UMEÅ UNIVERSITY **April 4, 2019**

Applikationsutveckling för internet vt19

Laboration1

API med Alternativt Gränssnitt

Albin Frick

Handledare
Per Kvarnbrink, Stefan Berglund

Contents

1 Inledning

Denna laboration har gått ut på att bekanta sig med node.js och react.js. Jag valde att bygga mitt API med hjälp av node.js och det konsumerande gränssnittet med react.js. Jag valde att använda både node och react för att jag ansåg det var ett effektivt sätt att lära mig två olika ramverk.

Jag har valt att göra uppgiften på plus-nivå.

2 Problembeskrivning

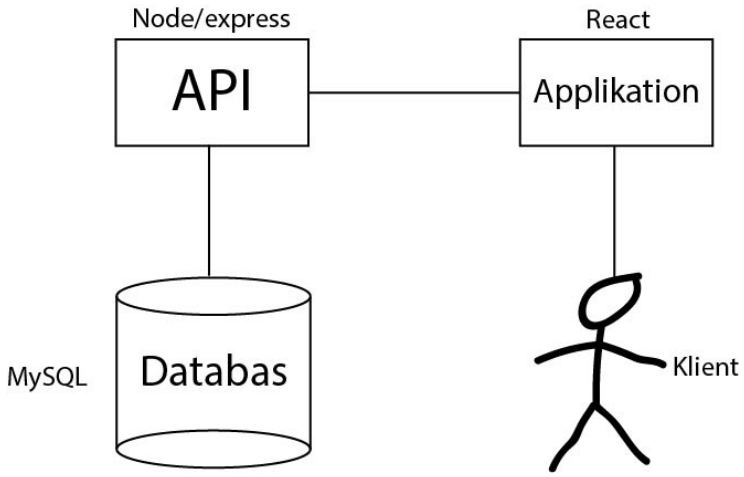
För att få detta att fungera måste det finnas tre huvudkomponenter.

- Databas
- Webbserver - API
- Webbserver - Konsumerande gränsnitt

Laborationen gick alltså ut på att skapa dessa tre och få dem att samarbeta på ett tillfredställande sätt.

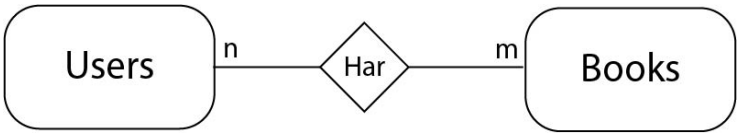
3 Systembeskrivning

Databasen är byggd i MySQL och består av tre tabeller, Books, Users, BookUser-Relation. Bara Books och Users blev använda under laborationen på grund av tids brist. Vid vidare arbetning av API:t skulle de fortfarande gå att använda.



Figur 1: Relationen mellan de olika komponenterna

I figur 1 finns ett ER-schema där man kan se hur databasstrukturen.



Figur 2: ER-schema över databasen

API:t är byggt med hjälp av node.js och express. API:t hämtar, visar och ändrar i databasen. Varje operation sker i en egen ”route”. Ett exempel på detta går att se i figur 2.

```
//Delete one given user
router.delete("/user_delete", checkAuth, (req, res) =>{
    const userID = req.body.user.ID
    const username = req.body.user.username
    const queryString = "DELETE FROM Users WHERE ID = ?"
    getConnection().query(queryString, userID, (err, results, fields) =>{
        if(err){
            console.log("Failed to delete user: " + err);
            res.sendStatus(500);
            return;
            }
            console.log(username + " was deleted with id " + userID)
            res.end()
    })
})
```

Figur 3: Kod som besrkiver routen /users_delete

En ”route” är alltså url:en till en viss plats i API:t, som /users, /books mm. Efter vilket route som är beskriven i figur 2 ser man checkAuth. Denna funktion sköter hanteringen om användaren är inloggad eller inte men mer om det senare.
Genom att använda pluginen ”mysql” i node så har en koppling gjorts med databasen i funktionen getConnection(). När en koppling är gjord kan man då skicka in en förfrågan till databasen. I figur 3 är förfrågan

”DELET FROM Users WHERE ID = ?”

Frågetecknet agerar som en placeholder för parametern för id:t som skickas med i queryn. Om det skulle bli något fel vid borttagningen ur databasen kommer en felstatus med 500 att skickas tillbaka som response. Blir inget fel kommmer användarnamnet och id att skrivas ut i konsolen.

