**ITTWEB dispositioner**

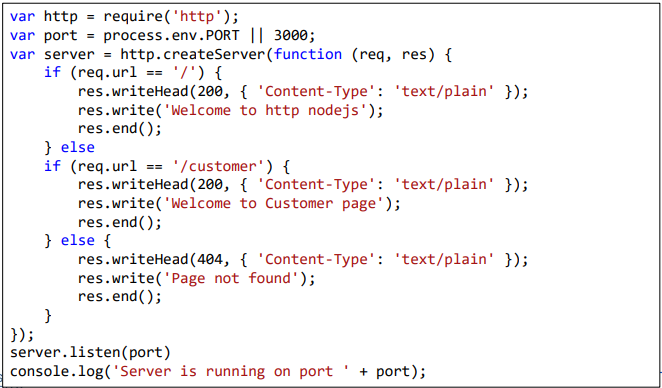
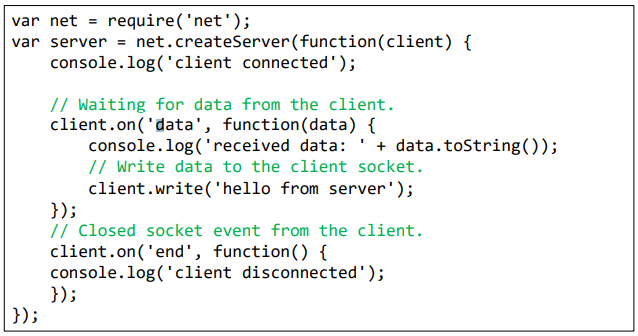
## Spørgsmål 1

Redegøre for hvordan Node.js kan bruges til at lave en Web server. Og redegør for hvordan Express kan bruges til at strukturerer programmet, så det følger principperne for et server-side MVC framework. Redegør for hvorfor man behøver en template engine, og vis eksempler på brug af denne.

Samt vis hvordan man kan designe og implementere en webløsning, som omfatter persistering af data i en dokument database med anvendelse af MongoDb.

**Hvad er node.js?**

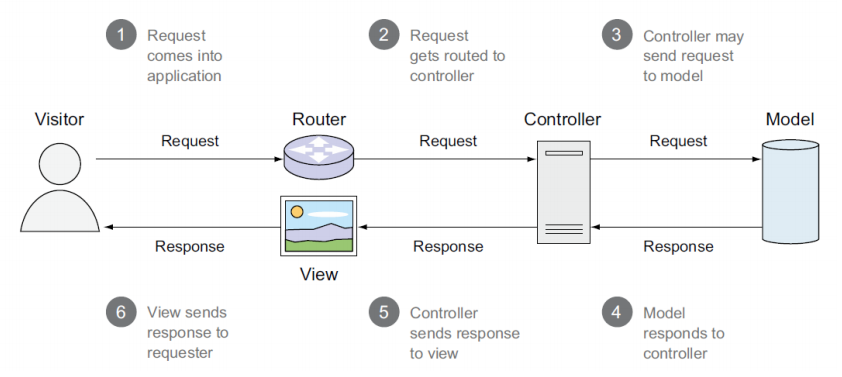
* Open source, cross platform runtime environment for serverapplikationer
* Skrevet I Javascript
  + Godt, så man kan have samme sprog for både server og client
* Er event-driven og async, så det henvender sig godt til realtime applikationer
* Kører på googles V8 engine, og består af nogle c++ biblioteker og et c
  + Libuv er et c bibliotek -> det er her de asynce funktioner er
  + Her event loopet også er
* Npm -> node package manager
  + Tool til at installere moduler ”*npm install moduleName*”
* **Vis eksempel 1**

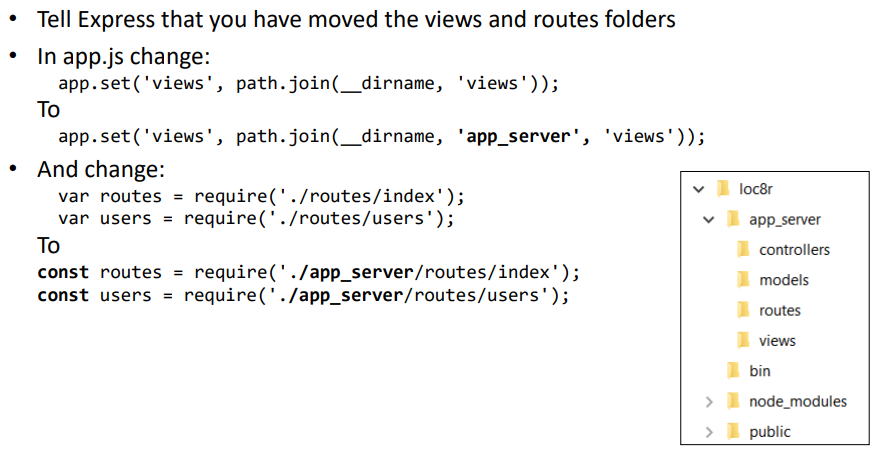
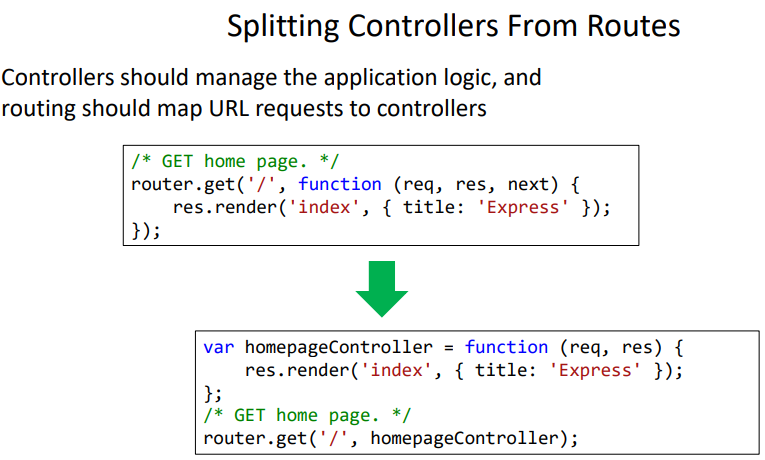
****

