**Projektuppgift**

*DT162G*

**Javascriptbaserad webbutveckling: Projekt**

AI Datorer AB:s MERN-stacksbaserade Intranät

**Max Karlstedt**

**MITTUNIVERSITETET  
Avdelningen för informationssystem och -teknologi**

**Författare:** Max Karlstedt, [maka2207@student.miun.se](mailto:xxxxx0000@student.miun.se)  
**Utbildningsprogram:** Webbutveckling, 120 hp  
**Huvudområde:** Datateknik  
**Termin, år:** HT, 2023

# Sammanfattning

**Syfte:** Webbprojektet avser AI Datorer AB:s MERN-stacksbaserade Intranät i syfte att kunna inventera sitt lager av datorkomponenter såväl som systemadministratörens förmåga att administrera behöriga användare med i sin tur olika behörigheter till det MERN-stacksbaserade Intranätet. **Teori:** MVC-arkitekturen är en vanlig strategi inom webbutveckling för att separera på vad för slags data ska visas, för vilka användare och under vilka omständigheter. MERN-stacken står för MongoDB, ExpressJS, ReactJS och NodeJS och är en teknikstack där MongoDB är databasen, ExpressJS + NodeJS är själva webbservern och ReactJS är den webbklient som konsumerar webbserverns erhållna databasdata. MongoDB är en icke-relationell databasform där databaser innehåller kollektioner vilket i sin tur innehåller dokument där data lagras i JSON-liknande format. NodeJS möjliggör serverbaserad JavaScript där ExpressJS är ett abstraktionslager för att underlätta JavaScript-baserade servrar vilket även kan kommunicera med MongoDB-databaser. ReactJS är ett JavaScript-baserat frontend-funktionsbibliotek för att rendera dynamiskt erhållna data i webbläsare. TailwindCSS är ett CSS-ramverk där klassnamn skrivs vilket i sin tur ersätts med faktiska CSS-regler vid rendering i webbläsare. REST API möjliggör ömsesidig kommunikation mellan mjukvaror där förfrågningar och svar kan skickas och där erhållen data ej behöver vara beroende på något tidigare tillstånd. **Metod:** Microsoft Visual Studio Code används som utvecklingsmiljö, MongoDB Compass och MongoDB-server lokalt används för databasimplementeringen, och publiceringen äger rum i form av länkad GitHub-repo med instruktioner om hur det körs lokalt. **Konstruktion:** MongoDB består av två kollektioner: pccomponents och users. Servern börjar med att konfigurera vad för slags data (JSON, URLENCODED) som kan hanteras och sedan skickar den vidare till lämpliga rutters där "mellankod" (eng. "*middlewares*") hanterar databasanslutning, inmatningsvalidering, och rätt slags CRUD-förfrågan. Behörighetskontroller sker både på klientsidan och på serversidan så lyckad förändrad erhållen JWT kommer ej förbi de extra databaskontrollerna. **Resultat:** De redogjorda skärmdumparna visar att AI Datorer AB:s kravspecifikationer har uppfyllts till fullo. **Slutsatser:** Beklagande är att trots alla fördelar med MongoDB så är det inte gratis att implementera - med fullständig funktionalitet - på egna servrar utan då krävs en kommersiell versionsimplementering. ReactJS är kraftfullt i att förenkla data som visas i användargränssnittet samtidigt som det kan vara utmanande att hantera globala tillstånd (eng. "*states*") samtidigt som komponentbaserade tillstånd. Källförteckningen avslutar det hela.

# Innehållsförteckning

[1 Introduktion 1](#_Toc156222878)

[1.1 Bakgrund och kunden 1](#_Toc156222879)

[1.2 Kundens målgrupp 1](#_Toc156222880)

[1.3 Kundens budget 1](#_Toc156222881)

[1.4 Juridiska överväganden 1](#_Toc156222882)

[1.5 Överenskommen kravspecifikation 2](#_Toc156222883)

[1.6 Kapitelöversikt 3](#_Toc156222884)

[2 Teori 4](#_Toc156222885)

[2.1 MVC-arkitekturen 4](#_Toc156222886)

[2.2 MERN-stacken 4](#_Toc156222887)

[2.2.1 MongoDB 5](#_Toc156222888)

[2.2.2 ExpressJS 5](#_Toc156222889)

[2.2.3 ReactJS 6](#_Toc156222890)

[2.2.4 NodeJS 7](#_Toc156222891)

[2.3 Tailwind CSS 7](#_Toc156222892)

[2.4 REST API 8](#_Toc156222893)

[3 Metod 9](#_Toc156222894)

[3.1 Integrerad utvecklingsmiljö 9](#_Toc156222895)

[3.2 Publicering 9](#_Toc156222896)

[4 Konstruktion 10](#_Toc156222897)

[4.1 MongoDB, REST API, NodeJS & ExpressJS 10](#_Toc156222898)

[4.2 ReactJS och TailwindCSS 13](#_Toc156222899)

[5 Resultat 17](#_Toc156222900)

[5.1 Registrering, inloggning och utloggning 17](#_Toc156222901)

[5.2 Hantera datorkomponenter 17](#_Toc156222902)

[5.3 Hantera användare 17](#_Toc156222903)

[6 Slutsatser 18](#_Toc156222904)

[Källförteckning 19](#_Toc156222905)

# Introduktion

Projektrapporten inleder med att berätta om kunden, deras bakgrund, målgrupp, budget (det vill säga, avgränsningar i vad som ska levereras), juridiska överväganden, samt den slutgiltiga överenskomna kravspecifikationen mellan parterna. Avslutningsvis under introduktionskapitlet presenteras de resterande kapitlen i form av en kapitelöversikt.

