Back-end and frontend exams,

Addition: Full-stack open reading list

by Juhani Välimäki

*Listing learning points or section contents, but **also** in some points adding annotations or additional hints for understanding. Lot of what we already have in backend and frontend exams. But also some completely new topics.*

This list is more for tracking the reading than a real list of what can be formulated as exam question. There are less exam question topics than points in these lists.

The list follows the Full-stack open course chronologically. Backend topics with blue, frontend with green. Of course sometimes a mix. Possible additions to the ECMAScript exam in some color later?

exercises: Skip if need to advance fast. Read if need quite fast pace. Do them if have lot of time.

Part 0b. Fundamentals of Web apps

- Mostly recapping earlier course topics in this section. Something new for most of you too!
- Go through this part somewhat fast
- In Full-stack open you can, but don't have to, visit the offered extra info links. If do so, just spend even just 30 seconds – max few minutes or so there in those *side track* links. Focus!
- It is good to recap at least these concepts
    - http request, GET method with possible query params in URL
    - looking into http request headers
    - DOM manipulation with JavaScript running in browser (JS defined in *DOM API*)
    - POST method, request with headers and the body
    - AJAX, https://en.wikipedia.org/wiki/XMLHttpRequest, e.g. the xhr object given to you by the DOM API JavaScript implementation of the browser. Or using via some AJAX library.
    - SPA idea (Frontend)
    - JavaScript libraries for Web UI and/or datamodels: jQuery, BackboneJS, AngularJS, React, Redux, VueJS
- "JavaScript fatigue" is true. Many new tutorials of the new versions of the libraries (e.g. React, Redux) utilize _all_ new features that appear in ECMAScript standard. Thus, people need to learn both new version of the library and new version of ECMAScript constantly.
- *exercises*. Do the exercises if have a lot of time. Otherwise you can just learn the theory fast first.
- *if you do the fullstack open, parts 0*-5 with the tasks (E.g. during the intensive weeks) you can get ECTS to your study transcript by bringing the certificate to academic advisor.

Part 1a. Introduction to React

- npx **both** temporarily downloads and runs e.g. the **create-react-app** tool
- what kind of React project template is created by create-react-app tool (+npm scripts)?
- learn/recap the function-based React, and the React hooks
- **JSX**, XML-like but not really XML, e.g. this would not be valid XML: `<xyz def={abc} />` (attribute values should be enclosed in " " or ' ' in well-formed XML)
- props as React component function parameter
- start the JSX code (~markup)
    - on same line as return statement starts,
    - OR even better: wrap the JSX inside the parenthesis: return (…….) OR
    - ( ) => (…….);
- React function should return
    - one root component,
    - OR an [ ] array of (parallel) components
    - OR a fragment, empty/non-rendered mother element with other elements inside
- **props** to pass data and event-handlers to child components. Discussed more later.
- build app in baby steps, monitor all the many console etc. windows you can have open
- <ReactComponents /> must start with Caps, html elements with small letter: <h1>
- *exercises*

Part 1b. JavaScript

- (Not listing here all the (basic) new ES features we have already learned. Only challenging ones or ones with particular interests)
- ECMAScript202**1** aca. ES1**2**. (E.g. ECMAScript201**5** was ES**6**)
- babel – transpiler library. From new ECMAScript syntax to older versions of ES/JavaScript that the used (browser's or e.g. backend server's) JavaScript engine supports
- e.g. Node.js (which uses the V8 JavaScript engine) supports the modern ES more and more also without a transpiler: https://nodejs.org/en/docs/es6/
- const t = [1, -1, 3]; const t2 = t.**concat**(5); // not changing original array, returns a new array object
- The concept of *destructuring assignment*
- JS object literals are not JSON, even if they look almost similar. JSON is "text" but object literals are JavaScript code way to create objects.
- referring to the object members with 'indexer' brackets: person1['first name'] = "Joe";
  - works even for illegal member names, like having spaces or starting with digits
- object-oriented JavaScript with inheritance **was** used in old React, now that is forgotten.
  - Nowadays we use function and Hooks -way React to define React components
- arrow function specialties, nothing new for us
- *exercises*

- old stuff: arrow function doesn't bind the keyword 'this' to itself (to the arrow function object) but refers to outer lexical context. That is, to the React component object, which is handy and needed e.g. in component's event-handlers.
- method = function that has been attached to an object. When run, the keyword 'this' refers to the object if you have used arrow function notation => please use arrow functions
- (funny fact: if you take the function=method object out of the object, it loses the 'this'-reference and kind of becomes a non-method function)
  const `referenceToGreetFunction = object1.greet`
- setting up timers in JavaScript is easy:
  - `setTimeout(<arrowFunctionToBeCalled>, 1000)  one shot after 1s`
  - `setInterval(<arrowFunctionToBeCalled>, 5000) periodically run/fired each 5 seconds`
- multiple timers are independent of each other
- again using 'class' in JavaScript can be skipped, we rarely use it nowadays
- Maybe look at this fast: https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

Part 1c. Component state, event handlers

- component **helper functions** defined inside **React component functions** have access to e.g. props of that React component function/ = that component
- destructing the props with three different ways (so that it's not too simple 😊)

```
    o  const name = props.name;              // 1.way needs two lines
    o  const age = props.age;                // 1.way needs two lines

    o  const { name, age } = props           // 2.way, one-liner


    o  ({ name, age }) => {                  // 3.way, in paramter list
   // looks like an object created, but no. Two vars created and the
props (that would be component functions parameter, we don't see it here)
is destructured to those vars.
```

