SENG 365 Summary Notes

# A picture containing text, diagram, screenshot, plan Description automatically generatedLecture Note Set 1 – Intro to HTTP and JS

### Concepts covered in assignments (might be useful):

Assignment 1:

* HTTP Request/Response.
* URL.
* HTTP Headers and Body.
* Rest methods.
* Status codes.
* Asynchronous requests.

Assignment 2:

* HTML, CSS, JS.

### HTTP

* Text encoded in ASCII.

### Uniform Resources

* URI (Uniform resource identifier) – String of characters to identify resource.
* URL (Uniform resource locator) – A URI that specifies the access mechanism e.g. https, http, ftp.

### Structure of a URL

http://www.example.com:80/path/to/myfile.html?key1=val1&key2=val2#Somewhere

* http:// - Protocol.
* www.example.com – Domain Name.
* :80 – Port.
* path/to/myfile.html – Path.
* # - Anchor.

### Headers

* General headers are always required, Entity headers only apply to the body of the request.

### Body Types

* Single-resource, known length, has headers Content-Type, Content-Length.
* Single-resource, unknown length, Transfer-Encoding=chunked.

### Response Codes:

* 1xx – Informational.
* 2xx – Success.
* 3xx – Redirect.
* 4xx – User Error.
* 5xx – Server Error.

### JS Functions

(function () {

Statements

})();

* + The outer brackets enclose an anonymous function.
  + The empty brackets () executes the function.
* JS executes any function using the variable scope defined at the definition of the function, not at the time of invocation of the function.
* Let x = …; - hoisted variable, pulled to the top of the function as if it was defined at the top.

### This

* By default, references the global object.
* In a browser, the window.
* In node, the global object.
* Can access on an object e.g. foo.bar.

# Lecture Note Set 2 – More JS and Asynch Flow

### Arrow/Anonymous Functions

* (a, b) => a + b;
* Arrow functions do not bind their own this.

### Promises

* Can be used when performing a nested/conditional set of API calls.
* Promise object – for deferred and asynchronous computations.
* Three states – pending, fulfilled, rejected.
* Use two branches for fulfilled and rejected promises e.g. response, error in front end calls.
* Dependent asynchronous operations can be chained with promises.
* .then() to chain and pass result to another function/promise (?), technically returns a promise.
* E.g. f1().then(f2(result)).
* .catch to handle rejected states.

### Async/Await

* Async makes a function return a promise.
* Await forces JS to wait for a promise to resolve.
* Await is only legal inside of an async function.

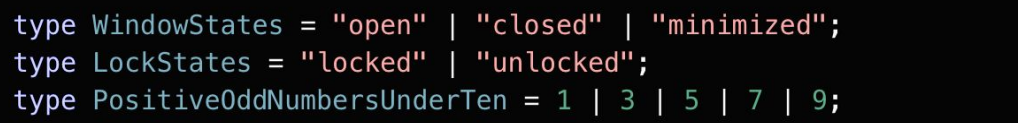
### Imports

* Export using module.exports = {}
* Import using require()
* var name = require(‘../../myModule.js’);

# Lecture Note Set 3 – TypeScript and Data Persistence

### TypeScript

* Adds types, static typing, compiler type checking to JS.
* Let decimal: number = 6; etc.
* Optional property: age?: number.
* Npm I -D typescript to make a typescript project.



### JSON

* All keys double quoted

### ACID

* Atomic – all or nothing.
* Consistent – db is consistent before and after execution (the db obeys the db rules before and after the transaction).
* Isolated – one transaction cannot see the effects of others in progress.
* Durable – transactions are persistent once committed.

### BASE

* Basic Availability – through replication.
* Soft State – the state of the system may change over time.
* Eventual Consistency - the data will be consistent eventually.

### CAP

* Consistency – Every read receives the most recent data.
* Availability – Every read receives a response.
* Partition Tolerance – System continues if network goes down.

### Key Value database

* Unstructured, primary key is only lookup.
* Create, read, update, delete.
* Simple and fast, but no validation or aggregation, and consistency is hard. No selects either.

### Document Database

* Semi-structured.
* Typically JSON or XML.

### Graph Database

* Node – entity.
* Edge – relationship.
* Can be uni or bi-directional.

# Lecture Note Set 4 – REST, APIs, GraphQL

* Get
* Post
* Delete
* Put (change whole resource)
* Patch (change part of resource)

### REST

* All requests are stateless i.e. no nothing about the status of the serer.
* Request/response – can’t push/alert or broadcast
* Multiple request/responses implies tree-structure, can cause underfetching and overfetching, and can increase latency.