*Tcp – socket page handling*

**Express (app\_server folder)**

* Web framework for Node
  + Bruges til routing og middleware, men uden den store funktionalitet i sig selv
  + Bruges også til at organisere sin webapp til MVC arkitektur
  + Det hjælper dig med at styre alt, fra routes til requests og views
* ”*Npm install express*”
* **Vis hvordan Express kan blive til MVC:**





**Template engine**

* Det er tidskrævende og error-prone at skrive raw strings ind i html kode
  + Derfor har man lavet template engines
* I TE’s giver udvikleren en html fil, som har placeholders der kan blive fyldt runtime, så man ikke behøver at hard-code fx strings ind
* Templating engines kan altså bruges til at indsætte dynamisk data ind i views
* Det gør så man separerer data fra præsentationen fra præsentationen af dataen
* Der er forskellige slags template engines
* **EJS (index.ejs)**
  + ****
* **PUG**
  + ****

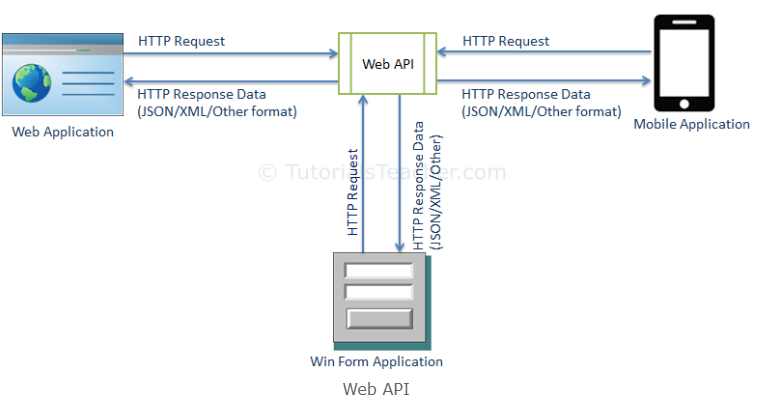
**MONGODB**

* **VIS EKSEMPEL 2 (indexcontroller.js)**

## Spørgsmål 2

Redegør for og vis eksempler på hvordan man kan designe og implementere et Web API med Node.js inklusiv persistering af data i en database. Redegør for principperne i et RESTful Web API. Og redegør for fordele og ulemper ved et RESTful Web API i sammenlignet med et GraphQL baseret API.

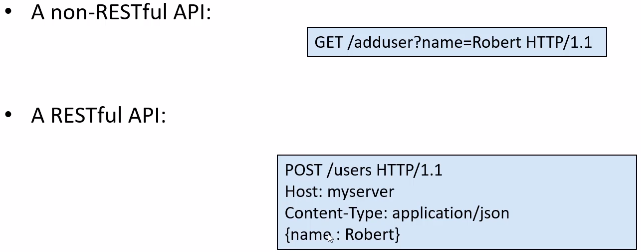
**Hvad er et Web API?**

* Et Web API er en web server som returnerer data i stedet for html
* Dette kan gøre det muligt for mange forskellige klienter at interagere med det
  + Kunne være en mobile devices, browsers og traditionelle desktop applikationer
* 
* Fx kan man vha express nemt sende data i form af JSON tilbage
  + Res.json({”msg” : ”fail”})
* Typisk vil man returnere json eller xml
  + Dog kan man i princippet returnere hvilken som helst data
* Der er 2 **key** komponenter i et respons
  + http status koden og det returnerede data
* Det er altså et API som kan tilgåes via http over nettet
* **VIS EKSEMPEL**