- relax and try to see how the three ways above lead to almost same (the last one is almost same even if two vars are created.
- why last one leads to vars and not consts? It happens in the parameter list of the React function. Like you know for the variables introduced in parameter list (=parameters) we cannot define var/let/const at all. And the parameters become function-scoped vars automatically.
- useState -hook => Stateful component the modern way (Forget the old object-oriented React and setState)
- useState is used to map these: 1. one state value holder and 2. its setter function and 3. provide the initial value
- don't be fooled by the state value holder being const - React will handle the actual running of the components behind the scenes, we just attach hooks to certain events
- passing event-handler arrow function objects to components in their props (You might remember, value passing only in JavaScript, here value passed is a function object that receivers can then call)
- if you need e.g. button click to call function foo(), do not call there immediately foo(), but create an arrow function which would, if/when later called, call the foo() inside its function body. = you are not doing something yet, you are giving the system chance to act on some event later, right?
- passing data (e.g. pieces of parent state, each item of some collection or so) to child components via props
- the concept of different kind of React components. Some e.g. only show data and provide buttons for doing something to just that data item (Presentational components). Some are parent components that have several child components (Container components). Those often fetch all the needed data and pass pieces of it to the presentational components.
- (re-)rendering happens at least in these cases: 1. state changes, 2. props change, (((~~3. render method called, 4. component object constructucted, these are mainly obsolete stuff~~))). What are the other real cases in modern function + hooks way, if there are more?
- we want to have the React components follow SRP (*the single-responsibility principle*). Doing just one thing. And if not, then refactoring them, e.g. splitting it potentially to sub-components. Sub-components defined outside of the parent component, so that you can reuse them possibly in another view too.

Part 1d. A more complex state, debugging React apps

- several pieces forming the React component **state** OR one, more complex object forming the state
- initial value kind of gives the data format of the piece of the state, and the methods that update or read the state need to follow the same format, e.g. keep it as an array or as certain object, with same member names. At least I do easily mistakes with React/Redux state object structures. Usually something put somewhere else on wrong structure level.
- (React dev tools introduced later helps in checking the state contents?)
- methods updating the state might **first** utilize the spread notation ... to take a copy of the current=old state as the basis, and **then** change the wanted parts in it.
- we will thus create a new object that will replace the old state object. We cannot mutate the old state object. (_his was the same rule in the traditional class-based React)
- array.concat (from earlier) will be handy when the model is an array. As it will not mutate original array, but will return a totally new array to be used as the state value. (unlike e.g. array.push() that would mutate the original object)
- in conditional rendering the authors have used if-(1st)return-(2nd)return -structure (two returns in row). If true, return something. If that happened the method will end. Otherwise return what the second return returns. You like that or not? To me if-else if-else would be more elegant. A bit nerdy solution this "shortcut return". Even the block for the if is left out. Well it's a JSX+JS combo so in some cases needed/ok.
- Old React, read, but **skip mostly**
- console.log('props value is'**,** props) // Other way to do console.log. No concatenation. For better debugging messages, try this out. Try also console.dir(someObject)
- React developer tools
- rules of Hooks we have also in the other Frontend exam reading list, but good to read also here
- Once again, the idea of passing a function object (defined with arrow function syntax) to the onClick, not executing code, but passing executable code for later execution, in case of …
- all in all, read the code examples here, and ask if don't understand everything out of it.
- in addition to the **reuse reasons**, components should be defined outside of other components so that the system can treat and **optimize** them as separate, but together orchestrated components, mosaic pieces.
- Useful reading for React, two links to learning sites. Certainly good, now just think do you want to advance fast to tasks/project? Or make a deeper dive into React? Up to you and depends on your situation
- *exercises*. Nice recap of the topics. Recommended **if** not in hurry to next chapter.

Part 2a. Rendering a collection, modules

- About the debugging and debugging tools again. Never emphasized too much!
- (Snippets maybe come handier in daily work than in learning. I like to type all myself with the thorough thinking while learning)
- Higher order functions (functions that are passed functions, functions that return functions to be executed by some other code, map/foreach/filter/find/reduce. Interesting topics, we have seen a lot of that topic, but not all)
- (again event-handler arrow functions returned from another arrow function that is passed some needed id:s or other configuration data)
- (BTW. All in all, you might want to use only arrow function syntax with modern React! Then you don't need to think where are the cases where traditional function will not work because of 'this', the self-reference)
- Rendering collections involves: 1. outer mother element (finally e.g. ol, ul, table, div), 2. map/foreach/etc…looping creating child elements (finally e.g. li, tr, div) **and** 3. the unique attribute 'key' to (each) child element
- Nowadays, when a React component is allowed to return an array of React components/elements, the map-function is the most used.
- Never use the array index for the key values in real cases. Use instead the UID or id (primary key, alternate key etc.) that the item has (/or will have) in database. Or an encrypted practically unique hash based on that id. Those 'keys' work.
    - whereas the array index gets ruined when an item in the collection is (re)moved.
- Visiting prop destructing again, React component refactoring, and splitting all views into single-responsibility principle components with child components further inside.
- And again, remember not to define Component inside the mother component's code.
- How does the root React component (Start of the SPA application) get 'injected' to the web page = to the DOM? What libraries and templates, files and components are involved? Just study fast:
    - index.js, App.js, index.html, react-dom, react
- node modules referred to without the folder path, e.g. 'react-dom', own react components and other JS files with the folder path in front './and_so_on'
- *exercises*