A picture containing text, screenshot, font, number

Description automatically generated

### Cookies

* Key-value pairs with attributes and expiry.
* Maintain state info.
* Types:
  + First party (set by server)
  + Session (deleted after browser is closed)
  + Persistent (not deleted after browser is closed)
  + Secure (only transmitted over and encrypted connection)
  + HTTPOnly (can only be transmitted through HTTP/S)
  + Third party (set by third parties e.g. ads)

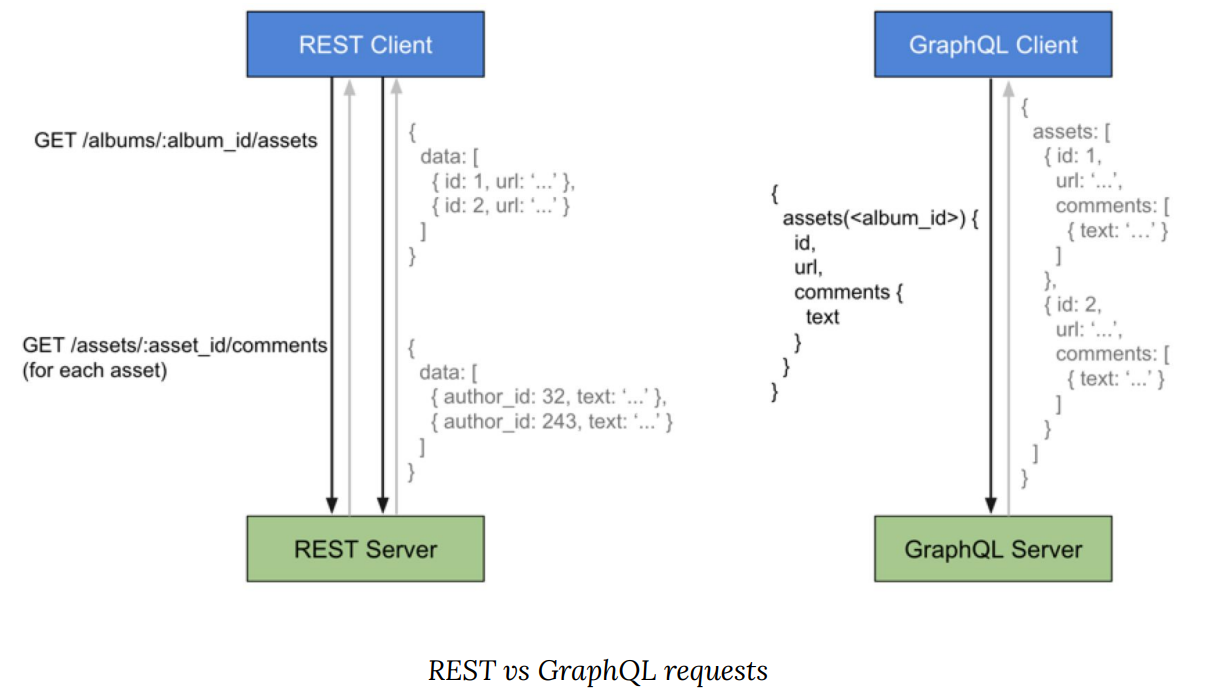
### API Versioning

* Major.Minor.Patch

# Lecture Note Set 5 – GraphQL and API Testing

### GraphQL

* Solves overfetching and underfetching by requesting exact data required
* E.g.



* GET -> query is specified using the URL query parameters ([http://myapi/graphql?query={me{name}}](http://myapi/graphql?query=%7bme%7bname%7d%7d)
* POST -> query in HTTP body, use a JSON.
* Can be used as an intermediate server between client and REST server.

### Backend Testing

* Can set up pre and post conditions – before(), beforeeach(), after().
* Tests are asynchronous, so test 1 may not necessarily finish before test 2. Use before(), after() etc.