Det konsumerade gränssnittet är uppbyggt i react och består av åtta sidor. Alla sidor har samma meny förutom ”Login” och ”Logout” då detta behövs för att uppdatera inloggningen. Från de andra sidorna kan man ta sig till andra sidorna.

Genom att klicka in på användare t.ex. får man en lista över de användare som finns därifrån kan man välja att radera eller redigera en användare dock fungerar detta enbart om na är inloggad.

För att skicka eller hämta data mellan API:t och klienten används react pluginen ”axios” detta är en förlängning från javascriptets ”ajax”. Det används för att kunna skicka get-, post-, put- och delete-förfrågningar.

Dessa förfrågningar är alltså:

- get - Hämta information
- post - Lägga till information
- put - Redigera information
- delete - ta bort information

Genom axios skickas ett objekt av det som ska läggas till, redigeras eller tas bort. Där det krävs inloggning skickas också ett ”token” med.

Validering är gjord på både klient och server. På klientsidan stoppas användaren om den försöker skapa en användare med för kort användarnamn eller glömmer att skriva in ett lösenord t.ex.

P.g.a. tidsbrist blev det ingen komplett validering på server-nivån. Det finns validering när en användare ska skapas. Då kollas det om användarnamnet redan finns i databasen och ser till så att de inte är null. Denna information skickas också vidare upp till klienten så användaren vet vad den ska göra.

När något går snett i API:t kommer det att skickas en status kod tillbaka beroende på vad som har hänt. Det finns väldigt många statuskoder där den mest kända kanske är 404 - not found. Genom en enkel googling kan man få upp listor om vad alla betyder.
De statuskoder som har använts mest i denna labb har varit:

- 404 - Not Found
- 401 - unauthoraized
- 500 - Internal Server Error
- 200 - OK

Första siffran i statusen grupperar den.

- 1 - Informerande
- 2 - Lyckad förfrågan
- 3 - Vidarekoppling

- 4 - Klientfel
- 5 - Serverfel

3.1 Inloggning

En inloggning kan vara bra att ha för ett API så inte vem som helst kan komma in och ändra och ta bort grejor. I tidigare kurser har vi använt oss av sessionsvariabler eller något linkande löper ut efter en viss tid. Detta går inte i ett API då den är så kallad ”stateless” API:t kan alltså inte veta om någon är inloggad.

För att lösa detta kan man använda sig av jsonwebtokens, tokens. Dessa token kan genereras till varje användare och kontrolleras API:t.

När man skapar en ny användare kollar API:t först att det inte finns en användare med samma användarnamn (kanske skulle varit bättre med email, men använde användarnman i mån av tid) sedan hashas lösenordet med saltning och läggs till i databasen. Allt detta är försståss om ingenting går fel.

När den nya användaren sedan loggar in kommer användar namnet och lösenordet att jämnföras igen. ’bcrypt’ har använts för att hasha och jämföra lösenorden. Om allting är OK kommer ett token att signeras. Det signeras med användarens användarnamn, id och en privat nyckel som är en miljö-varibel (så man inte kan se den i koden). När den signeras sätts också en tid, en timme i detta fall. Till slut returneras responsen med status 200 (OK), ett meddelande att allting gick bra och tokenet.

För de routes på API:t som enbart användare har åtgång till kontrolleras då detta token varje gång genom en så kallad middelware. Denna kontrollerar det givna token från användaren och den privata nyckeln stämmer allt kommer man vidare till routen annars kommer en status av 401 att skickas tillbaka.

När användaren loggar in på klientsidan kommer klienten att ta emot det nyssa skapade tokenet och lägga till det i en global variabel. Denna kommer att ligga där till användaren loggar ut. Hinner tokenet löpa ut kommer det att skrivas över då användaren loggar in igen.