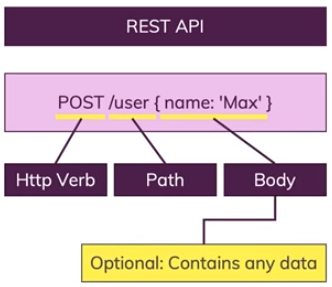
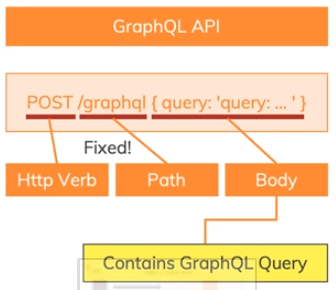
**Design**

* Man laver en app\_api folder som indeholder
  + Routes, controllers og models som er specifikt til API
* Så inkluderer man ruterne i app.js
  + 

**RESTful (REPRESENTATIONAL STATE TRANSFER)**

* Et RESTful (REST api) er et api som bruger http requests til CRUD operationer
* REST er i et set af arkitektur principper, hvor du udfra disse kan designe en web service som har fokus på et systems ressourcer, og hvor ressourcer er sendt over http
  + Ergo er det ikke en standard, men nogle retningslinjer at designe udfra
* **En konkret implementering af et REST api følger syv design principper:**
  + **Eksplicit bruge http metoder** 
    - Brug POST for at oprette ting på serveren
    - GET for at få ting fra serveren
    - PUT for at ændre
    - DELETE for at fjerne
    -  **(api/routes/todolistRoutes)**
  + **Stateless (todolistcontroller.js)**
    - Serveren skal ikke holde information om klienterne
    - Lageret har staten, ikke serveren
    - Når serveren får et request, loader den fra lageret (databasen) og sender det repræsentationen tilbage til klienten
  + **Expose directory structure-like URIs (todolistroutes.js)** 
    - Eksempelvis example.com/api/users/1
  + **Transfer XML, JSON eller andre media typer (todolistcontroller.js)**

**RESTful vs GraphQL**

* GraphQL har ét endpoint som er et POST/graphQL hvor REST har mulitple (http metoder)
* Man bruger JSON data i begge api som standard
* Man kan også bruge alle server-side sprog, og samme frontend framework på begge
* Begge er samme stateless, de har ingen data fra klienten
* Forskellen er **hvordan** man sender requests imellem dem
  + REST er URL-driven hvor GraphQL er Query-language
  +  vs. 
* Derfor kan gøre flere ting i ét request, hvor REST engang imellem skal bruge flere requests
  + Det er querien der sørger for hvad der skal ske
  + I en query er der en **operation type**, som siger ”hvilken operation” der skal køres
  + Et **endpoint** som kunne være en ’user’ som ressource du vil arbejde med
  + Og **fields** man vil arbejde med

## Spørgsmål 3

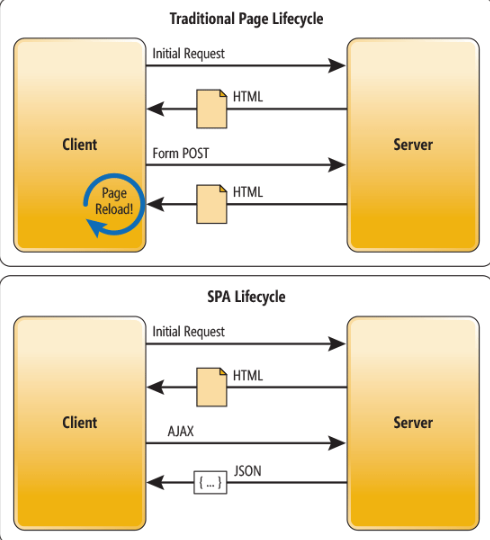
Redegør for og vis eksampler på hvordan man kan designe og implementere en single-page applikation med brug af Angular version 7 eller nyere.

**Angular**

* Angular er et open source javascript client side MVC framework lavet af google
* Det er et rewrite af AngularJS, og bliver nu skrevet i typescript, dart eller javascript
* Mange web apps har meget client side kode, som man gerne vil have struktureret i MVC
  + Angular er et komplet framework til dette

**SPA**

* Single page applikationer er applikationer hvor klienten kun requester html én gang
  + Herefter er det bare data til at opdatere de forskellige komponenter i viewet
* SPA sikrer altså dynamiske views, hvor man ikke hele tiden behøver at opdatere siden
  + Det giver en slags desktop-app følelse i en web-app

****

**Angular CLI**

* Command line interface
* Hjælper med at lave skelet til angular
* **Oprette projekt:** ng new myapp
* **Køre server til host**: ng serve
* --minimal flag generer mindre og simplere projekter, hvor tests ikke er der og der er inline styles og inline templates

**Angular Modules**

* Angular apps er modulære og har sit eget modulsystem (NgModules)
* Det er en klasse som har en NgModule decorator funktion
  + Den identificerer sine egne komponenter, direktiver og pipes
  + Nogle items kan være public, så eksterne komponenter kan bruge dem
* Alle apps har mindst ét modul som er AppModule der er root modulet