Part 2b. Forms

- `event.preventDefault()` used to stop the normal behavior of HTML forms, form submission to the backend etc. We want to stay in the same SPA and let the SPA do the routing of the views.
- And React code to handle the "submit" behavior too, with its own AJAX parts etc.
- The 'event' refers to the Web browser's GUI event objects. Have information e.g. about event target, how long was some button pressed. Etc. etc. Think e.g. all things related to what you can do with mouse in UI, click, drag, how long hold. Which exact pixel was clicked etc. Event types might record different set of info? E.g. (on)click vs (on)mousedrag.
- `onSubmit={addNote} //`When user presses the submit, what event-handler function should be called (addNote). Note this is not a function call, but binding a function object to the event, to be called later if needed.
- You can create 'controlled components' having own mapping between html input fields and the state of the component. When value in input field value changes, the change event-handlers also update the component's state, and thus also update any components showing the state.
- Here that model used where in the state there are all the items, plus the 'temp' for the just in-read input value(s) **before** they/it also go/goes to the all items -collection, to the more stable part of the state and temp cleared/ignored.
- nice example, but need to read the final/full version below
- ternary operator (conditional operator) `a ? b : c` used in conditional rendering in JS or JSX code
    - unary operator NOT: `!doorClosed` // 1 operand
    - binary operator AND: `doorClosed && lightsOff` // 2 operands
    - ternary operator: `doorClosed ? callJanitor() : goInside()` // 3 operands
- always use === in JavaScript, unless you know some rare special case very well, and you have tested all possible probs out.
- quite nice example continues. Not a bad idea to write all of it to a pristine create-react-app project template just for practicing? If extra time.
- *exercises*

Part 2c. Getting data from server

- Using tool json-server in these first examples (Later with Node.js server & e.g. Express)
    - fast output API generated based on the JSON file in the folder. Handy in some cases!
- (possibly JSONView plugin needed for Browser if the browser doesn't show the JSON response data nicely)
- use AJAX via some promises-based library, e.g. Axios, no need to learn those old *directly-using-the-DOM-api-ajax-objects* -ways used here
- JavaScript engines are single-threaded, even the one behind the Node.js server. (But Node.js simulates multi-threading as well as it can, and as well as your code allows. Browser JavaScript on the other hand might be more eager to hang or become non-responsive.)
- Web workers are for running multiple threads in Browser JavaScript. You might see one in the create-react-app
- npm, the **n**ode **p**ackage/module **m**anager, that should not be called node package manager…
- npm used also to start / update/ test / build / clean the project
- package.json file you don't want to lose.
    - package.json is the place where the dependencies and some other tooling/scripting are defined, primary source of that info/configuration. Should go to git repo.
    - package-lock.json etc. on the other hand can be removed and generated again.
- development dependency = only included in the development version of the product, not to the build version!
- npm run server (read this script from the package.json and understand how it works)
- like often on this course site the last version of Axios usage starts to be what we would use. (They kind of show the amateur versions first and refactor it to be better. Always read the last one)
- Even in the last version the request object could be maybe configured first and then sent in separate step. (URL, parameters, body data, etc.)
- useEffect -hooks // do something else too, than the React component state/props <-> render cycle. = 'side effects'. E.g. run the AJAX call to backend.
- the nice example starts to expand, but of course in real create-react-app projects we would have a lot of folders and files. E.g. one folder for views = "Pages", one for SPA routing files, one for other components than whole views, some folders for style files…
- the create-react-app project is, by default, just a development environment with e.g. auto-refresh monitoring and running new code after changes you make
- the real build version of the project does **not** run dynamic code in Node.js server, and does**n**'t have all of your folders and files separately.
- The built version runs only in the browser JavaScript environment after browser has downloaded the few generated and bundled/packed HTML and JS and CSS files. Just running then JavaScript in browser and of course also making AJAX calls to backend.

Part 2d. Altering data in server

- json-server tool at least claims to be really good for creating fast mockup REST API server, before getting the real backend created. Frontend developers can create their frontend parts for testing and development sooner.
- just remember to define the backend server IP address / server URL or any such changing configuration only once and only in one git-ignored file = easy to change later and details not going to the GitHub etc. remote git repo
- (even better would be reading that from environment variables = not part of the tested nor GitHub submitted project at all)
- REST is explained in other materials. But read these texts here and remember e.g. statelessness of each call. Not related to previous call. All calls=requests must include all info needed to serve that call.
- POST request with body = data going to the backend
- browser tools for checking the requests and responses are important.
- so are also the PostMan or e.g. VS Code REST Client request test senders, saving few successful and purposefully unsuccessful test case requests for future repetition of the tests.
- tests could be saved in a separate second repository OR in the same repo (if you can define which parts of the demo are monitored for code changes and which ones are not. As we do want to develop more and more, better and better test cases, but also differentiate between real code changing or not)
- (((mono repo idea would kind of solve that though, but not without complications)))
- The 'bang' operator, or the exclamation point ! is like Trump, turns true to false, and false to true.
- The *shallow copy* we have studied is used here too. In this example shallow copy is enough, as the Note objects do not refer to further objects, they do not contain references values.
- (rest of the example is actually based on already learned skills, repeating the concepts learned in parts 0,1, and 2a-c)
- *template literal* used, with the backticks ` ` and placeholders (JS code injections), simplest case like this: ${name}
- alert dialog is one of the other good old tools (used in fast debugging but **nowhere else** today. Use the debugger instead where possible/appropriate).
- *exercises.* Definitely good thing to do later so that you remember all issues recapped here