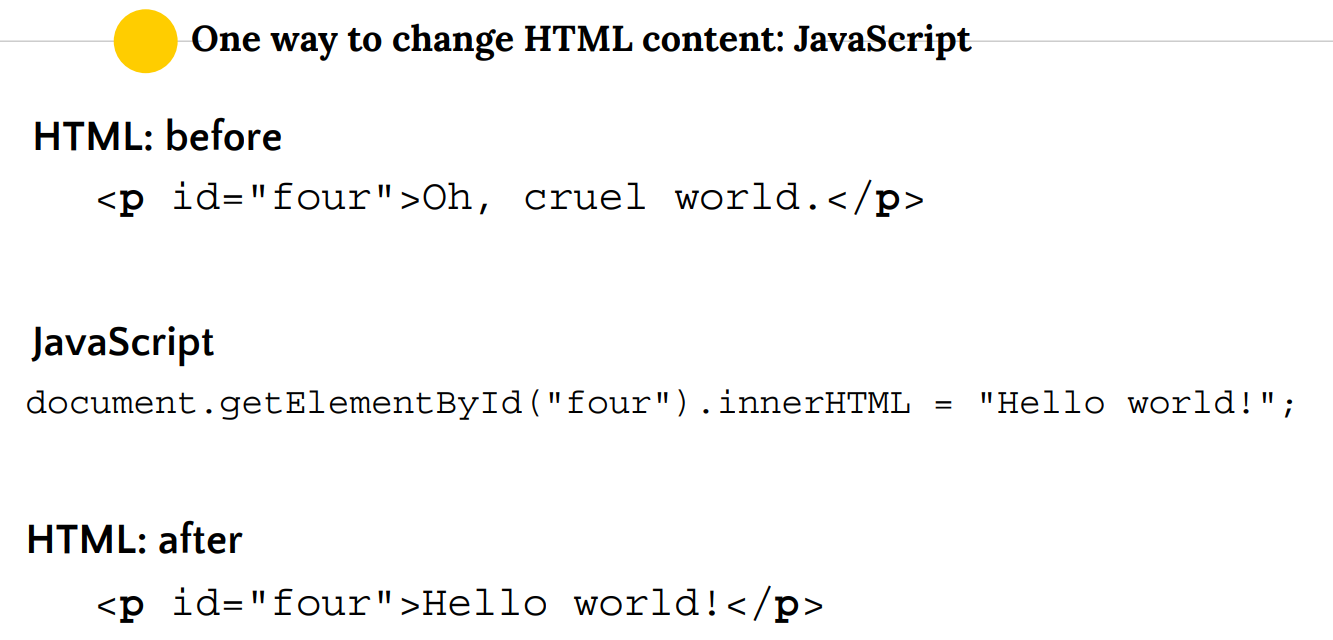
# Lecture Note Set 6 – Security and Intro to Client Side

### Security

* Injection – sending malicious code through an application (e.g. sql injection).
* All inputs are injection risks.
* To secure passwords:
  + Hash username and password.
  + Require users to change passwords frequently.
  + Use multifactor.
  + Salt the username/password.
* Session Ids are just as good to an attacker as username/password.
* Credentials go with every request.
* Beware side doors – change my password, forgot my password etc.
* DOM based xss injection – Document Object Model used to introduce hostile code into client side js. Untrusted data should be interpreted as plain text, not code.