**Angular Components**

* Det vigtigste aspekt ved Angular er komponentbaseret udvikling
* Et komponent i angular følger web components standard (i modsætning til react)
* Komponenter bliver placeret på websider med custom elements
  + Fx HTML5 Audio og video (<video> <audio>)
* **Component.ts** -> typescript fil som indeholder selector id, templateurl og styleurl
  + Også her logikken er til componentetn
* **Component.html** -> her er htmlkoden til componenten
  + Alt kører over databinding fra .ts filen! {{}}
  + ****
  + Two-way = [(ngModel)]

**Angular Services**

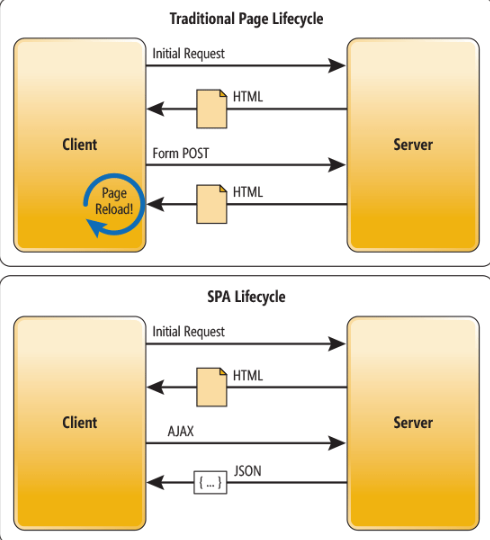
* En service er en klasse uden et ui
* Service klasser skal have @injectable() decorator
  + Sådan den kan injiceres i de komponenter som skal bruge servicen
* Bruges pga single responsibility principle
  + Komponenter skal ikke fetch og gemme data direkte
  + De skal kun præsentere data og delgere data adgang til en service
* Services kan dele data mellem komponenter og klasser som ikke kender hiannden
* Kan genereres ved **ng g s**

## Spørgsmål 4

Redegør for og vis eksampler på hvordan man kan designe og implementere en single-page applikation med brug af React version 16.2 eller nyere.

**SPA**

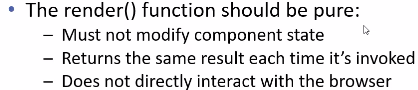
* Single page applikationer er applikationer hvor klienten kun requester html én gang
  + Herefter er det bare data til at opdatere de forskellige komponenter i viewet
* SPA sikrer altså dynamiske views, hvor man ikke hele tiden behøver at opdatere siden
  + Det giver en slags desktop-app følelse i en web-app

****

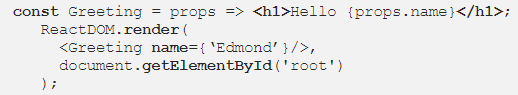
**React**

* React er et JS library til at bygge UI
* Det er lavet og bliver maintained af facebook
* React er kun et UI library, men hvis man kombinerer det med Redux og React router ender man op med et komplet framework, meget ligesom Angular
* Man bruger react til at bygge komponenter som man kombinerer og indlejrer for at bygge en komplet webApp

**React Components**

* React komponenter er simpelt set bare en JavaScript klasse eller funktion, som accepterer inputs (props) og returnerer et React element, som så sørger for hvordan UI skal fungere
* **Class components (stateful)** 
  + Komponeter man definerer I en klasse med en state
  + Smarte fordi der kan være logik i dem
  + De holder state
  + Container, da de kan holde andre (funktionelle) komponenter
  + Har render funktion, og extender React.Component
  + ****
* **function components (Stateless)** 
  + Inline komponenter som egentligt bare skal præsentere og output UI Elementer
  + De holder ikke state
  + Er i princippet bare funktioner

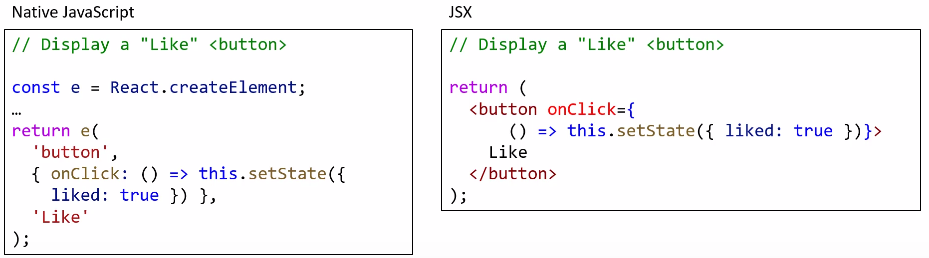
**Props**

* Gør komponenter dynamiske og customizable
* Giver mulighed for at pass data fra et komponent til et andet, normalt fra parent til child
* Props er read only, og må ikke modificeres af de komponenter de er passed til
* 