Part 2e. Adding styles to React app

- (These style examples in part 2e are mainly for **very basic way of styling**. Fully-fledged / professional styling would involve injecting the Theme (e.g. Material UI Theme) to the React-dom via some tree root React component and e.g. style files written in their own .js code files)
- But this is a good start / introduction to the topic
- based on CSS basics of styling HTML elements, that are produced based on the JSX code(s).
- and the fact that the index.css was (was it not?) added to the create-react-app project template
- and using HTML attribute class first, later applying the JSX attribute 'className(s)', which will **dynamically** produce the wanted 'class' attribute to the output when JSX rendered to final HTML
- notice again how HTML elements are with all small letters, and in JSX you need to use exactly the given casing, e.g. className.
- Same as with the difference between static on**c**lick (HTML) and dynamic on**C**lick (JSX, to be rendered to HTML).
- finding in which order the CSS file and inline styles affect is sometimes a difficult task. Finally of course the CSS cascade rules rule. But the rendering from JS and JSX code to HTML is of course not too easy to read.
- You write the JS, JSX, and edit the HTML, but the processing/rendering magic happens hidden inside the React library code. But also here practice and experience helps.
- Idea of having a state piece for 'errorMessage' too, updated like the state was updated before. Used to show the possible error message. React will render it to the page if state has the errormessage. Removed when the errorMessage in the state removed.
- notice the differences between .css files and .js style files (or style definitions) that define style objects. Difference in use and also in syntax. A bit like difference between JSON text and JS object literal. But also CSS is static and JS can do dynamic tricks.
- What is here called inline styles can be also kept in external style files and so on. A better, more modular and reusable coding style.
- *exercises*

Part 3a. Node.js and Express

- "*NodeJS, which is a JavaScript runtime based on Google's Chrome V8 JavaScript engine*".
- Notice: Node can be and is normally used to create backend, but also used also e.g. in create-react-app generated **development time** frontend environment for fast development
- NodeJS's ECMAScript support has constantly improved, so could we use e.g. ES6 import/export (instead of the require and module.exports from older CommonJS) without transpiling with babel? "*we can use the latest features without having to transpile our code*"
- npm init, package.json (dependency definitions and scripts, plus more), node_modules folder
- const app // Usually/later refers to the Express object we use to define more and more features to the starting Node (w. Express) backend app/server instance. Like configurations, security settings, routing, starting the server via it, …
- Now here, first the const app refers to a different object from the Node's own npm module 'http', a bit cumbersome way to do things, but later again to Express.
- remember that if running front and back in same machine, they might both have Node server and thus eager to use port 3000 => clash. Thus we use e.g. 8686 for front and 8787 for backend Node. Thus 3000/3001 are free for your other Node projects.
- "*At the time of writing this material, Node does not support ES6 modules, but support for them is coming somewhere down the road*" Hmm. is the material up-to-date for this part?
- npm install somemodule // adds the dependency to the package.json and also downloads the needed file(s) to the node_modules folder tree
- "express": "^4.17.1" // caret ^ means at least that version or more recent, 4=major, 17=minor, 1=patch version numbers. Major has to stay at 4, unless you explicitly update it
- npm update
- npm install // this you need to run e.g. after cloning a project from git. As you get package.json from git repo, but not the node_modules. So repo does get the dependency definitions, but not the physical module files. One dependency might depend on other modules, so you will get more than one file/module downloaded per each dependency. Usually actually tens of thousands of tiny files that would be painful to track and transfer!
- app.get, app.listen,
- response.send (200, text/html), response.json (200, application/json)
- nodemon (node-mon(itor), not: no-demon 😊): *nodemon will watch the files in the directory in which nodemon was started, and if any files change, nodemon will automatically restart your node application*
- node index.js vs. nodemon index.js
- We have some aspects about REST services presented elsewhere on the course. Only the core stuff
- HTTP verbs = HTTP methods https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods Some systems only use(d) GET and POST (a new resource posted to the backend) to do everything (instead of including e.g. the useful DELETE, PUT (replace) and PATCH (modify). You could use these five or four of these, skipping PATCH.
- The HTTP verb names are thought from end-user point of view. User GETs a page/data from backend. User POSTs something to the backend. Backend handles the POST request by using whatever it receives from the user's POST.

- the nature of app.get we have already discussed (We look at our asynchronous code examples elsewhre, the code might be executed in three different times / phases, the app.get being executed at server setup, others later if the service called, and last after some async resource like database responds)
- app.get('/api/notes/:id', (request, response) => {
  … const id = request.params.id
- notes.find … // refers to the array/collection method *find* that is the sibling of filter/map/foreach/reduce methods. Returns the first item in collection for which the provided arrow function returns true
- notes is so far just a hard-coded JS array object let notes = [...something here…]
- (Notice: Why it's let and not const? Is some action going to replace the array with a new object (=reference to a new object)? Follow this issue!
- response.end( ) // from Node itself, just ends the response processing, you need to set the status code and response type etc. before this.
- response.send( ) // from Express, sets /sniffs some stuff for you automatically (sometimes/usually the one you need to use), uses internally the response.end( ) of Node. "*res.send() implements res.write, res.setHeaders and res.end*"
- deleting is again done just funny way of filtering out from the local object, no database yet. Would const notes have functioned ok here?
- **Postman** or **VS Code REST client** add-on.
- const note = **request.body**; // handling POST. Remember before **request.params.id** for GET
- (GET method can also have a body = data, but not all implementations support it like they **should support** based on the HTTP standards. So if you use body for GET, make sure to test it well before! Or don't use the data body for GET, as people have forgotten/decided/been lazy to support it)
- remember how array.concat(newObject) did not modify the existing array, but returned a copy with the newObject added to it
- Math.max(... notes.map(n => n.id)) // interesting pile-up of execution: 1. map 2. arrow function used inside of map for each item, returns an array, 3. spread notation called for the array
- { important: body.important || false, } // test this with simple example of yours to understand. What are the two possible values for the one value saved into the variable important?
- *exercises*
- safe and idempotent
- "middleware". For e.g. schema validation, centralized error handling, routing management, json parsing, …
- *exercises*