## Bakgrund och kunden

AI Datorer AB bedriver försäljning av datorkomponenter online sedan ett par år tillbaka. Företagets hantering av sina datorkomponenter sker dock mycket ineffektivt då data om datorkomponenterna är lagrade på ostrukturerade vis. Således är bolaget i behov av ett underhållbart intranät vilket i sin tur är lätt att förvalta för framtida ansvariga inom bolaget.

Bland annat önskar kunden att via en ny slags databas kunna lägga till, ändra, radera såväl som utläsa diverse verksamhetsnödvändig data om datorkomponenterna (namn, beskrivning, pris, antal, status huruvida de är begagnade eller nya, samt förknippade bilder till komponenten ifråga) de har till salu.

Även möjligheter att - via en så kallad systemadministratör - hantera användare (skapa nya, samt utläsa/radera och/eller ändra befintliga) och deras behörigheter att kunna sköta arbetet (inventariehantering av datorkomponenterna) via intranätet önskas av kunden. Med "användare" menas anställda vid bolaget AI Datorer AB.

## Kundens målgrupp

I och med att det rör sig om ett intranät åt bolaget så är målgruppen deras anställda såväl som den ansvariga systemadministratören vilket kommer att ha den högsta nivån av behörigheter i det färdiga MERN-stacksbaserade Intranätet.

## Kundens budget

Kunden är redo att lägga en del på denna beställning då det handlar om att kunna sköta inventeringen av datorkomponenter såväl som administrera ansvariga anställda att kunna göra det via det MERN-stacksbaserade intranätet.

## Juridiska överväganden

Kunden har gett samtycke till uppdragstagaren att ta fram en MongoDB där alla deras nuvarande data om datorkomponenter såväl som dess anställda kommer att överföras till.

## Överenskommen kravspecifikation

Följande har kunden kommit överens om att det ska implementeras i detta MERN-baserade webbprojekt:

* En backend med hjälp av NodeJS, ExpressJS och MongoDB där databas i MongoDB kommer att innehålla information om datorkomponenter och bolagets anställda. Här ska det gå att "CRUD:a" (skapa, utläsa, uppdatera, och radera) all data.
* En frontend med hjälp av ReactJS, TailwindCSS och ett JavaScript-baserat REST API som konsumerar denna backend så att bolagets anställda samt dess utnämnda Systemadministratör kan logga in i intranätet för att "CRUD:a" nödvändiga databasdata.
* Besökare vid intranätet ska kunna registrera sig vars konton sedan måste aktiveras av Systemadministratören innan de registrerade användarna kan logga in för att se något från intranätets databasdata.
* Endast en Systemadministratör med användarnamnet sysadmin kan registrera sig och får då den dedikerade rollen som Systemadministratör för intranätet.
* Registrerade användare kan först endast utläsa datorkomponenter medan övriga behörigheter måste läggas till manuellt av sysadmin. Behörigheter som går att få för en given användare är: utläsa, skapa nya, ändra och/eller radera befintliga datorkomponenter och/eller dess bilder. Med andra ord kan en användare - om så önskas - få behörigheterna att endast ändra bilder i datorkomponenter men inte något annat i datorkomponenterna eller ändra dem på något annat vis (radera, ändra, skapa nya).
* Systemadministratören ska ha en särskild undersida i frontend som endast syns för användarnamnet sysadmin där användare går att hantera: utläsa användare utifrån om de är (in)aktiverade, skapa nya användare med valfria behörigheter, ändra befintliga användare, radera befintliga användare.
* sysadmin har alla behörigheter som standard när kontot har skapats.
* En datorkomponent i MongoDB-databasen består av: id, namn, beskrivning, pris, antal, status (ny eller begagnad), kategorier och tillhörande bilder för datorkomponenten ifråga.
* En användare i MongoDB-databasen består av: registrerat användarnamn, samma användarnamn i endast små tecken, fullständigt namn, krypterat lösenord, en stränglista över roller (behörigheter) för användaren, en åtkomstnyckel, en uppdateringsnyckel, status om kontot är (in)aktiverat, och datum för senast inloggning för användaren ifråga.
* Utlästa datorkomponenter ska även inkludera särskilda knappar för att kunna utläsa, ändra eller radera en vald datorkomponent (beroende på behörighetsnivå; saknas behörighet för ändring så ska den knappen ej finnas och den undersidan ska ej gå att kunna navigera till).
* När en datorkomponent skapas i intranätet ska det gå att skapa det med eller utan bilder. Bilder som väljs ska kunna förhandsvisas och enskilda bilder ska kunna ändras/raderas innan själva datorkomponenten sedan skapas/läggs upp i intranätet.

## Kapitelöversikt

I kommande kapitel kommer följande att avhandlas:

[Kapitel 2](#_Teori) går igenom grundläggande teori för MVC-arkitekturen, MERN-stacken (MongoDB, ExpressJS, ReactJS, NodeJS), CSS-ramverket TailwindCSS, samt REST API.

[Kapitel 3](#_Metod) lägger fram metoden för hur det hela kommer att genomföras under de kommande kapitlen därpå genom att berätta om (utvecklings)verktyg som används samt hur det hela har publicerats.

[Kapitel 4](#_Konstruktion) berör konstruktionen av hela MERN-stacken indelat i de olika delarna: MongoDB+REST API, ExpressJS+NodeJS, ReactJS+TailwindCSS.

[Kapitel 5](#_Resultat) visar upp bilder och beskrivningar på de färdiga implementeringarna och hur de förhåller sig till kundens kravspecifikationer, det vill säga, i vilken utsträckning de har uppfyllt enligt kundens kravspecifikationer.

[Kapitel 6](#_Slutsatser) är slutsatser med reflektioner om (projekt)arbetet samt knyter ihop säcken för hela projektrapporten. Sedan följer vedertagen källförteckning och eventuella bilagor.