**Shadow DOM**

* I stedet for at re-render hele DOMet når man opdaterer noget, så har man noget som hedder Shadow DOM
* Man injekter ikke sine komponenter i DOMen med det samme, men kan kører det ind i Shadow DOM
  + Her finder react ud af, hvor der er blevet lavet ændringer, og så sætter den KUN ændringerne i Shadow DOM ind i den rigtige DOM
* Det gør react meget hurtigt, da alt bliver gjort på den virtuelle dom og kun opdateres når der kommer ændringer ind i den rigtige DOM

**JSX**

* Et sprog så man kan skrive inline HTML i javascript
* Ligner meget HTML men giver mulighed for at strukturer komponent rendering
* JSX er meget brugt i return statements i komponenters render funktion
* ****

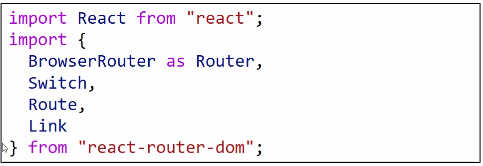
**States**

* Da props aldrig må skifte, bliver vi nødt til at have noget som kan mutate nogle værdier i en komponent
  + Dette er states, her kan du skifte værdier
* States er private, og fuldt kontrolleret af komponenten
  + De kan dog blive passed til children som props
* Kun tilgængelige i component classes, da de skal initializes i constructoren

**Forms**

* I princippet er der intet dedikeret komponent for at lave forms i react
  + Intet specielt api til det
* Man bruger komponenter, state, props og events til at lave forms
* HTML form elements holder automatisk intern state
  + Her bruger man **controlled components**
* Elementer som <input> og <select> holder intern state, og opdaterer baseret på brugerinput
* I React er det kun states som bliver opdateret via setstate
* Her kan man kombinere dem , så REact staten er den state der trumfer
  + React komponenten vil også kontrollere hvad der sker i fx et user input
* Hvis man laver sådan et element som bbliver trumfet af en React state kalder man det et **kontrolleret komponent**
  + Man binder fx et <textarea>s value til en state i komponenten, så den bliver styret derfra
* **Ukontrollerede komponenter** 
  + Brug ref til at få form værdier fra DOM
  + I komponenten: this.input = React.createRef()
  + I html: <input type=”text” ref={this.input}>

**Routing**

* Er ikke I react som standard
* Kan installeres med npm i -A react-router-dom
* Her får man: **Routers, Route matchers og Navigation**
* Import: 
* ****

**Redux**

* Er en state container for js apps
* Man kan bruge den til at opretholde states rundt i sin applikation
* Hjælper med at give hver react komponent den præcise state den behøver
* Redux holder staten indenfor en enkelt lokation

## Spørgsmål 5

Redegør for og vis eksempler på udviklingen af en webløsning med responsivt design, samt anvendelse af CSS preprocessors og build tools.

Redegør for og vis eksempler på test af en React App med brug af Jest.

**Responsive design**

* Formålet er at lave den optimale ”viewing experience”
  + Ting skal være letlæselige, med minimum resizing, panning og scrolling
  + Skal virke på alle devices, telefoner og computere osv.
  + Er blevet relevant efter at smartphones nu udgør halvdelen af webtrafikken

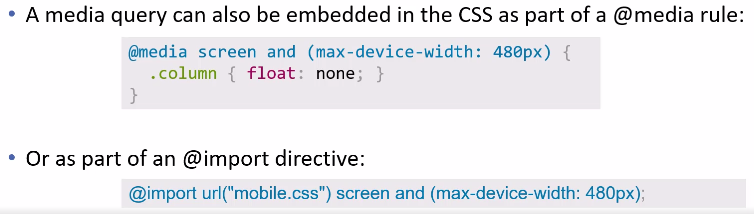
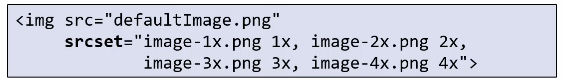
**Guidelines**

* Websider skal kunne render læseligt til alle opløsninger
* Man skal have kun ét sæt af markup, css (container) som skal tilegne sig alle devices
* Der må aldrig være en horisontal scrollbar

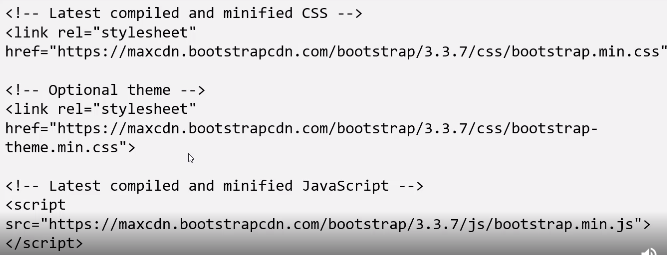
**Responsive vs. Adaptive**

* De sigter begge efter at optimere brugeroplevelsen ved at optimere ”viewing experience”
* Responsive bruger kun et layout til at adapt til skærmstørrelser
* Adaptive har forskellige layouts til forskellige størrelser