Part 3b. Deploying app to internet

- look at the baseUrl as an example of "define only once and in one place". Later of course from env variables.
- CORS, or Cross-Origin Resource Sharing. By default the CORS mode is restrictive and may forbid AJAX calls, as our AJAX in full-stack apps is naturally calling the backend, which we have on a different server than the frontend.
- Here note that dev time frontend might have some code running against the React-dev Node server, but in final stage, the frontend is running totally on end-user's web browser. But the web browser has got the HTML and JS from some web server, and then the AJAX code wants to connect to a backend server.
- Thus you can see, that later we need to change the CORS config so that the browser will get content from all the backend**s** (REST API backend, media server backend?) in the frontend.
- npm install cors, const cors = require('cors'); app.use(cors());
  The last step creates CORS configuration with no restrictions. Ok for helloworld, but for real environments we shold only allow the domain URL of the backend.
- When you look at the "*The setup of our app looks now as follows:*" picture you can see how in the full-stack course example the frontend and backend are in the same computer (student's PC). Usually that is not the case, we e.g. want to separate the backend and frontend servers, thus we could separate the storage symbol at the botton into two storage symbols. app.js and index.js on one frontend server computer and the index.js of the backend server in another.
- Heroku – When **learning any cloud environment** try to see the difference between Heroku-specific learning and common learning of any cloud infrastructure. Not spending too much on details, but more with what exactly are we doing, the big picture. That kind of understanding can be move to e.g. Google cloud, or Azure, or AWS. But the details and exact commands usually not!
- const PORT = process.env.PORT || 3001;
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_OR
  "if this operator is used with non-Boolean values, it will return a non-Boolean value". Thus you can use it in conditional assignments.
- Notice the "template literal" with backticks:
  - `Server running on port ${PORT}`
- "Frontend production build" we have already been talking about this. Finally, the React-dev Node disappears, and the produced mashed up HTML+JS code could be served by a static server even, like myy.haaga-helia.fi even. No dynamic server needed, no node, no apache tomcat.
- we will not put the frontend deployment files to the backend server. We want to be 'kosher', keep front and back totally separate project and server. Why? because later we could have e.g. native mobile app that uses the same backend, but has no relationship with the frontend whatsoever.
- bad bad bad, though this is kind of beginners' course to full-stack. But think about this prob too: You want to monitor if something has changed in the server. If both front and back are on the same server, you lose the knowledge which one has changes. Testability (e.g. regression testing) and so on complicated unnecessarily!

- "*The setup that is ready for product deployment looks as follows:*" picture. Looks confusing, front and back mixed up to the same server. Otherwise nice picture, of course.
- nice scripting help in this part! including how to make Windows use bash console to run the scripts nicer
- "for example placing both backend and frontend code to the same repository" Please, Noooouuuuuu! ☹ Full-stack means that the architecture consists of front, back and storage, but in no way should they be mixed as one entity.
- I have seen people do a hack where the REST API backend and the frontend to run in the same runtime. Not React/Node, but was it Ruby on rails or something. Even if it's interesting technically, it's against any sensible principles (SRP, modularity, testability, decoupling, replaceability…).

Part 3c. Saving data to MongoDB

- I sometimes counted the possible windows you could get open and monitor while running a full stack app with React, Node and database. I got to 8 different windows to monitor, that of course includes the web page itself, but then the browser console etc. Eight screens where some error or info message could tell what is happening or what is the problem!
- This of course depends on the tools you are using, e.g. React dev tools and Redux dev tools where two of the eight windows!
- The eight windows provided a view inside the full-stack "black box".
- VS Code debugging for Node.js, launch.json
- (Interesting, using Chrome dev tools to debug Node.js code. Though, why not, Chrome and Node both utilize the same V8 JavaScrpt engine)
- Old good advice, stay lean, test, add more, test. Test as soon as anything is testable. = test even just SQL if RDBMS. Write test and test even an empty API end point. = be in control of the code rather than code takes control of you.
- Document database is official and better name than noSQL. MongoDB, Google/Amazon/ etc. have their own and also offer MongoDB service, (ElasticSearch is behind of many very fast applications)
- Remember that document databases are not replacing relational databases, but complementing the spectrum. Document databases are fast in bulk output, bulk reports and summaries. Relational databases guard the integrity of data model and more complex business rules better. And are, actually, faster in more complicated searches.
- So certainly needed, but not replacing. Many information systems use both. Relational database for input to the normalized, guarded, high-integrity data model. And then document database for bulk output. Data flowing e.g. from user => RDBMS => document database => user (big lists, summary reports, BI mashups).
- Database - * Collections - * Documents -* key:value pairs
- BSON, Binary JSON, https://docs.mongodb.com/manual/reference/bson-types/
- BSON fully supports JSON datatypes and then adds more specific definition possibilities that JSON cannot "enforce" or relay, but they go through as JSON **data** fine, just lose the BSON restrictions.
- Heroku/Azure/Google/Amazon/IBM/mLab/DigitalOcean would provide MongoDB as service, this full-stack course uses: https://www.mongodb.com/atlas/database Atlas MongoDB
- Mongoose. Schema + model (Model is like the 'ORM' programming side object. At the same time kind of our access to and from the database, enforcing the schema on the way)
- Document databases on their own are schemaless, so we put the schema validation "at the Mongoose gate to MongoDB"
- (((In relational databases, SQL, SCHEMA has a totally different meaning. It's the namespace where tables and other objects can reside. Sometimes called DATABASE, like in MySQL. In MySQL and MariaDB *CREATE SCHEMA Xyz* and *CREATE DATABASE Xyz* mean the **same thing**)))
- **note** (document), **notes** (collection of documents). Though with capital letter **Note** refers to the model and the model able to fetch both single and multiple items.
- The search object: Note.find({}) Note.find({important: true})
- *exercises*