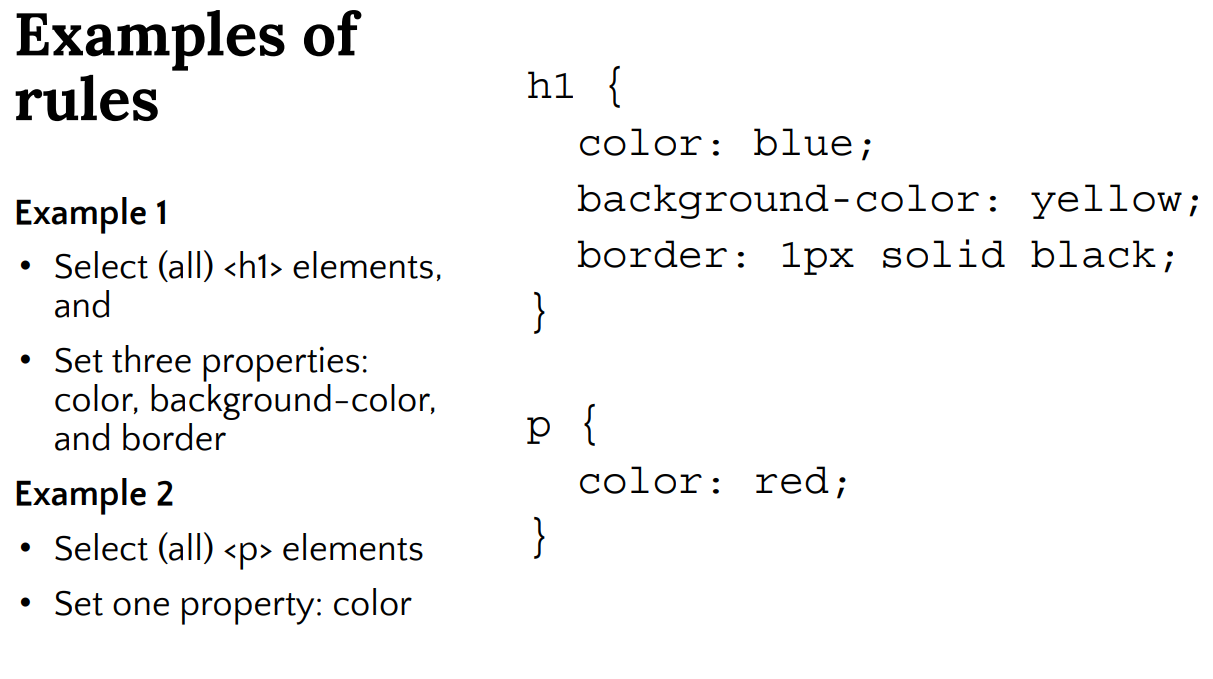
### HTML

* Hierarchical.
* <tag attribute=’value’> <tag/> etc.
* Normally has one <head> and one <body> per document inside a root <html> element. I think.
* Elements can be referenced by:
  + Element type e.g. <p>.
  + ID e.g. id=”…
  + Class e.g. class=”…
* Data -> JS data, variables etc. Content -> stuff in HTML.

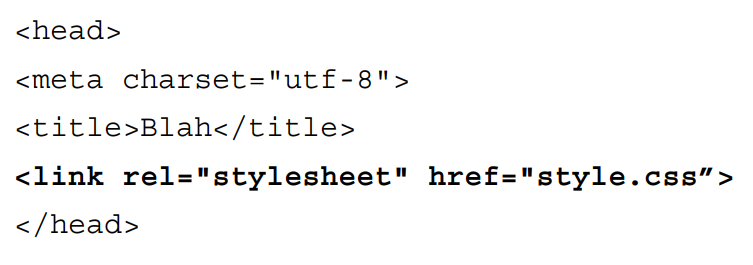


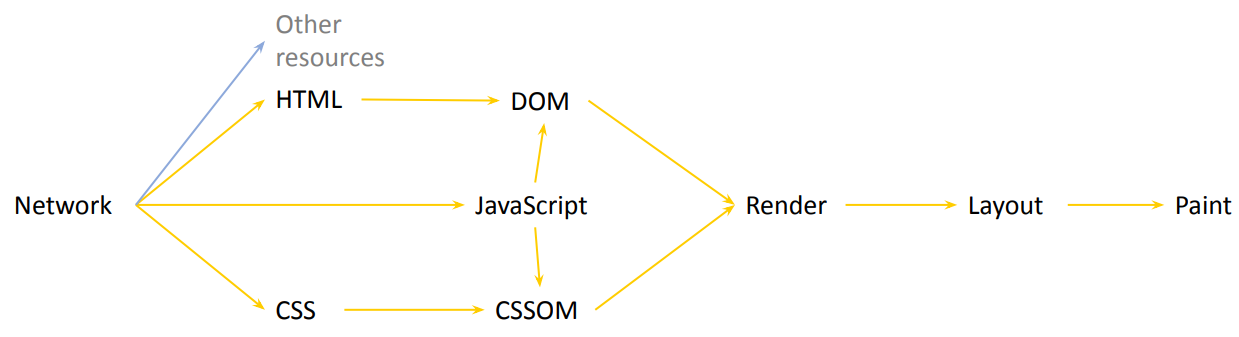
### CSS

* Defines how content is presented to a user.



* Normally in an external style sheet e.g. app.css.
* Import example:





# Lecture Note Set 7 – Single Page Applications (SPA) and JS Frameworks

### Single Page Application

* Separates data from content (presentation of data). Server requests aren’t required for the UI to be redrawn, since view layer is on client side.
* Makes webpage faster by reducing server communication, only fetching data from the server asynchronously.
* Client-side JS handles browser routing, page history instead of browser.
* Effectively rebalances workload between two applications – client and server side.

### Design Patterns

* Separation of concerns – Separate out code into different sections e.g. object oriented, MVC – divide and conquer.

### MVC

* + Can have multiple, synchronized views.
* + Views and controllers can be easily exchanged.
* - Complex.
* - High coupling

### MVP

* Events from model go through presenter to view instead of straight to view.
* Model is therefore completely isolated from view – less coupling (?).
* 3 layer architecture.

### MVVM

* ViewModel (intermediary) is just the data required by the view.