**Hvad kan RWD indebære?**

* **Liquid layout** 
  + I stedet for at bruge width bruger vi max-width , og i stedet for height bruger vi min-height
    - Height så multi-line text ikke går ud af containerens boundaries
* **Viewports** 
  + Mobil devices browser opløsning er mindre end de fortæller, så derfor ender man tit med en horisontal scrollbar
  + ****
* **Media queries**
  + Vi skal adapt, og der kan man bruge media queries
  + Kan gøres ved at bruge media option til at lave conditionals til at bestemme hvorvidt det skal være det ene eller andet
  + ****
  + ****
* **Img srcset**
  + ****

**Værktøjer til RWD**

* **Bootstrap**
  + Er et CSS og javascript ui framwork
  + Kan nemt og effektivt skalere websider med en enkelt kodebase
  + Ikke kun for at lave responsivt design, men også generelt til bare at få det til at se godt ud
  + Kan installeres fra hjemmeside, npm ellers kan man referere dem via CDN
  + 
  + **Hvordan bruger man bootstrap?**
    - Alt skal være i en CSS Container eller container-fluid
    - Inde i containeren bruger man rows og columns
      * Man bygger sit views op med rows
    - Bootstrap leverer en masse komponenter som man kan genbruge og bruge
* **FlexBox**
  + Er en CSS3 layout metode lavet til et-dimensionelle layouts
  + Det er et fleksibelt og float free css layout, som kan bruges til forskellige skærmstørrelser og display devices
  + Giver muligheden for at bruge alt den plads der er tilgengælde på skærmen
  + **Flex container** er det boxelement man laver, og childrens eller elementerne derinde er **flex items**
  + **To axis:** cross axis (ned) og main axis (hen)
  + **Flex direction:** hvilken axis den skal køre efter row, row-reverse, column, column-reverse
  + **Flex wrap:** kontrollerer om det er single line, multiline, eller hvordan nowrap, wrap, wrap-reverse
  + **Justify-content:** Definerer hvordan flex items skal ligge på en linje (center, space between osv).
  + **Align-items:** Definerer hvordan ting er lagt på den modsatte axis (cross axis)
  + **Alignself:** man kan override et specifikt flex-item ved at sætte alignself på den specifikke item
  + **Order**: man kan give flex itmes order property med en value, derved ved browseren display efter den værdi
  + **flex-grow:** Definerer om et item skal gro hvis der er rum
  + **flex-basis:** definerer default width på hvert flex item
  + **Flex shrink:** definerer om et flex item må shrink hvis der er plads
  + **Flex:** ^kombination af de 3

**CSS-Grid-Layout:**

* Normale html sider uden CSS følger ”normal flow” block direction nedad, inline direction henad
* Væk fra normalt flow: **Floats** er brugt til at skifte en boks til højre eller venstre, så content bliver wrapped omkring. **Positioning:** er brugt til at skifte placering på et element fra dets normale flow, dog uden content-wrap
* **CSS-Grid**: Definerer et to-dimensionelt grid-based layout system, optimeret for ui
* **Grid container**: det element man sætter display: grid på
* **To akser:** inline og block

**CSS-preprocessors**

**SASS**

* + **Syntactically Awesome Stylesheets**
  + Er et scripting sprog som compiler til CSS
  + Er normalt kodet i Dart
  + Den nye syntax bruger block-formatting som CSS, den gamle brugt indented ligesom Python fx
  + Man kan definere variabler i SCSS, arithmetics osv
    - De kan også blive redefineret
* **Mixins:** er en slags funktion ,som man kan referere
  + Man definerer den og inkluderer den senere
  + Så kan man genbruge koden mange gange, i stedet for at skulle paste hele koden
  + Man kan også bruge argumenter i mixins, så man kan lave dem dynamiske
* **Extends:** man kan extend styles (se eks)

**Build tools**

* **Gulp**
  + Skal optimere arbejdsgangen ved at køre taskrunners
  + Automatiserer og enhancer ens workflow
  + **Nemt at bruge**, gulp holder alt simpelt
  + **Effektivt**, hurtige builds da det bruger node streams
  + **Nemt at lære:**
* **Webpack (slags taskrunner, bygge bundles):** Er en slags taskrunner, men det er ikke det specifikke formål. Det er mere beregnet til at bygge de her module bundles (er en javascript fil som indeholder aktiver, der tilhører sammen og skal serveres til klienten)
  + **Bruges** af Angular og React
  + **Brugbart for klient:** Tager alt koden fra applikationen og gør det brugbart i browseren
  + **Ng build:** Sker når man fx kører ng build
    - **Byggeprocessen** (som udføres af Webpack), presser mange af de her forskellige filer ind i en enkelt fil kaldet en bundle
    - **5 bundles, Default** **config** for angular CLI outputter 5 bundles
      * Runtime (angular runtime)
      * Main (Applikationskoden)
      * Polyfills (Bruges til browserkompatibelitet)
      * Styles (Alt styling i appen)
      * Vendor (Angular og andre biblioteker der bruges)
  + Gør det nemmere at fx **debugge** og **test** koden