4 Implementation

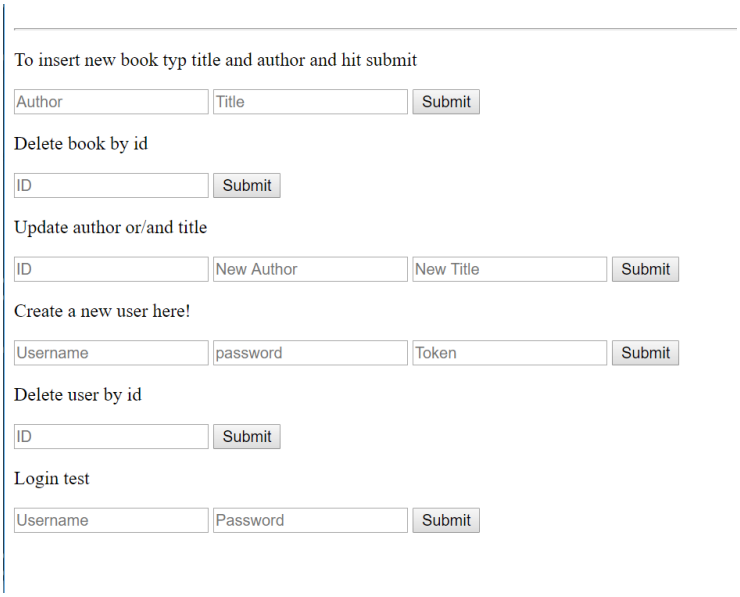
Som nämt så har node.js, react.js och MySQL använts för att implementera denna lösning. Jag ska skrivit både backend och frontend i visual studio code. Genom den integrerade terminalen kunde jag har koll på vad som hände på API och gränssnitts servern. För node användes ”nodemon”. Detta är en plugin som startar om servern varje gång någonting ändras. Detta var till stor hjälp då man

gör många ändringar under utvecklingen.

För att skapa react projectet användes:
npx create-react-app
Detta generarar nödvändiga filer och mappar för att komma igång med utvecklingen.
Byggandet av alla komponenter gick till i följande ordning:

1. Databas
2. API
3. Konsumerande gränssnitt

API:t testades genom att skapa ett html sida på samma host. Sidan kan ses på figur 4 och innehåller bara inmatningsfält och knappar.

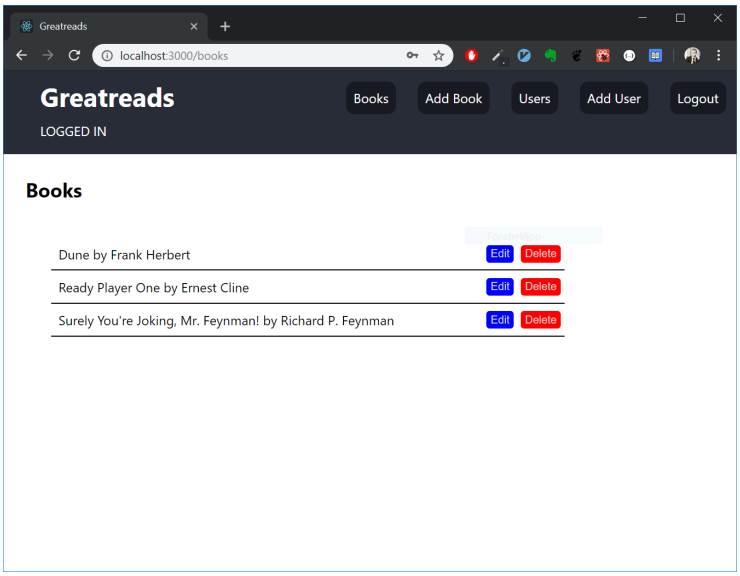


Figur 4: Testsida för API

Det hade nog varit bättre att använda sig av något som postman men detta fungerade också.