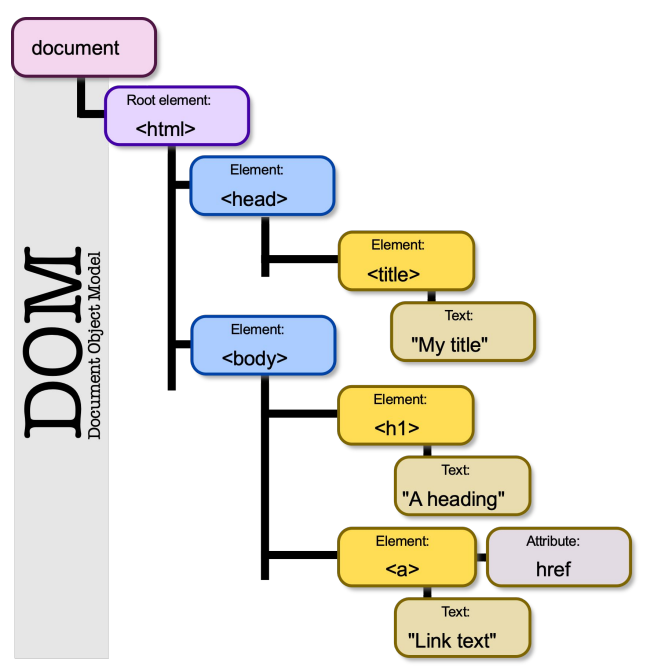
### Client-Side Rendering

* JS Renders the page in browser.
* Logic, data-fetching templating, routing also handled by browser code.
* First contentful page – First component to appear on page after JS bundle is loaded.
* Time to interactive – time when all functionality is fully loaded and render function is executed.

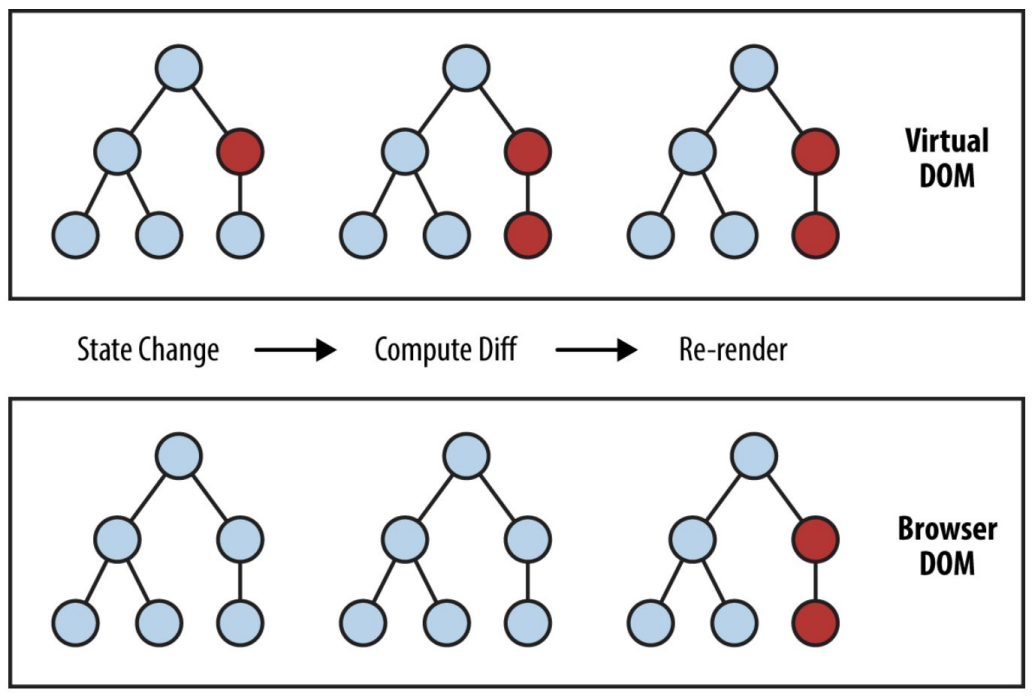
### Templating

* One-way binding -> model and template are merged once and sent to the view.
* Two-way binding -> View change -> update model, Model change -> view change.

### Document Object Model



* HTML Dom: standard model for HTML documents.
* Virtual DOM – intermediary to the real DOM that only updates the real DOM when it needs to.
* Updating the virtual DOM: When?
  + Dirty check: poll the data at regular intervals to check the data structure.
  + Observable: Observe for state change. If something changes, we know what to update.
* Updating the virtual DOM: How?
  + Efficient differencing algorithms
  + Batch DOM read/write operations.
  + Only update subtrees.
* i.e. perform individual changes on virtual DOM, then combine and make a batch change on browser DOM.