# Teori

Detta kapitel redogör grundläggande teori som behövs för att förstå de valda implementerade lösningarna åt AI Datorer AB där MERN-stacken inom webbutveckling kommer att användas.

## MVC-arkitekturen

MVC-arkitekturen eller *Model-view-controller*-arkitekturen är ett mjukvarudesignmönster för att ta fram användargränssnitt där logiken är fördelade i olika delar: en modell (eng. *model*) för vad för data som är tillgänglig, kontrollers (eng. *controllers*) som bestämmer vilka data som ska visas för vilka användare och vyer (eng. *views*) vilket är där dessa modellvalda data sedan visas(2).

Ursprungligen användes MVC-arkitekturen först i skrivbordsapplikationer - exempelvis Smalltalk-80(2, *History*) - men efter att internet slog igenom runtom i världen så blev det även vanligt där för webbaserade applikationer(2, *Use in web applications*). Exempel på webbaserade applikationer som använder sig av MVC-arkitekturen är bland annat AngularJS, VueJS, och Laravel(3).

Modellen i MVC-arkitekturen utgör data (och vilken (slags) data) som ska kunna bearbetas av kontrollers vilket sedan kan visas via vyerna(2, *Model*). Med hjälp av så kallade *datamodeller* kan modeller tas fram. Det förstnämnda är ett sätt att standardisera hur olika (slags) data förhåller sig till varann - vanligen baserat på entiteter från den verkliga världen(5). Exempelvis en datamodell för en person så är personen en entitet medan egenskaper/"data" om personen (exempelvis fullständigt namn, ålder, med mera) då skulle vara attribut för den entiteten.

Vyerna i MVC-arkitekturen utgör vad (oftast i samband med data) som ska visas utifrån vad kontrollers har bestämt är tillåtet efter förfrågningar har skickats från användaren inuti en vy(2, *View*) vilket vanligen sker med hjälp av *front-end-webbutveckling* där HTML+CSS skapar den statiska strukturen och det statiska utseendet medan exempelvis JavaScript används för interaktivitet och därmed dynamiskt utseende i slutändan(6).

Kontrollers i MVC-arkitekturen är det som agerar mellan modeller och vyer för att bestämma vilken vy som ska använda sig av vilken modell (vilken data att visa med andra ord) och här tar den då hjälp av förfrågningar - exempelvis HTTP-anrop - från slutanvändare(2, *Controller*).

## MERN-stacken

MERN(MongoDB, ExpressJS, ReactJS, NodeJs)-stacken är en så kallad *mjukvarulösning* vilket är uppsättningen av mjukvaror som behövs för en fullständig plattform så att inga övriga komponenter sedan behövs för att fullborda den färdiga slutgiltiga mjukvarulösningen(7).

MERN-stacken består som sagt var av den icke-relationella databasen MongoDB(M), det modulära webbapplikationsramverket ExpressJS(E) vilket i sin tur körs i JavaScript-körmiljön NodeJS(N), och frontend-ramverket/-funktionsbiblioteket ReactJS (R; 8, *Software components*).

### MongoDB

MongoDB är en databas av typen NoSQL ("Not Only SQL") och således icke-relationsbaserad som exempelvis MySQL är. Detta möjliggör att varje "tabell" i MongoDB (kallad *dokument* vilket finns i *kollektioner* vilket i sin tur är en samling av flera dokument) kan struktureras lite hur som helst så länge det är giltig BSON vilket är en lättviktig binärvariant av JSON(1, s.191-192).

Varje dokument - vilket får ha en maxstorlek på 16 MB - kan identifieras med hjälp av ett unikt \_id vilket MongoDB automatiskt skapar varje gång ett nytt dokument skapas. Varje \_id består av först 4-bytes för antalet sekunder sedan senast epok, sen 3-bytes för maskinidentifiering, sen 2-bytes för processidentifiering och sen en 3-bytes räknare som börjar med slumpvalt värde(1, s.193).

Trots att MongoDB inte är relationsbaserat så går det ändå att nyttja det genom att hänvisa till rätt \_id som en egenskap i ett dokument för att hänvisa till ett annat dokument och så vidare(1, s.196).

Med hjälp av kommandot npm install mongodb så kan MongoDB installeras som ett paket i ett givet MERN-projekt förutsatt att MongoDB-motorn redan är installerad på samma enhet först(1, s.221).

### ExpressJS

ExpressJS är en modul som implementerar http-modulen från NodeJS på ett utvecklingsvänligare vis så att det blir lättare att hantera HTTP-anrop (eng. *HTTP requests*), rutter (eng. *routes*), serversvar (eng. *responses*), kakor (eng. *cookies*) samt HTTP-statuskoder (eng. *HTTP status codes*)(1, s.343).

Rutter i ExpressJS är hur olika delar i ett URI ("Uniform Resource Locator") vid ett givet HTTP-anrop ska hanteras. Varje rutt i ExpressJS måste då först ha vad för slags URI den ska hantera och sedan vad för slags HTTP-anrop till den URI som gäller: GET, POST, UPDATE eller DELETE(1, s.345-346).

I ExpressJS är det tänkt att hantera ett givet HTTP-anrop (lagrat i mottaget req-objekt) i rätt rutt och sedan skicka tillbaka (via det lagrade res-objektet) en statuskod och ett "svar" (HTML-fil, ren text eller JSON) vilket går att göra med hjälp av exempelvis: return res.status(200).json({success: "Giltigt HTTP-anrop!";}) (1, s.353,355).