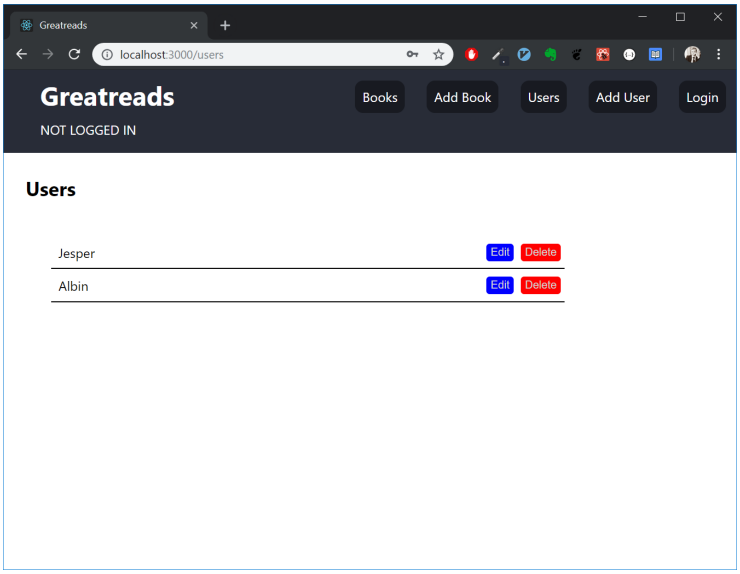
5 Gränssnitt

Gränssnittet har varit väldigt simpelt men med en del extra finnesser. Som man kan se från figur 5 så är det inte väldigt vackert men det kan var användbart.



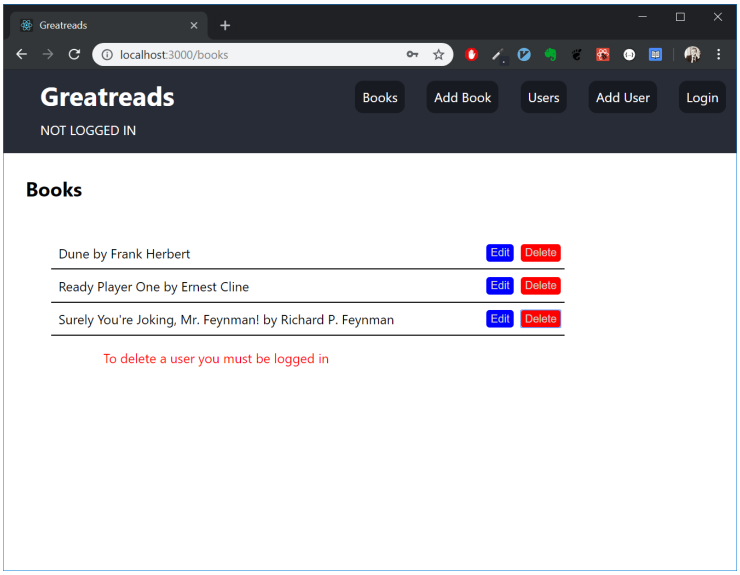
Figur 5: Lista över böcker

Från figur 5 kan man se under loggan att man är inloggad. Längst till höger finns det också en logga ut knapp. Klickar man på den får man logga ut. Kommer man till listan över användare som figur 6



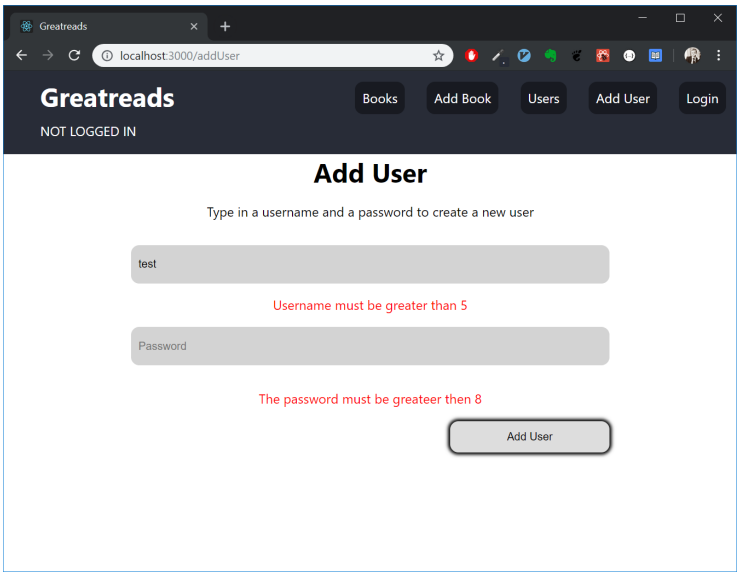
Figur 6: Lista över användare

När man loggat ut kommer knappen för att logga ut att försvinna och ersättas av en logga in knapp. Om man försöker att ta bort en bok utan att loggat in kommer det att se ut som på figur 7