- a student told (2021-10-26) that mongoose has to be 'required', 'import' not supported yet. These ECMAScript etc. support things tend to change in the future.
- Just follow the example growing with the REST API end-point like in our async examples. Thist time just MongoDB with Mongoose, we had RDB with Knex.
- toJSON + transform: some data manipulation added to the Schema for the from DB to backend way
- here we could change fields, scale values, remove fields, or add metainfo or such fields (sometimes adding is called 'decorating')
- GUI = Globally Unique Identifier. Can be created both in frontend (user interface) **and** in database (data model, storage side). So random hash generated that two exactly will 'never' occur.
- Backend project folder structure: 'models' folder, 'routes' or 'controllers' folder, … and shorter SRP principle abiding files that do only one thing
    - (the requests folder is for the VS Code REST client tool saved test requests)
- … const url = **process.env.**MONGODB_URI … (to read the (usually Linux) OS system variables in)
- dotenv module, its config function, and the .env file (.env files are naturally git ignored)
- *exercises*
- error handling and the HTTP status codes. Covered in some of our materials.
- look here for the HTTP status codes. https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.1 E.g. '201 created' would be possible, but almost everybody uses just '200 ok' to tell that new item was successfully added=created.
- Using middleware to e.g. do the error handling brings a little bit of more complexity, and a little bit overhead, in the beginning. In large app though the investment pays off later.
- ((Express middleware docs: https://expressjs.com/en/guide/using-middleware.html))
- middleware: logger before anything, unknownEndpoint route handler after all real routes, errorhandling at the end.
- *exercises*

Part 3d. Validation and ESLint

- Schema: Adding the validation rules to protect the database from incorrect input
- Remember that MongoDB doesn't enforce these rules, like RDBMSes do. The rules are only applied if nobody makes the mistake and codes something that will bypass the Mongoose schema validations!
- In RDBMS the rules are in the database, guaranteed to be applied no matter what.
- for MongoDB, the rules are in the backend code. Hopefully in all of the backend code! But it is possible to make mistake and forget to do the schema validation.
- By the way don't create too restrictive limits for fields without consideration. E.g. in Swedish language there are some words of one letter
    - å = river
    - ö = island
- chaining the promise-powered handler methods .then().then().then()
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise Promise wraps the pending/fulfilled/rejected 'status' and the final output value of an asynchronous operation as one object. The idea is that the final user of some system doesn't have to spend extra time coding the async, but just use .then() .catch() methods of the promise.
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then
- (Heroku specifics)
- *exercises*
- Lint = stuff that you can remove from the clothes with tape roller or a lint brush (delinting).
- Or the tool for flagging (and later programmer removing) bad code and coding style from your project. Lint or linter tool
- ESLint for JavaScript=ECMAScript
- When all the developers use same code formatter and lint tool, (with same configuration settings) they get code where only real code changes are regarded as changes by git version management.
- Otherwise one developer's IDE might put an extra empty line at end of each file, and for another developer remove it. Git might regard those as real code changes!
- Lint tool would be used by developers while writing code, and then by the DevOps tool chain, e.g. GitHub actions would check that there are no violations, otherwise the pull-request won't be merged. Also there we need same config settings
- *exercises*

Part 4a. Structure of backend application, introduction to testing

- The project folder structure is important for learning perspective as well

```
├── index.js
├── app.js
├── build
│   └── ...
├── controllers
│   └── notes.js
├── models
│   └── note.js
├── package-lock.json
├── package.json
├── utils
│   ├── config.js
│   ├── logger.js
│   └── middleware.js
```

- The build folder is for the deployment output, so you can forget that for the time being. The deployable version of the project will be found there after the npm build command. (Not needed while just developing and coding)
- index.js is the starting point of the node backend server building. app.js is next in line adding more and more features utilizing Express.
- controllers provide routing and end point code
- models provide the models and their schemas
- (package-lock.json file is not important, you can delete it and it will be re-created)
- package.json contains the dependencies (which node modules needed) and e.g. scripts for starting the backend, or for making a deployment
- utils is then common, shared stuff needed by the whole app, by many features
- read the code examples.
- a lot of repetition from previous parts.
- note that we have three totally different kinds of routing in full-stack apps:
    - front-end routing inside a SPA app, like React routing = making certain components to become visible and old ones disappearing, making an illusion of navigating between multiple 'pages'.
    - internet routing of IP packages between frontend and backend and database server
    - backend routing based on the request URL finally to the correct handler method for that API call.
- with the Express Router we decouple the URL routing and the handler method. And can also make the routing be built modularly. Makes modifications easier in the future. But of course, again means few more files to do same thing as before. Investment now + rewarded later.
- loggers typically have different levels, e.g. error, warning, info. Then maybe development time you are interested in all of them, but while running the deployed code in production, maybe just errors. = filtering the information
- *exercises*
- *module.exports = { // so it's possible to export multiple things*
  palindrome,

average,
}

- …. and later select which part of that to require require('../utils/for_testing').palindrome
- jest - JavaScript (unit and integration) test code tool, Test framework. Consists of:
    - matcher functions (expect().toBe())
    - test running scripts ("test": "jest --verbose")
    - configurations
    - test reports (with green or red)
    - test suites, test cases, test descriptions (used by the reports)
- *exercises*