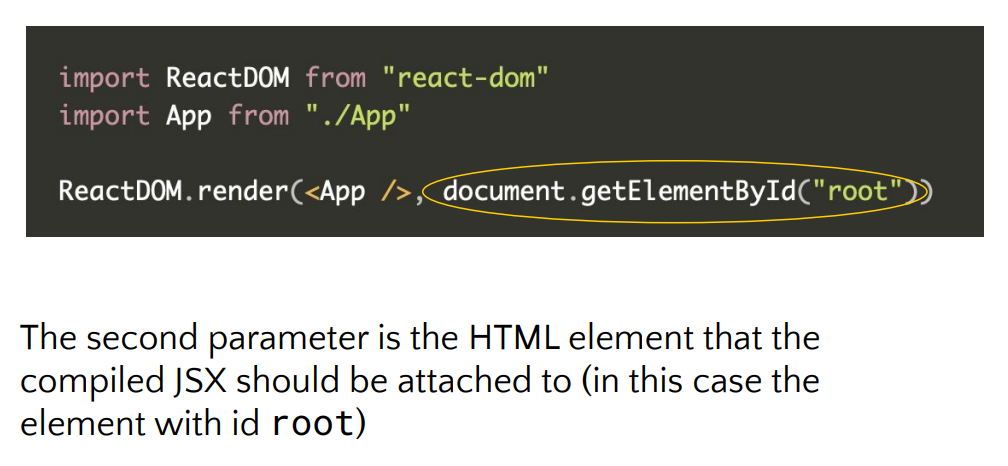


* React does not use templating, uses components in JSX (HTML inside JS), virtual DOM.

# Lecture Note Set 8 – JSX and Components

### React

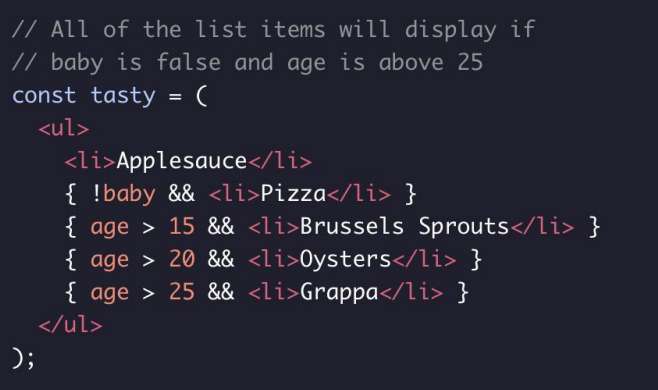
* ReactDOM.render – renders an element into the React DOM. Allows JSX to be rendered into dom.



* All app components are written in JSX.
* JSX is compiled to HTML before running.
* Actually just fancy syntax to call React.createElement.

### JSX Examples

* Multiline



* Nested JS

A picture containing text, font, screenshot

Description automatically generated

* List

A screen shot of a computer code

Description automatically generated with low confidence

* Creating a component



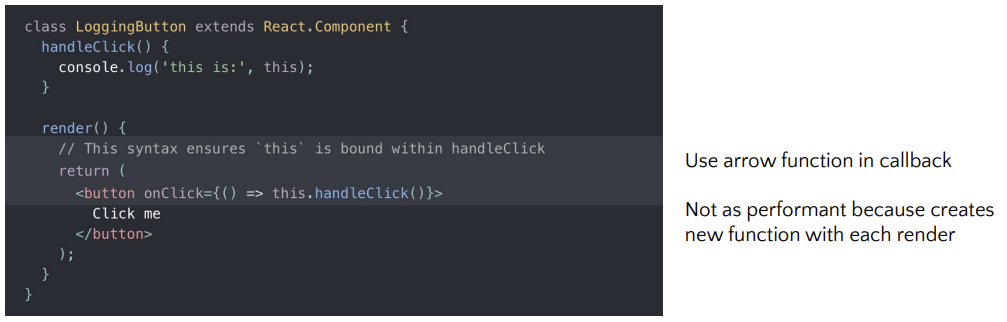
### Props and States

* Props: Read only, passed from component to component.
* States: Can be modified, state is kept per component.
* this.state() for a component’s state.
* this.setState() to set a component’s state.