I vissa fall så kan det vara önskvärt att hantera req-objektet innan det kanske hanteras för att skicka tillbaka ett särskilt svar och för dessa används då så kallade middlewares vars namn antyder om "mellanmjukvara" vilket betyder kod som körs innan det hanteras(1, s.367). Se kodexemplet nedan:

// POST /api/register = Register new user

router.post("/register",validateFormInput.registerNewUser,

  mongoDB("maka2207", "users"),registerUser.registerUser);

I kodexemplet ovan så hanteras ett POST-anrop för URL som slutar på "/register" men innan det faktiskt hanteras av registerUser.registerUser så körs två middlewares innan: validateFormInput.registerNewUser och sedan mongoDB("maka2207","users").

Skulle någon av dessa två middlewares skicka tillbaka ett svar istället för att bara "passa vidare" req-objektet så skulle registerUser.registerUser aldrig köras för att HTTP-anropet skulle då redan ha hanterats i körflödet.

En middleware kan också placeras precis innan nästa kodrad med hantering av en rutt för diverse HTTP-anrop. Kodraden nedan exemplifierar:

router.use(validateJWT);

Kodraden ovan använder en middleware lagrad i validateJWT och om inte denna skickar vidare req-objektet via next()-funktionen(1, s.322) så kan den istället skicka eget svar och då upphör hanteringen av det inkomna HTTP-anropet som annars kanske hade kommit till en ytterligare rutt.

### ReactJS

React eller React.js eller ReactJS är ett frontend JavaScript-bibliotek vilket är till för att bygga komponentbaserade användargränssnitt. Det utvecklades först av Jordan Walke med det första versionssläppet den 29:e maj 2013 och numera sköts det av bolaget Meta (tidigare Facebook). Den senaste versionen i skrivande stund är 18.2.0 släppt den 14:e juni 2022. Det kan köras mobilt, som serverrendering och/eller *Single-page Application* eller "SPA"(10).

ReactJS använder sig av något som heter JavaScript XML (tidigare kallat JavaScript Syntax eXtension) eller bara JSX förkortat vilket möjliggör skapandet av DOM-träd med XML-liknande syntax vilket gör det semantiskt enklare att läsa och/eller vad som är ren HTML och vad som är dynamiskt innehåll från JavaScript(12).

ReactJS använder sig av så kallade funktionella komponenter vilket i sin tur består av funktionsnamn vilket är det som exporteras ut och sedan importeras in och används i andra komponenter och/eller i huvudkomponenten som exempelvis App.js(10, *Function components*).

En ReactJS-komponent innehåller körlogiken för de data som ska kunna uppdateras dynamiskt sedan inuti dess returnerade JSX-kod(10, *Function components*).

Med hjälp av så kallade *hooks* så kan ReactJS köra särskild logik innan färdigrenderade JSX-koden returneras av komponenten. Exempel på *hooks* är useState, useContext, och useEffect bland annat(10, *React Hooks*).

När useState används så kan data för en given komponent - såväl som dess eventuella underkomponenter - bevakas av ReactJS och så kan JSX-koden uppdateras returneras om på nytt vilket kallas för *rendering* inuti ReactJS. Den som heter useContext påminner om useState men är istället tänkt att fungera globalt. Till sist så fungerar useEffect genom att köra sidoeffektkod efter att *rendering* har ägt rum(10, *React Hooks*).

Då ReactJS körs som en så kallad *SPA* i webbklienten så behövs en särskild lösning för att kunna navigera till undersidor på ett liknande vis som att navigera till olika webbsidor i vanliga webbplatser så därför finns React Router där *HashRouter* är en lösning där det finns /#/<undersida> i webbadressfältet som ett sätt att navigera till undersidor i ReactJS-applikationen när det gäller rör sig om att ladda fram andra komponenter utifrån konfigurering(15).

### NodeJS

NodeJS eller Node.js är en JavaScript-körmiljö för att kunna köra JavaScript (via V8 JavaScript-motorn) utanför webbläsaren som exempelvis JavaScript som körs på serversidan. NodeJS är skrivet i programmeringsspråken JavaScript, C++ och Python. Node.js följer händelsedriven arkitektur med förmåga att hantera asynkrona I/O-hanteringar(4).

NodeJS kom först den 27:e maj 2009 varav den senaste stabila versionen i skrivande stund är version 21.4.0 släppt den 5:e december 2023(4, *History*). Node.js innehåller moduler vilket möjliggör skapandet och körningen av http(s)-servrar, filhantering på serversidan, sökvägshantering, kryptografi, dataströmshanteringar, med mera(4, *Overview*).

Det är bland annat http-modulen(9) - från NodeJS - som ExpressJS har använt sig av för att förenkla webbutvecklingen av JavaScript-baserade servrar.

## Tailwind CSS

Tailwind CSS är ett ramverk för CSS ("Cascading Style Sheets") och skiljer sig från andra CSS-ramverk genom att det ej kommer med fördefinierade CSS-klasser för element utan istället använder sig av korta prefix för typiska CSS-regler(11).

Exempelvis kanske ett CSS-ramverk skulle använda sig av en fördefinierad klass msg-warn vilket applicerar gul bakgrund och fettext medan Tailwind CSS istället använder sig av två korta CSS-regler bg-yellow-300 och font-bold för att uppnå samma effekt(11).

Dessa CSS-regler skrivs - med ett mellanslag mellan varje två CSS-regler - i ett och samma klassnamn inuti ett och samma HTML-element som det ska appliceras på(11). Exempelvis: <div class="bg-yellow-300 font-bold">Varning!</div>.

Trots att Tailwind CSS innehåller korta prefix för alla tillgängliga CSS-regler så kommer endast de faktiskt använda i HTML-elementens klassnamn att inkluderas i den slutgiltigt genererade CSS-filen när det blir dags för (att leverera ut till) produktion(11).