Part 4b. Testing the backend

- Different scripts starting the backend either in development or production mode
- mongo-mock (Mocking means replacing some a) not yet implemented real module OR b) not-easy/free/handy-to-call real module with a mock-module, that e.g. returns some hard-coded result per a certain call. Kind of we would have that module for real, but it actually doesn't work dynamically yet, only gives correct answers to certain test input)
- "It is common practice to define separate modes for development and testing."
    - "start": "NODE_ENV=production node index.js",
    - "dev": "NODE_ENV=development nodemon index.js",
- (then material goes even more to DevOps and testing, we need to stop somewhere…)
    - Actually new course on e.g. automated testing is coming, includes naturally a lot of DevOps too…
- Just remember that in all this dev env vs. deployment to real running environment etc. our final goal/dream could be automating and container:ing everything. But go step by step.
- **async** is a keyword telling that the function is asynchronous.
    - thus it has the parts that will be started immediately but the result will be **await**:ed for. It means that the execution of the thread will finish but the **await resume-spot** resumed when result comes back. (Like a fork, the synchronous thread will stop executing the function and possibly executes any other code of the calling context, but the asynchronous resume-spot **await** will later start a new thread to deal with the result and the following code.)
- (lot of this backend and test tech stack testing stuff skipped)
- *exercises*
- (lot of this backend and test tech stack testing stuff skipped. Useful, but not topical for us now!)
- *exercises*

Part 4c. User administration

- Authentication = identifying and verifying the user, Authorization = empowering the user and linking him/her to the allowed actions and/or resources.
- **OAuth** = Open Authorization (With some method authenticated user (0.) gets a token that authorizes (1.) some actions)
- *lookup aggregation queries*. Making relational database kind of left outer join between collections in document database MongoDB
- (relational databases are intrinsically better suited for all kinds of combinations of data. document databases like to output hierarchical trees)
- **bcrypt** package for creating password hashes
    - (You have most likely heard it already: Never keep clear-text passwords anywhere. Big security risk also because people use same passwords across different services)
- more Mongoose Schema definition options: unique
- one-way-hash function: https://en.wikipedia.org/wiki/Cryptographic_hash_function just look at the colorful picture. Digest is of fixed length and changes drastically based on even the tiniest change in input. E.g. capital or small letter.
- TDD – Test driven development. "First write the tests, tests fail as implementation is missing, write implementation and test until tests go through"
- (skipped MongoDB and Mongoose details)
- *exercises*

Part 4d. Token authentication

- Authentication = identifying and verifying the user, Authorization = empowering the user and linking him/her to the allowed actions and/or resources.
- JSON web token (JWT) -tools allow you to decode, verify and generate JWT:s
    - some claims (usually encrypted) moved together with the hash to check the integrity of the claims (that they are not forgery). So both data and the verification hash
    - either for
        - authorization to access some service and/or resources
        - information exchange
- These tokens passed in the headers of the HTTP Requests. And frontend is including the needed tokens in headers while calling the backend API.
- (more details related to usage of JWT in this case that can be skipped for now)
- *exercises*

Part 5a. Login in frontend

- Note how these two lines are not similar by any means:

```
const [notes, setNotes] = useState([])
const [notes, setNotes] = []        // JUST WRONG!!!, READ BELOW
```

In the first one the empty array goes to be the initial value of the notes state, but also the useState returns an array, where the first item is the empty array and the second is the setter function. And those two values from that array would be destructered to be the values of the two consts created on the left side.

To understand the first line one needs to understand three separate things:

- The JavaScript destructuring assignment, e.g. const [a,b] = [12, "abc"];
- The React states (~component state fractions) and state setter functions
- The useState hook that is given at least the initial value of the state fraction

The second version would be totally wrong even from the JavaScript destructing assignment point of view!

- Notice how this component's (the main page/view of that app) state fractions have different datatypes: array, string, Boolean, object, string, string
- useEffect = https://reactjs.org/docs/hooks-effect.html: "The Effect Hook lets you perform side effects in function components". Side effect = doing other things than just putting the component to the DOM and setting the state(s).
- Here "side effect": Fetching the all notes data from backend. So "side effect" can be part of the main wanted activity! Just not just the component state etc. React-kind of updating.
- What's happening here?

```
<input
   onChange={({ target }) => setUsername(target.value)}
/>
```

It's an HTML element, but the attribute **onChange** is JSX (not yet onchange that would be already ready HTML attribute) will still be first rendered by the React parser and handler.

The outer { } change us to the JavaScript mode. There we define an arrow function ( ) => setUsername(target.value) and pass that code. As we pass the event-handler to be bound to the event, we don't naturally call it (as the event hasn't yet happened!), but pass it as function object (because of the definition). "Not the cake but the recipe on how to make the cake"

And the {target} is a function parameter list placed special destructuring assignment that takes whatever that function is given (the props!) and takes from that props ('event') the value 'target' and puts it to local variable = to the function parameter variable 'target'.