# Lecture Note Set 9 – Event Handling, Hooks, Global State

### Event Handling

* onClick, onSubmit etc.
* Accessing component event example:

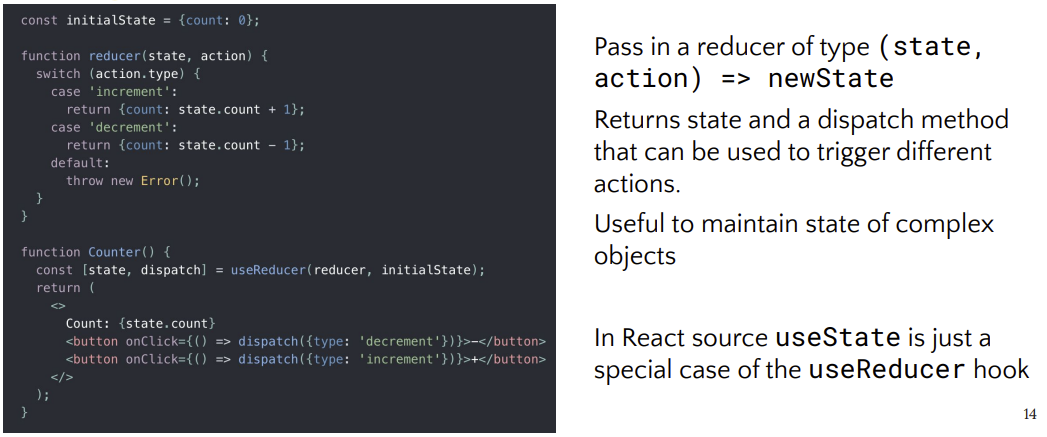


### Hooks

* useState – [state, setState] = useState(initialState). Can make an arrow function in body if we don’t want one in the render (for better performance).
* Accessing previous state in setter function i.e. optional parameter:



* useReducer – [state, dispatch] = useReducer(reducer, initialState).
* Basically a useState with a switch case.



* useEffect – useEffect(function, dependencyArray). Call function whenever anything in dependency array changes.
* Hook rules:
  + Only call at top level – not inside loops, conditions etc.
  + Only call from React functions or custom hooks.
* React Phases
  + Reconciliation: Detect any changes and update virtual DOM. Implemented using fibres.
  + Render: Update the app view based on new info.

### Global State

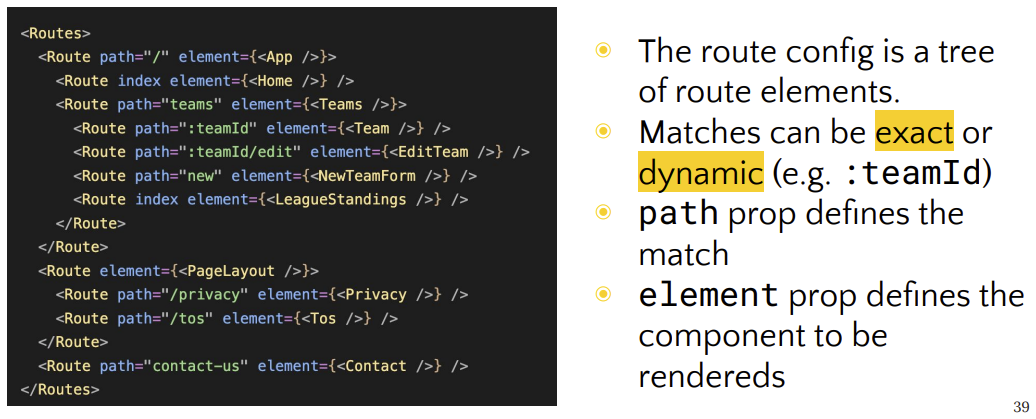
* Redux Model
  + View Actions Dispatcher Reducer State View
  + Action – JS objects that describe the action to be taken e.g. ‘login’. Switch statement.
  + Reducer - Pure functions that take actions and update state
* useContext

A screenshot of a computer code

Description automatically generated with low confidence

### Routing

* In SPA, when clicking a link:
  + The page is not reloaded.
  + Any new content needed from server is made by client-side code.
  + Don’t need to re-render entire page.
* In React:



# Lecture Note Set 10 – SPA Communication with Server

* AJAX – Asynchronous Java and XML
* XHR – XMLHttpRequest. An api. Uses JS to send HTTP requests. Used in Axios. Specify:
  + HTTP Request.
  + Expected Response.
  + Callbacks for handling each response type.
* CORS – Cross Origin Resource Sharing
* Fetch:



### CORS

* Defines a way to determine whether or not it is safe to allow a client-side app to make a cross origin request.
* Describes HTTP headers which provide browsers and servers a way to request remote URLs only when they have permission.
* Origin: Same protocol, domain, port.
* Some headers are forbidden and can’t be edited e.g. Origin.
* Examples:
  + Accept only from one origin - Access-Control-Allow-Origin: <https://foo.com>
  + Accept from anywhere - Access-Control-Allow-Origin: \*
* In essence: Accept cross origin resources only if accepted/defined by server. Reject otherwise.