**React test med Jest**

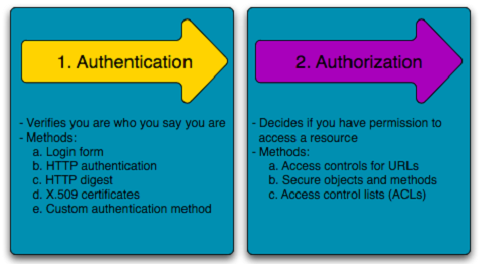
* **Jest** 
  + Jest er et testing library lavet af facebook, og det kan også blive brugt som test runner
  + Det er godt suited til react apps, men kan bruges til alle former for js kode
  + Man vil ikke have sine unit tests koblet til andre dele af koden for at depend, så derfor kan man bruge Jest til et **mocke** dele af koden ud
  + Jest kommer også med et **assertion library** som er bygget ind
  + Jest har også indbygget **coverage tools** som bruger et tool kaldet istanbul

## Spørgsmål 6

Vis hvordan man kan implementere authentication og authorisation i en webløsning. Vis hvordan man kan bruge JWT til authentication i et Web API.

Redegøre for hvordan man kan publicere webløsninger til driftsmiljøer, her under også cloud baseret hosting.

**Authorization vs. Authentification**

* Authorization er retten til at tilgå noget bestemt. Det kunne være retten til at tilgå noget data.
* Authentification er at bevise at man er hvem man er; altså at bekræfte ens identitet
* 

**Authentication**

**HTTPS**

* http over SSL, eller nu over TLS
* det er en ganske almindelig http connection men over en enkrypteret TLS connection
* https er med port 443 normalt, hvor http er port 80
  + protokollen er det samme, den er bare wrapped i TLS
* alt er krypteret, request url, query parametre, headers, cookies og body
  + udover IP adressen
* Serveren skal have et private, public key pair
  + Private skal blive hos serveren

**Passport**

* En npm package som er lavet til at gøre authentification i Node nemt
* En af key styrkerne ved det er, at den har en masse forskellige metoder til authentication, kaldet **strategier**
  + Facebook, google, twitter, oath, local
* Egen brugerdatabase -> passport-local == npm install passport npm install passport-local
* Man laver et **passport.js -> passport.use**
* **LocalStrategy ->** her kan vi bestemme hvilke fields man skal bruge til login
* Så laver man en funktion af passport.use til at validere
* App.js **-> require passport, require strategy, initialize passport**
* Man kan bare bruge **isAuthenticated** fra Passport for at lave authorisation
  + Man kan også bruge ensurelogin

**Jwt for APIer**

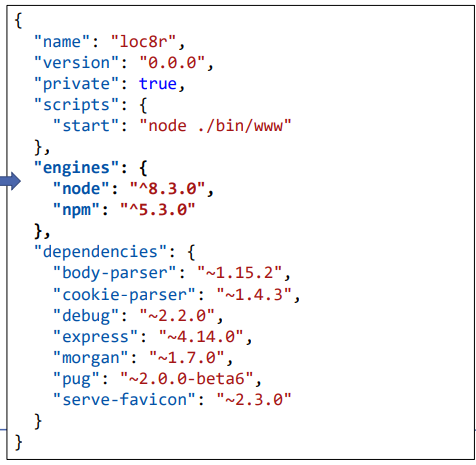
* Cookies virker ikke med alle klienter
* No sessions while Restful og ikke cookies
* Derfor kan man bruge JWT i Restful APIer
* Det er en json-based open standard
* **1.** brugeren poster sine credentials til serveren via API  
  **2.** Serveren validerer disse i databasen, og returnerer et token til browseren  
  **3.** Browseren gemmer det her token til reuse
* Da dette ikke er i en cookie skal man gemme det på klienten, typisk i localstorage
  + Derved sender man det med i en authorization bearer i sin header hver gang
  + Browser software til at validere brugerens token

Sign jwt (**fitness backend -> app\_server -> controllers -> api -> user.js**)

Store jwt localstorage (**fitness angular -> src -> app -> Apiservice**)

**Heroku**

* Installer Heroku CLI, for at manage og skalere ens app
* Så logger man ind på sin heroku login i CLI
* Nu skal man gøre sin app klar til heroku, og det kan man gøre ved at lave engines i sin package.json fil, for at fortælle heroku om hvvilke versioner ens app kører med i npm og node

****

* Så skal man skrive i sin Procfile, hvilken command der skal bruges for at starte ens app (npm start fx)
* Alle filer skal desuden være i et lokalt gitr epo, for at man kan deploy det til heroku
* Nu kan man bruge **heroku create** som så gør at heroku gør sig klar til ens kode, og at heroku genrerer et navn for ens app
* Nu kan man bruge **git push heroku master** for at deploy til heroku og **heroku ps:scale web=1** for at få en instans kørende. Til sidst kan man åbne den med **heroku open**

## Spørgsmål 7

Redegør for konceptet Progressive Web Apps og vis hvordan man kan implementerer en Progressive Web App. Redegøre for hvordan man kan optimere performance for en webapplikation. Vis hvordan man kan implementere to-vejs data kommunikation ved brug af WebSocket.