## REST API

REST API ("Representational state transfer" och "Application Programming Interface"; 13,14) så är första ordet en typ av mjukvaruarkitektur för att vägleda utvecklingen på internet genom att följa särskilda principer som REST förespråkar(13), medan det andra ordet beskriver ett kommunikationsgränssnitt mellan två program utan att programmen ska kunna känna till varandras interna detaljer för övrigt(14).

Tack vare ett välbearbetat API så kan kommunikationsgränssnittet mellan två program - exempelvis en klient såsom ReactJS och en server såsom ExpressJS - begränsas så att endast särskilda förfrågningar (exempelvis endast CRUD:a särskilda data) kan göras under vissa särskilda omständigheter (exempelvis med rätt och påvisade behörigheter) och allt detta i sin tur ökar säkerheten hos den tillhandahållande parten i kommunikationen(14).

I REST så förespråkas bland annat något som heter "*Client-server architecture*" vilket innebär att vad som sker på klientsidan (exempelvis i en webbläsare) ska vara separat för vad som möjligen sker på serversidan (exemeplvis i ExpressJS i detta fall)(13, *Architectural constraints*).

En annan sak som också förespråkas i REST-arkitekturen är "*Statelessness*" vilket innebär att användaren ska kunna skicka en förfrågan till en server utan att behöva berätta om vilket tillstånd (eng. *state*) användaren är i eller gör den förfrågan i. Servern ska kunna skicka korrekt svar så länge förfrågan är korrekt utformad och med tillräckliga nödvändiga data i samma veva(13, *Architectural constraints*).

Exempel på det sistnämnda för att illustrera det hela är att en användare som vill visa en given sida ska inte tvinga servern att veta vilken sida användaren är på. Användaren ska ej skicka förfrågan om "Nästa sida" utan snarare "Sida X" där X avser vilken sida som önskas erhållas från servern.

Detta avslutar då teoridelen. Nästa del är metoddelen där integrerad utvecklingsmiljö och publiceringsform redogörs för.

# Metod

Följande kapitel redogör för vilka verktyg som har använts i webbprojektet.

## Integrerad utvecklingsmiljö

I webbprojektet har *Microsoft Visual Studio Code* (VSCode) i *Windows 10 Pro Eng* använts som integrerad utvecklingsmiljö för att skriva all HTML-, CSS- & JS-kod. Hämtat här: <https://code.visualstudio.com/>.

I det inlämnade GitHub-repot för hela webbprojektet (både klient & server i ett och samma GitHub-repo) - tillgängligt här <https://github.com/WebbkodsLarlingen/dt162g-projekt-maka2207> - så har följande npm-paket och deras rekommenderade versioner använts:

* "bcrypt": "^5.1.1"
* "cookie-parser": "^1.4.6"
* "cors": "^2.8.5"
* "dotenv": "^16.3.1"
* "express": "^4.18.2"
* "jsonwebtoken": "^9.0.2"
* "mongodb": "^6.3.0"
* "nodemon": "^3.0.1"
* "multer": "^1.4.5-lts.1",
* "axios": "^1.6.5",
* "react": "^18.2.0"
* "react-dom": "^18.2.0"
* "react-router-dom": "^6.21.1",
* "react-scripts": "5.0.1"
* "tailwindcss": "^3.3.6"

De första nio paketen gäller för serverdelen (NodeJS, ExpressJS, MongoDB) medan de sex resterande gäller för klientdelen (ReactJS, TailwindCSS).

Beträffande databasen MongoDB så har användargränssnittet [MongoDB Compass Version 1.41.0](https://www.mongodb.com/try/download/compass) använts för att kunna lättare felsöka databasens användning, medan [MongoDB Community Server version 7.0](https://www.mongodb.com/try/download/community) har använts för själva lokala MongoDB-databasservern - med anslutningsadress: mongodb://localhost:27017.

## Publicering

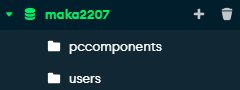
Publicering förekommer endast i den utsträckningen att [GitHub-repot](https://github.com/WebbkodsLarlingen/dt162g-projekt-maka2207) laddas ned och körs lokalt förutsatt att rätt MongoDB Community Server version 7.0 finns installerat och att det då kan anslutas till anslutningsadressen: mongodb://localhost:27017. Läs och följ sedan instruktionerna inuti README.md i GitHub-repot för att komma igång lokalt.

# Konstruktion

Detta kapitel avhandlar det faktiska praktiska arbetet i hur saker och ting har lösts samt varför de har lösts på de sätt de har lösts på.

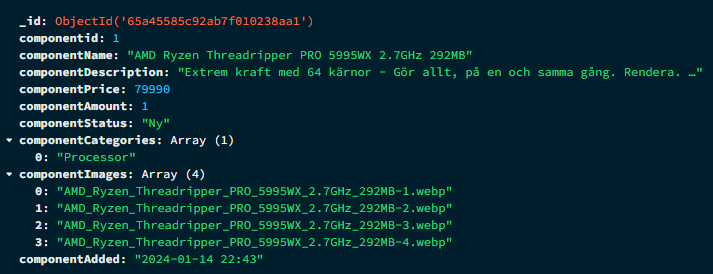
## MongoDB, REST API, NodeJS & ExpressJS

Lösningen för MongoDB och REST API:t börjar med MongoDB-databasen som består av två kollektioner: pccomponents och users (se Figur 1 nedan). Dessa går att skapa testdata automatiskt genom att köra npm run installmongodb i terminalen efter nedkloning av GitHub-repot.



