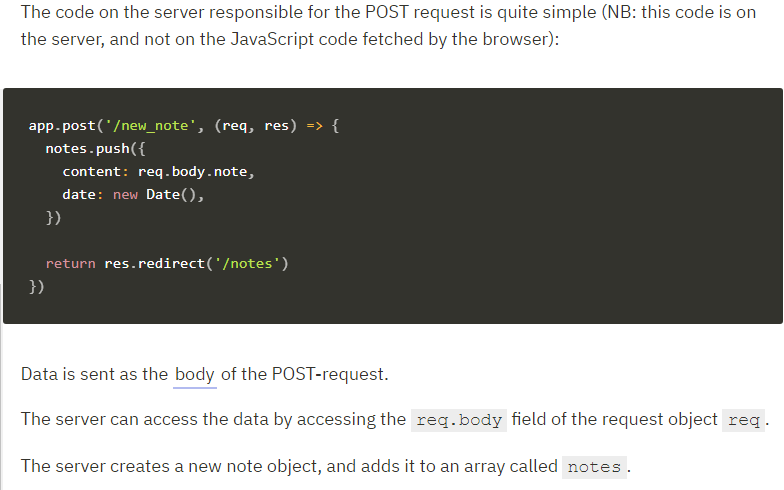
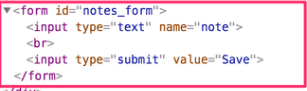
1. Fundamentals of Web Apps
2. Introduction to React
3. Communicating with server
4. Programming a server with NodeJS and Express
5. Testing Express server, user administration
6. Testing React apps
7. State management with Redux
8. React router, custom hooks, styling apps with CSS and webpack
9. GraphQL
10. Typescript
11. React Native
12. CI/CD
13. Containers
14. Using relational databases

# Fundamentals of web apps

Setting up repo via Git / Github

* Commands used
  + Cd / dir
* Git add / git status – checking which files are tracked, adding files for Git to track
* Git commit -m
* Create new branch - git checkout -b <my branch name>.
* Git branch – check for branches
* Sync with Github
  + git remote add origin <https://github.com/JulianGoh/fullStackOpen-ReactCourse.git>
  + git branch -M main
  + git push -u origin main

HTTP GET

* Using the Chrome Developer console
  + Request / response infrastructure – browser first sends a HTTP GET request to the server to fetch the html code/png image then the browser renders the html and the image to the screen
  + Request / response “headers”
  + 
  + Sequence of events
  + 
  + The HTML looks like this:
* const getFrontPageHtml = (noteCount) => {
* return(`
* <!DOCTYPE html>
* <html>
* <head>
* </head>
* <body>
* <div class='container'>
* <h1>Full stack example app</h1>
* <p>number of notes created ${noteCount}</p>
* <a href='/notes'>notes</a>
* <img src='kuva.png' width='200' />
* </div>
* </body>
* </html>
* `)
* }
* app.get('/', (req, res) => {
* const page = getFrontPageHtml(notes.length)
* res.send(page)
* })
* Traditional web applications
  + In traditional web applications, the browser is ‘dumb’ – only fetching HTML data from the server, and all application logic resides on the server
  + Event handlers and event handlers’ functions
* xhttp.onreadystatechange = function() {
* if (this.readyState == 4 && this.status == 200) {
* // code that takes care of the server response
* }
* }
  + - Onreadystatechange – event handler
    - When the state of the object changes, the browser calls the event handler function – checking that readyState == 4 && this.status === 200
  + Event handler functions are also called callback functions
    - First the event handler has to be established, before being able to invoke them
    - **It is the runtime environment, the browser, that invokes the event handler function, when the event has triggered**
* Document Object Model (DOM)
  + We can think of HTML pages as implicit tree structures
* html
* head
* link
* script
* body
* div
* h1
* div
* ul
* li
* li
* li
* form
* input
* input
  + The DOM is an API which allows for programmatic modification of the *element trees* corresponding to web pages
  + The topmost node of the DOM tree of a HTML document – ‘document’
* CSS – class selectors, classes are attributes that can be added to HTML elements
* Loading a page containing JavaScript
  + 
  + Browser fetches the HTML code, defining the content and structure of the page, from the server via HTTP GET
  + Links in the browser allows browser to also fetch .css and .js
  + Browser executes the JS code that requests JSON data from server, which then returns the data.json information
  + Once the data is fetched, browser executes the event handler that renders the notes to display
* Forms & HTTP POST
  + 
  + The HTTP POST request returns a 302 status code (URL redirect), where the server asks the browser to do a HTTP GET for a different address
  + 
* AJAX – 90s style web development -> Asynchronous-Javascript-And-XML released in 2005
* Single Page App (SPA)
  + The example app behaves like a traditional web-page – all of the logic is on the server, browser only renders HTML as instructed
  + The nodes page – browser executes a JS file which grabs data from a server and then returns the data to the browser to display
  + SPAs comprise of one HTML page fetched from server, the contents of which are manipulated with JS that executes in the browser
  + 
  + Vs
  + 
  + Removing the form action away from the HTML & avoiding the need to redirect - Relying on JS to handle the server side POST
* var form = document.getElementById('notes\_form')
* form.onsubmit = function(e) {
* e.preventDefault()
* var note = {
* content: e.target.elements[0].value,
* date: new Date(),
* }
* notes.push(note)
* e.target.elements[0].value = ''
* redrawNotes()
* sendToServer(note)
* }

var sendToServer = function(note) {

var xhttpForPost = new XMLHttpRequest()

// ...

xhttpForPost.open('POST', '/new\_note\_spa', true)

xhttpForPost.setRequestHeader(

'Content-type', 'application/json'

)

xhttpForPost.send(JSON.stringify(note))

}

* JavaScript libraries
  + Vanilla JS – using only the DOM-API and JS to manipulate page structures
  + jQuery – easier tools to use than the DOM-API
    - What made it successful? Cross-browser compatibility
    - jQuery was succeeded by BackboneJS then AngularJS then Facebook’s React and Redux then VueJS
* ‘Full stack web development’
  + The buzzword of ‘full stack’
    - All web applications require layers – the browser (user end) and the server (bottom). Below the server is the database layer. Therefore we can think of the architecture of web design as a kind of stack of layers
* JavaScript fatigue
  + Full stack web dev. is challenging. Things happening in many places at once, which makes debugging harder – the async way its runtime environments work brings its own challenges
  + Communicating on the web demands HTTP understanding
  + Also be able to manage databases, server administration and configuration
  + JS fatigue has become associated with the pace of change within the ecosystem – tools, libraries, the language itself