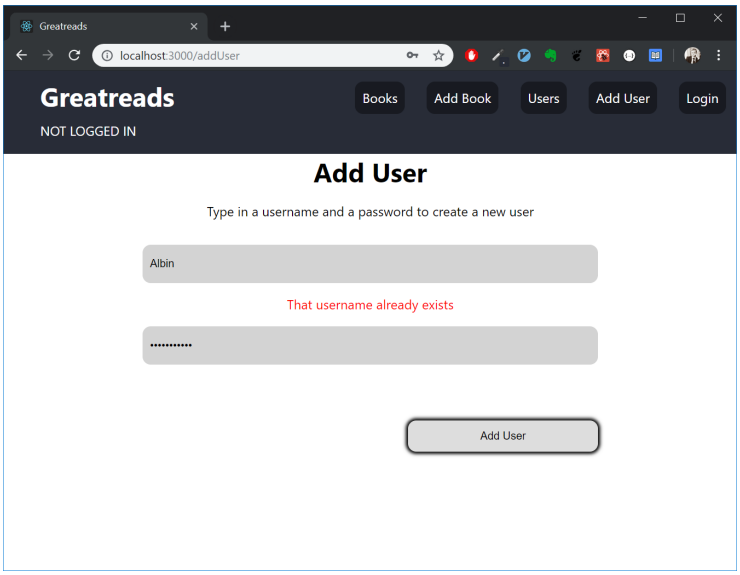
Figur 7: Felmeddelande då man inte loggat in

Som validering när man skapar en ny användare så måste användarnamnet var fem karaktärer långt och lösenordet åtta.



Figur 8: Felmeddelande vid förkorta inputs

Försöker man med en användare som redan finns ser det ut som figur 9.



Figur 9: Felmeddelande vid samma användarnamn

De två sista bilderna är exempel på validering på olika nivåer. När användaren skriver för kort användarnamn kollas det i klienten. Detta skulle kunna kollas i

API:t men det är ganska onödigt då klienten vet hur långt det får vara och kan göra det snabbare och behagligare för användaren.

Klienten kan inte se in i databasen så därför skickas informationen med det inskrivna användarnamnet och lösenordet till API:t här kontrolleras det och ser att användarnamnet finns. API:t släpper då båda de inskrivna användarnamnet och lösenordet och skickar tillbaka en response med statusen 401 unauthorized och meddelandet: "That username already exist". Detta är alltså validering på server-nivå för att det inte kan ske på klient-klient nivå.

Går det att lösa valideringen på klient-nivå är det bättre då det blir mindre jobb för datorn och bättre upplevelse för användaren.

6 Lösningens begräsningar

Lösningen har en del begrännsningar. Just nu kan inte användare välja böcker till sig själva utan bara se alla och lägga till. För att göra detta måste relationstabellen för mellan users och books att implementeras. I mån av tid blev detta inte möjligt.

Just nu kommer man till en vy där man inte kan se menyn när man loggar in och loggar ut så klickar man på någon av dem finns det ingen återvändo. Detta gjordes för att på ett enklare sätt uppdatera headern med "NOT LOGGED IN" texten och knappen. Detta är också något som är möjligt att fixa genom att skicka information mellan komponenterna i react med hjälp av props. Detta är också något som jag inte fått gjort i mån av tid.

7 Problem och Reflektioner

Jag tyckte att denna labb var väldgit rolig och utmanande. Jag hade sedan tidigare aldrig jobbat i node så det var helt nytt. React var jag lite bekant med då jag gjort några mindre sidor i det men aldrig något som kommunicerar med en databas eller API.

Jag har lärt mig väldigt mycket om hur JavaScript fungerar, hur http-satser fungerar och speciellt hur man kan skapa en säker inloggning till ett API. Det var det som tog absolut mest av tiden men gav också mest.

Jag hade en del problem med att få en koppling mellan databasen och node ibörjan. Det löste sig till slut genom att jag fick byta lösenord till "password" för att det skulle fungera.

Om jag skulle göra om laborationen skulle jag nog vilja pröva att göra databasen av en mongodb. Många av de jag har läst på internet och de videor jag har sätt så har de använt mongodb. Det verkar vara väldigt smidigt och användbart.

8 Referenser

Node och express (ep 1-4) - `https://www.youtube.com/watch?v=F7NVpxxmmgM`
React validering - `https://www.youtube.com/watch?v=FM2RN8rHCTE`
Node säker inloggning (ep 11-13) -
`https://www.youtube.com/watch?v=_EP2qCmLzSE&list=PL55RiY5tL51q4D-B63KBnygU6opNPFk_q&index=11`