Target of course being the event.target = the web page component for which the (on)change event happened.

If / when that change event finally possibly happens the setter for the state will be called and the changed 'value' of the component will be set to the 'username' state

(Ouch and then I see almost all same stuff explained on the page too. But not exactly with the same detail. Read both and you have a chance to understand it all?)

- **async** – keyword that tells that a definition of an asynchronous function will follow
- **await** – keyword that means something like this: while executing that async function
  1. initiate some inner function call immediately and then stop executing the rest of this async function and continue with other things of the application in this first thread, though…
  2. at the same time setting a return point / a waiting point (that await spot) where system should resume (possibly in another thread if in multithreaded system) when that await marked function call is ready. And then execute the remaining code lines too. They are usually dependent on the result of the awaited call.

```
async function asyncCall(input) {
    console.log('calling');
    const result = await someFilesystemAjaxOrDatabaseOperationThatTakesTime(input);
    console.log(result);
    // expected output: "something from the other system"
}


// later somewhere initiating the whole activity:

asyncCall("chocolate");
```

- Conditional formatting utilizing the shortcut evaluation of the AND operation. Only if user variable has a null reference will the left side be true = only then the other side of the AND be executed = login form rendered to the DOM to that place.

```
{user === null && loginForm()}
```

- Local storage support came to browsers with so called HTML 5 some 10+ years ago
- Using the Effect hook also for storing the login credential to Local Storage. Again this "side effect" is important and crucial part. Just not part of React state handling and react rendering, thus a "side effect".
- *exercises*
- By the way the Browser's JavaScript/ES support is implementation of so called DOM API. Thus e.g. the local storage methods are part of the standardized 'DOM API' since so called HTML5 version 'this and that'. https://caniuse.com/?search=localstorage

Part 5b. props.children and proptypes

- in CSS, the rule **display: none;** => hides the block-level element (box) of the web page, keeps it in DOM, but doesn't reserve any space for it on the web page. = totally collapsed and disappeared. => in most use cases better than **visibility: hidden;** which still affects the layout: https://developer.mozilla.org/en-US/docs/Web/CSS/display-box
- props.children = child components of the current component!
- Though self-closing tag, e.g. <Abc def={ghi} /> doesn't have any content = child components, thus props.children is an empty array [].
- {props.children} can be used to render the children, (((if children are not a list of list items or so. The forEach or similar iteration with unique id:s needed?)))
- *"Often, several components need to reflect the same changing data. We recommend lifting the shared state up to their closest common ancestor."* This React doc advice is for the case when we do not have e.g. Redux to share the common frontend copy of the needed data (or the "state")
- …and then passing the data down to children via props.
- Ref in React is a bit like Goto in programming. https://reactjs.org/docs/refs-and-the-dom.html Offers shortcut to components to do special things like setting focus to certain place, but should be avoided if possible to do with the declarative normal React way.
- useRef hook
- *exercises*
- PropTypes, npm install prop-types, setting e.g. type and isRequired check
- ESlint, .eslintrc.js, .eslintignore,
- ESlint with shared setting file means all developers write better git-mergeable code, as mere changes in the white-space etc. will disappear, as they are no real code changes.
- Linux will not show system .files unless say e.g. 'ls -Falls' (actually ls -a is enough, **a**ll)
- *exercises*

Part 5c. Testing React apps

- Jest – JavaScript **testing framework**
- **Jest** contains e.g. how to do following: unit tests, mocking, reports, test coverage, HTML component snapshots, …
- Typically test frameworks have library modules for specific test target, e.g. React apps
- Jest can be then used even to test rendering of single components
- Watchman to see which files have changed => which **regression tests** are needed to run. (Best would of course to run all tests always after any changes, but that sometimes takes more than the night for large systems)
- jest-dom library to check what really went to the DOM
- https://en.wikipedia.org/wiki/Mock_object In object-oriented programming, mock objects are simulated and/or hard-coded objects that mimic the behavior of real objects in controlled ways
    - if we do not have the working real object yet, or
    - if calling the real object/system would cost money
    - (or if we want to get certain values from a source that might give lot of random values. Then our tests can expect those values. => tests less flaky and more deterministic)
    - can you think of more reasons for mocking?
- Test coverage % of: Statements, or Branches (if-else), Functions, code lines, … sometimes e.g. colors of lines not covered too.
- *exercises*
- What tests make sense to do depends on the system and architecture. The TDD has maybe a bit eased off nowadays. More need basis testing nowadays, even if high coverage aimed.
- Snapshot testing to see if we still get same HTML output or something has changed

Part 5d. End to end testing

- **unit testing** (classes, objects or functions), **integration testing** (some independent aggregated entities, e.g. REST API with endpoints), **end-to-end testing/ system testing** (full-stack with end-user or e.g. automated: scripted or recorded use of the user interface
- User interface included E2E testing e.g. with one of these libraries:
    - Selenium, robotframework-browser, playwright, cypress
- Headless-mode – Testing graphical user interface without having graphical user interface. Running "inside memory" "simulator behind the scenes".
    - Even being able to take and save screenshots, without a screen
    - useful when running tests in cloud server, without monitor and such
- Regression testing : Testing that the new changes (to other parts) did not spoil previously working features
- cypress test project could be even its own project/repo
- Each E2E tool has its own code or scripting for creating the UI tests, how to find certain component, how to write something to the input field, how to 'click' a button, …
- … more testing details …
- *exercises*