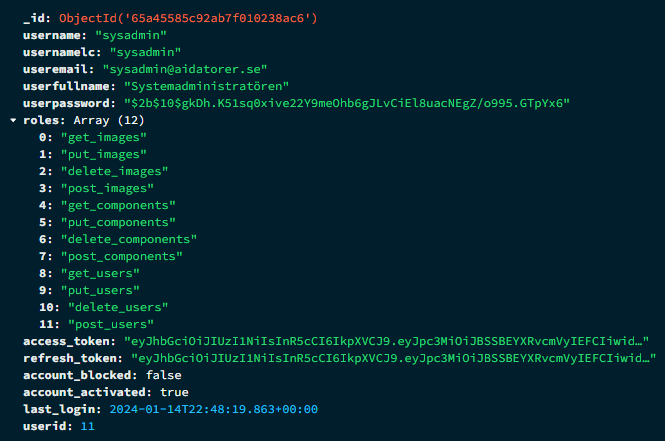
Figur : Kollektioner i MongoDB-databas

I Figur 2 nedan visar exempel på en datorkomponent som lagras i form av ett MongoDB-dokument. Den har först vedertagna \_id: för att MongoDB själv ska kunna identifiera dokumentet. Sedan har det ett componentid: som används för att kunna CRUD:a given datorkomponent. Varje datorkomponent har sedan ett namn, en beskrivning, ett pris, ett antal, status på om det är "Nytt" eller "Begagnat", en array med kategorier, och en array med eventuella bilder, och till sist vilket datum när det lades till eller ändrades senast.



Figur : Exempel på MongoDB-dokument ur kollektionen pccomponents

I Figur 3 på nästa sida visas exempel på en användare där registrerat användarnamn, samma användarnamn i bara små tecken, e-post, fullständigt namn, och krypterat lösenord lagras. Sedan tillkommer en array med roller vilket utgör behörigheter för användaren. Med hjälp av userid: kan enskild användare CRUD:as. Fälten access\_token och refresh\_token används för att kunna göra anrop mot REST API:t respektive förnya access\_token när det utgått (90 sekunder livslängd). En refresh\_token har en livslängd på 24 timmar. Fältet account\_activated gör så att om inloggning lyckas men kontot är blockerat så kommer användaren ej in. Fältet account\_blocked används ej. Fältet last\_login ändras varje gång användaren ifråga lyckas logga in.



Figur : Exempel på MongoDB-dokument ur kollektionen users

Serverns utgångspunkt - där webbservern startar NodeJS+ExpressJS - ligger i /server/server.js vilket initierar stöd för JSON, URLEncoded (formulär), CORS (konfiguration lagrad i /config/corsOPTIONS.js), webbkakor, och sedan använder den en statisk rutt för bilder vilket webbklienten sedan kan hämta bilder från (bilder för testdata såväl som egenuppladdade bilder via webbklienten lagras i /server/images/{componentid}):

app.use("/images")

Ett ogiltigt REST API-anrop (i alla huvudsakliga rutters och underrutters) hanteras av följande vilket då skickar tillbaka antingen en bild om det är i webbklienten eller JSON-svar om att det är ogiltigt REST API-anrop:

router.all("\*");

Alla REST API-anrop - de som börjar med /api/ - går mot /server/routes/root.js rutten där det i sin tur finns fyra publika rutters varav vissa även har inmatningsvalidering som *middleware* (se /server/middlewares/validateUsersInputs.js):

router.post("/login")

router.post("/register")

router.post("/logout")

router.get("/refreshatoken")

En annan relevant *middleware* för i princip alla rutters som behöver göra databasanrop (vilket är i princip alla) är då den som heter mongoDB (se /server/middlewares/db.js) som tar sträng för vad databasen heter och vilken kollektion som ska anslutas. I vissa fall kan båda kollektionerna behöva användas och då kan båda kollektionerna skickas med i anropet till den *middlewaren*.

Efter dessa fyra publika rutters så körs en ytterligare annan *middleware* (se /server/middlewares/validateJWT.js) som kontrollerar att giltig access\_token har tillhandahållits från slutanvändaren först. Annars måste klienten anropa rutten /refreshatoken för att förnya sin access\_token.

När access\_token är giltig så finns två ytterligare underrutters som hanterar CRUD:s gällande pccomponents (se /server/routes/api/pccomponentsRouter.js) respektive users (se /server/routes/api/usersRouter.js):

router.use("/pccomponents")

router.use("/users")

Nedanför visas alla rutter tillgängliga i /server/routes/api/pccomponentsRouter.js och dess olika tillgängliga CRUDs (POST, GET, PUT & DELETE) och huruvida bilder kan laddas upp (se upload.array eller upload.single):

router.route("/")

  .get()

  .post(upload.array("componentimages"));

router.route("/:id")

  .get()

  .put()

  .delete();

router.route("/:id/images/:arrayindex")

  .put(upload.single("componentimages"))

  .delete();

router.route("/:id/images")

  .post(upload.single("componentimages"));

Som det framgår ovan så går det att skicka med flera bilder vid POST-anrop mot /api/pccomponents/ och en bild vid PUT-anrop mot /api/{componentid}/images/{componentImagesArrayIndex} eller POST-anrop mot /api/{componentid}/images/. Det är npm-paketet multer (lagrad i variabeln upload) som används för att först skicka upp bilder som sedan raderas om anropet ej lyckades hela vägen (t.ex. nekades pga. fel filtyp och/eller databasfel).

Gemensamt för alla dessa rutters ovan är att först skickas de igenom till rätt funktion lagrad i /server/middlewares/validateUsersInputs.js och valideras allt OK så skickas den vidare till ansvarig funktion lagrad i /server/controllers/pccomponentsController.js vilket är den som faktiskt gör databaskontroller och sedan eventuell CRUD i MongoDB-databasens kollektion pccomponents.