### WebSockets

* 2-way communication.
* Maintains state.
* Fast, good for real-time.
* Server can push data to client.
* Compared to HTTP/AJAX:
  + Uni-directional
  + Stateless
  + Polling required to get updates.
* Web socket establishment:
  + Client sends initial HTTP Get to server.
  + Server responds with information on how to connect to socket server.
  + Client sends HTTP Get with header “Connection: Upgrade” and connection is upgraded to a socket connection.

# Lecture Note Set 11 – Web Storage and Progressive Web Apps

* Cookies limited to 4KB.
* HTML5 Web storage up to 5MB, with local and session storage.

### Web Storage

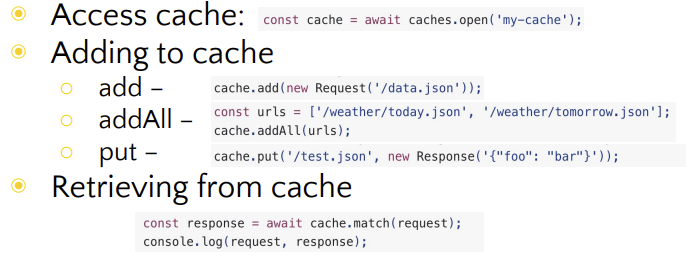
* Local storage: Permanently stores data for a site locally in key-value pairs, both as strings.
* Use localStorage.setItem, localStorage.getItem.
* Session storage: Temporarily stores data. Deleted when browser/tab is closed.
* Same getters and setters as local.

### IndexedDB

* One or more databases, each with one or more object stores.
* NoSQL
* Records as key-value pairs.
* Creates indexes on object stores.

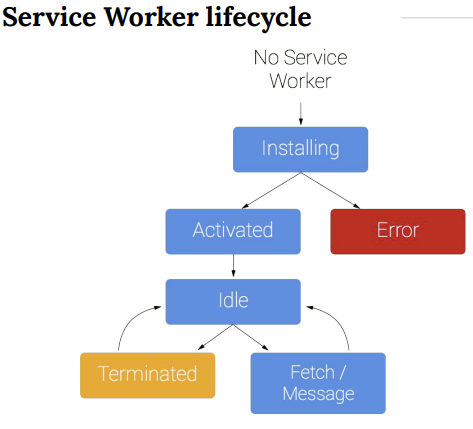
### Cache

* Stores pairs of request/response objects.
* Can be hundreds of MB in size.



### Progressive Web Application

* Web applications designed to appear ‘installed’ as native applications.
* Can go from developing two things in parallel to one app i.e. web site and native app to one progressive web app.
* Technical definitions of a PWA:
  + Originates from a secure origin.
  + Loads while offline.
  + References a Web App Manifest.
* Service Workers – JS that runs in the background.
  + Can be used for long-running processes.
  + Must be started by a web page.
  + Allows websites to run offline by serving cached data.
  + Must be served over HTTPS.



* Data storage in a PWA: Web storage, IndexedDB, Service Workers and cache.

# Lecture Note Set 12 – Front End Testing

### Bundling

* Effectively compiling the web code into a form the browser can consume.
* Usually prefer (result of bundling):
  + Fewer network requests (fewer files).
  + Smaller and/or parallel network requests.
  + No network requests at all (caching)

### Simple Bundling (fewer files)

* Concatenate all .js files to one big file.
* Same for .css files.

### Simple Bundling (smaller files)

* Browsers don’t care about function and variable names, so shorten all var and functions.
* Makes it harder to read, but more optimal to run.

### Lazy Loading

* Break SPA into chunks and only download what is needed e.g. login, settings.

### Automated Testing

* Challenges for testing web apps:
  + Different browsers (Chrome, Firefox etc.)
  + Different browser versions.
  + Different HTML, JS versions.
  + Differences in APIS.
  + Differences in the handling of the DOM, JS etc.
  + Differences in libraries and versions.
* Advantages of automating testing:
  + Humans don’t have to test manually.
  + More frequent testing.
  + Quick and regular feedback.
  + Finding defects missed by manual testing.
* When not to automate:
  + UI changes rapidly.
  + Server API changes rapidly.
  + Tight timescales – no time to develop tests.
* WebDrivers
  + Enables web devs to write an automatic user agent to test. Can be used to discover and manipulate web elements in web documents.
  + Mostly for automated testing.
* Selenium
  + Automates a browser for testing, executing routine tasks etc.
  + Please automate your tests on different browsers.