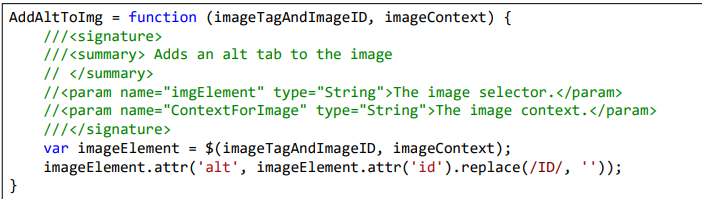
**Progressive web apps**

* Definitionen er, at det skal kunne bruges af alle brugere uanset browser og platform. Der er nogle guidelines eller follow-throughs man skal igennem før ens app er en PWA.
* **1. Progressive –** skal virke for alle brugere, uanset browser og enhed  **2. Responsive –** Skal fit alle form faktorer: desktop, mobil, tablet osv  **3. Connectivity independent –** skal virke offline **4. App-like –** skal føles som en app **5. Fresh –** skal være up to date **6. Safe –** skal være under HTTPS  **7. Discoverable –** Search engines skal kunne finde din app, så det kan indekseres på google fx  **8. Re-engageable –** Skal bruge nye features som fx. Push notifications og 2-vejs kommunkation  **9. Installable –** Brugere skal have muligheden for at installere appen på sit device **10. Linkable –** Da det er en webapp, skal brugere have muligheden for at linke appen

**Hvordan bygger man?**

* Bygger et **app shell** som skal indeholde alle static ting som billeder, markup osv.
* Bruger **service workers** til at gøre appen klar til offline brug. Service workeren vil koble sig på ved første login på appen, og derved cache ting i klienten. (**public/service-worker.js**)
* **Service workeren** skal registreres når klienten går ind på appen (**index.html linje 51**)

**Performance**

* **N second rule** – hvor lang tid brugere er forberedt på at vente på at en webpage loader og render
  + 10-second 1997
  + 8-second 2001
  + 4-second 2006
  + 3-second 2010
* Hurtigere webapps -> mere omsætning (amazon)
* 80-90% af end user response tiden bliver brugt på front-end delen
  + Start her (golden rule)
* **De 14 performance regler**
  + **1.** Make fewer HTTP requests   
    **2.** Use a CDN   
    **3.** Add an Expires header   
    **4.** Gzip components   
    **5.** Put stylesheets at the top   
    **6.** Put scripts at the bottom   
    **7.** Avoid CSS expressions   
    **8.** (Make JS and CSS external )   
    **9.** Reduce DNS lookups   
    **10**. Minify JS and CSS   
    **11.** Avoid redirects   
    **12.** Remove duplicate scripts   
    **13.** Configure ETags   
    **14.** Make AJAX cacheable
* **Brug lighthouse**
  + **Eller Yslow**
  + **Eller webpagetest**
* **Caching**
  + **Cache control**
    - sørg for at sætte et ordentligt time limit på de cachede ting, som rent faktisk ikke skal valideres af serveren ofte. Kan gøres via cache control headeren.
  + **Conditional GET** 
    - Sender et request i stedet for bare at spørge efter pagen, og her spørger den om siden er blevet opdateret siden sisdt, hvis ikke, så opdaterer den ikke.
* **Minification**
  + Laver kodeoptimeringer, så funktioner og variabelnavne bliver så kort som muligt
  + ****
  + ****
* **Bundling**
  + Gør loadtime bedre ved at reducerer antallet af requests til serveren

**To-vejs kommunikation med websockets**

* **Websockets generelt** 
  + Protocol som giver mulighed for full-duplex kommunikation mellem kanaler over en enkelt TCP forbindelse
  + De er designet til at blive implementeret i web browsers og web servers, men kan i princippet blive brugt på hvilken som helst klient eller server applikation
  + Websocket endpoints URLs starter med **ws** i stedet for http
  + Default tcp port 80 eller 433 hvis wss
* **Hvorfor websockets?** 
  + Serveren kan sende beskeder til klienten
  + Websockets kan traverserer firewalls og proxies
  + De kan bruges til streams af beskeder ovenpå tcp
* **How-to** 
  + En websocket starter som en http eller https forbindelse, og skifter til **ws** efter handshake (**client/src/highscore/highscore.js)**
  + Browseren sender et request og forespørger om at skifte protocol fra http til ws
    - Dette sker gennem ”upgrade headeren”
  + Serveren accepterer, og svarer tilbage med update header også
  + Det er her at det bliver til en ws forbindelse i stedet for http, over den underlæggende TCP forbindelse
  + Når forbindelsen er established kan client og server sende websocket adta eller text frem og tilbage i full duplex **(app.js)**