Exakt samma lösning har implementerats för eventuell CRUD i MongoDB-databasens kollektion users efter databasvalidering och inmatningsvalidering i /server/middlewares/validateUsersInputs.js där då all REST API CRUD finns i /server/controllers/usersController.js:

router.route("/")

  .get()

  .post();

router.route("/:id")

  .get()

  .put()

  .delete();

Ovanför framgår det att det går att hämta alla användare, posta en ny användare, hämta/ändra/radera en enskild användare utifrån korrekt tillhandahållen userid.

De kontrollers som hanterar förnyelse av access\_token, registrering, inloggning och utloggning är följande under /server/controllers/: refreshATokenController.js, registerUserController.js, respektive loginAndLogoutController.js. Filen för inloggning och utloggning är alltså en sammanslagen fil där både inloggning och utloggning sker genom två separata funktioner inuti filen.

## ReactJS och TailwindCSS

Lösningarna för ReactJS och TailwindCSS ligger i /client/-katalogen vilket i sin tur ligger i samma nivå som /server/-katalogen. I Figur 4 längst ned på nästa sida visas komponentträdet där först HashRouter omfamnar alla underkomponenter i /client/src/App.js-filen vilket i sin tur först monteras och renderas i /client/src/index.js-filen.

Här nedan framgår komponentträdet inuti App.js-filen och det som ska betraktas här nedan är <Routes> (syns först i början på nästa sida) vilket utgör alla ReactJS-rutters i webbklienten vilket då kan navigeras till i webbläsaradressfältet i stil localhost:3000/#/{Route} där det också framgår vilka objekt som skickas av AuthContenxt.Provider vilket gör variablerna globalt tillgängliga för alla komponenter istället för att de måste skickas ned via varje komponentnivå.

<AuthContext.Provider value={{ aToken, setAToken, isAdmin,isLoggedIn,setIsAdmin,setIsLoggedIn,accesses,setAccesses,}}>

<Header isLoggedIn={isLoggedIn} isAdmin={isAdmin} setAccessToken={setAccessToken} setLoginSuccess={setLoginSuccess}

 setAdmin={setAdmin} />

<Routes>

<Route path="/login" element={<Login

setAccessToken={setAccessToken} setAccess={setAccess}

setLoginSuccess={setLoginSuccess} setAdmin={setAdmin} />}></Route>

<Route path="/register" element={<Register />}></Route>

<Route path="/" element={<Start isLoggedIn={isLoggedIn} />}></Route>

<Route path="/products" element={<Products isLoggedIn={isLoggedIn} />}></Route>

<Route path="/products/:id" element={<Product isLoggedIn={isLoggedIn} />}></Route>

<Route path="/products/:id/edit" element={<EditProduct isLoggedIn={isLoggedIn} />}></Route>

<Route path="/products/add" element={<AddProduct isLoggedIn={isLoggedIn} />}></Route>

<Route path="/admin"element={<Admin isLoggedIn={isLoggedIn} />}></Route>

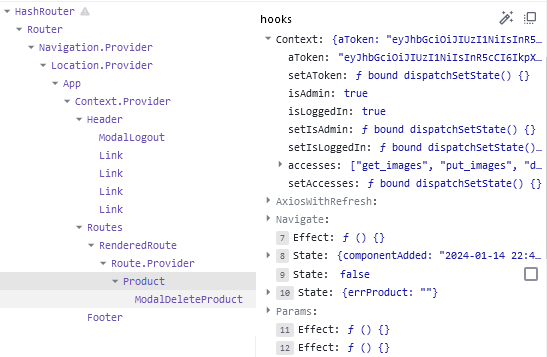
<Route path="/admin/adduser" element={<AddUser isLoggedIn={isLoggedIn} />}></Route>

<Route path="/admin/edituser/:id" element={<EditUser isLoggedIn={isLoggedIn} />}></Route>

<Route path="/\*" element={<NotFound />}></Route>

</Routes>

<Footer /></AuthContext.Provider>



Figur : Komponentträdet med HashRouter i ReactJS-webbklienten

Lägg märke till det ofta förekommande isLoggedIn-objektet som skickas in i alla rutters vilket betyder att användaren måste vara inloggad för att få besöka undersidorna i webbklienten annars skickas de till /login-rutten.

I Figur 4 på föregående sida framgår då komponentträdet mer visuellt till vänster och till höger framgår något som heter Context: vilket är de globala tillstånd (eng. "*states*") som alla komponenter kan komma åt:

* aToken är access\_token i webbklienten lagrad i en variabel (ingen localStorage eller dylikt).
* setAToken är funktionen som ändrar värdet för aToken.
* isAdmin är boolean om inloggad användare är sysadmin eller ej.
* isLoggedIn är boolean om användaren faktiskt är inloggad eller ej.
* setIsAdmin är funktionen som ändrar värdet för isAdmin.
* setIsLoggedIn är funktionen som är ändrar värdet för isLoggedIn.
* accesses är en array som innehåller alla behörigheter som den inloggade användaren har, dessa kontrolleras också dock på REST API-sidan så även om användaren ändrar manuellt så kommer det ej att fungera på serversidan.
* setAccesses är funktionen som ändrar värdet för accesses.

Under /client/src/components/-katalogen ligger alla komponenter som inte är undersidor till webbklienten medan /client/src/router/-katalogen har alla faktiska undersidor till webbklienten.

Exempelvis Start.js i /client/src/router/ utgör standardstrukturen för de flesta undersidorna i ReactJS-webbklienten:

1. Först importeras nödvändiga ReactJS-komponenter och funktionalitet.
2. Funktionskomponenten tar emot isLoggedIn-objektet för att sedan under sin första useEffect kontrollera om användaren är inloggad vilket annars navigerar användaren till rutten /login.
3. Åtkomst till de globala tillstånden så används useContext(AuthContext) där aToken och accesses bland annat behöver kommas åt.
4. Den andra useEffect hämtar nödvändig data med hjälp av det importerade axiosWithRefresh-objektet (se /client/middleware/axiosWithRefresh.js) vilket i sin tur är ett axios-objekt från npm-paketet axios. Det som gör detta axios-objekt speciellt är att det kontrollerar först om det finns någon access\_token och sedan extraherar den ut exp-objektet vilket är datumet för när den löper ut (90 sekunder efter utfärdande) och är dagens datum då större än när det löper ut så försöker det erhålla en ny access\_token genom att anropa /refreshatoken.
5. Varje komponent som ska hantera data på något vis lagras dessa i en useState(null) som sedan lagras med erhållen data.
6. Varje komponent som ska hantera data ska också ha meddelanden och har en annan useState({}) som lagrar meddelanden för framgång och/eller felmeddelanden.
7. Till sist så kontrolleras behörighet lagrad i accesses med hjälp av if-satser för att avgöra vad som ska visas eller inte i komponenten ifråga (detta förutsätter att användaren ens är inloggad innan detta ens inträffar):

if(!accesses.includes("get\_components")) return (<></>

Kodraden ovanför kontrollerar om behörigheten för att "visa komponenter" *inte* finns inuti arrayen accesses och stämmer det så kommer ingenting att visas från denna komponent i ReactJS-webbklienten.

Detta avslutar då redogörandet för lösningarna och kapitlet om konstruktion. Sammanfattningsvis så är det som sagt var useState() som lagrar data som kan hanteras i webbklienten som sedan skickas till REST API:t och får den OK så kommer den även att ändra data i webbklienten för att reflektera att förändringen gick igenom på serversidan.

Med hjälp av useEffect() så kan sidoeffekter köras som exempelvis att hämta data från servern. Trots att behörigheter lagras i en array inuti ReactJS-webbklienten så kontrolleras dessa alltid på serversidan innan någon CRUD av data i MongoDB-databasen ens görs vilket då förebygger riskerna med en erhållen och modifierad JWT som skulle kunna ha gjort annars otillåtna CRUD-anrop mot MongoDB-databasen.

Nu återstår resultat vilket visar upp visuella resultat av hur det hela ser ut när lösningarna har förverkligats i praktiken genom att lokal webbserver, lokal webbklient, samt lokal databas körts och skärmdumpats.

# Resultat

I detta näst sista kapitel innan slutsatser redogörs AI Datorer AB:s färdiga webbaserade intranät samt dess funktionaliteter. Först redogörs hur det ser ut att registrera sig, logga in och även logga ut. Sedan redovisas hur det går till att hantera datorkomponenter i form av att utläsa, skapa nya, ändra och/eller radera befintliga datorkomponenter. Till sist visas det upp hur det går till för den utnämnda Systemadministratören att hantera användarna genom utläsning, skapande, ändrande och radering av användare.

## Registrering, inloggning och utloggning

a.

## Hantera datorkomponenter

a.

## Hantera användare

a.

# Slutsatser

a.

Källförteckning

1. B. Dayley. *Node.js, MongoDB and Angular Web Development (Second Edition)*. USA: Pearson Education Inc. 2018
2. Wikipedia, "Model-view-controller", <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller> Ändrad senast: 2023-12-22 Hämtad: 2023-12-31
3. Wikipedia, "Comparison of server-side web frameworks", [https://en.wikipedia.org/wiki/Comparison\_of\_server-side\_web\_frameworks#Comparison\_of\_features](https://en.wikipedia.org/wiki/Comparison_of_server-side_web_frameworks%23Comparison_of_features) Ändrad senast: 2023-12-29 Hämtad: 2023-12-31
4. Wikipedia, "Node.js", <https://en.wikipedia.org/wiki/Node.js> Ändrad senast: 2023-12-30 Hämtad: 2023-12-31
5. Wikipedia, "Data model", <https://en.wikipedia.org/wiki/Data_model> Ändrad senast: 2023-11-08 Hämtad: 2024-01-02
6. Wikipedia, "Front-end web development", <https://en.wikipedia.org/wiki/Front-end_web_development> Ändrad senast: 2023-11-11 Hämtad: 2024-01-02
7. Wikipedia, "Solution stack", <https://en.wikipedia.org/wiki/Solution_stack> Ändrad senast: 2023-11-09 Hämtad: 2024-01-02
8. Wikitia, "MERN (solution stack)", <https://wikitia.com/wiki/MERN_(solution_stack)> Ändrad senast: 2020-05-14 Hämtad: 2024-01-02
9. Node.js, "HTTP | Node.js v21.5.0 Documentation", <https://nodejs.org/docs/latest/api/http.html> Hämtad: 2024-01-02
10. Wikipedia, "React (software)", <https://en.wikipedia.org/wiki/React_(software)> Ändrad senast: 2023-12-18 Hämtad: 2024-01-02
11. Wikipedia, "Tailwind CSS", <https://en.wikipedia.org/wiki/Tailwind_CSS> Ändrad senast: 2023-12-23 Hämtad: 2024-01-02
12. Wikipedia, "JSX (JavaScript)", <https://en.wikipedia.org/wiki/JSX_(JavaScript)> Ändrad senast: 2024-01-02 Hämtad: 2024-01-02
13. Wikipedia, "REST", <https://en.wikipedia.org/wiki/REST> Ändrad senast: 2023-12-28 Hämtad: 2024-01-02
14. Wikipedia, "API", <https://en.wikipedia.org/wiki/API> Ändrad senast: 2023-12-19 Hämtad: 2024-01-02
15. React Router, "HashRouter v6.21.2", <https://reactrouter.com/en/main/router-components/hash-router> Hämtad: 2024-01-15