Erfarenhetsrapport

2022

—

Webbaserade ramverk och designmönster - Grupp 13

Daniel Lundgren

Hanna Johansson

Jesper Stolt

Introduktion

Krav → design → färdigt system

Kravet för det här projektet var att skapa ett system för att hyra och hantera elsparkcyklar. Systemet skulle bestå av app och webbklient för kunden, webbklient för administrationen och ett cykelprogram för hantering av cyklarnas data. Samtliga delar skulle integrera med vår databas.

För vårt projekt följde vi de teknikval, tekniska lösningar och arkitektur som vi hade som ambition och planerat med från start med undantag för cykelns program. Vi har skapat de olika delarna som den administrativa webbklienten, kundens webbklient och app som samtliga är kopplade till Oauth autentisering. Vi har även koppling för simulering mellan kundens app och cykelprogrammet. Samtliga delar är kopplade till databasen med API-anrop via GraphQL.

Vi har använt Javascript för flera delar av projektet. Vilket förenklar vårt arbete med att kunna hjälpa och guida varandra när vi bygger våra delar då vi arbetar med samma språk och teknik inom gruppen.

Planering, kommunikation, arbetsfördelning.

För att vi skulle få ett lyckat samarbete och bra arbetsfördelning i vårt projektgrupp valde vi att dela upp de olika kraven i vår grupp på ett sätt som vi alla kände oss nöjda och okej med, och vi utsåg även en projektledare, Jesper Stolt. Vi har under projektets gång haft kommunikation mellan varandra och avstämningar för att kunna stötta och hjälpa varandra. Vi planerade tillsammans hur vi skulle lösa de olika delarna och vad vi hade för ambitioner.

Under den senare delen av projektets gång stötte vi på hinder då vår ena gruppmedlem meddelade att han inte kunde lösa sin del av projektet pga personliga hinder. Den delen som vi nu stod utan var cykelprogrammet och vi fick nu tänka om och fundera hur vi skulle hantera det här för att kunna leverera ett fullständigt projekt. Vi diskuterade alternativa lösningar mellan varandra och kom fram till att skapa ett nytt cykelprogram, vilket Jesper Stolt och Daniel Lundgren tog tag i tillsammans och kom fram till en lösning för cykelprogrammet.

Det som vi hade kunnat göra annorlunda är att vara mer inblandade i varandras delar av systemet. Se till att förstå varandras delar i bättre detalj och mer noggrant hänga med i varje dels utveckling. På så sätt hade det blivit tydligare vad varje delsystem behöver av de andra för att det ska fungera som förväntat. Det hade även kunnat undvika problemet med det första cykelprogrammet där det behövde överges helt på grund av dålig kompatibilitet med resten av systemet.

God kommunikation är ytterst viktigt vid genomförandet av ett projekt. Även om kommunikation höll en hög standard till en början blev den bristande samt uppdateringarna glesa när projektet var i full gång och alla var fokuserade på sina egna delar.

Delsystemen

Databas

En databas utvecklades för att lagra all data som varje delsystem behövde för att fungera.

Val av databas

När det kom till valet av databasen fanns det många olika typer att välja mellan och att veta exakt vilken typ man bör använda för våran lösning var ett svårt val. Det första beslutet som vi behövde ta var ifall vi ville ha en SQL eller en NoSQL databas.

Det första förslaget som kom fram var att använda sig av Mongodb. Det fanns flera anledningar till det här förslaget. En första anledning var att mongodb är väldigt enkelt att hosta (från engelska “hosting”) på cloudet. Mongodb sköter det här själv genom en webbplats som går att använda för att hantera databasen. En annan anledning var just den här webbplatsen. Med hjälp av den är det väldigt lätt att få en översikt över all data och snabbt göra direkta ändringar av datan. En tredje anledning var även att de förfrågningar som skickas till databasen från klienten kan anses vara enklare att skriva och tyda än SQL-förfrågningar.

Till en början var vi överens om valet av Mongodb men det varade inte länge. När strukturen på databasen började diskuteras och vi försökte föreställa oss hur vi skulle koppla ihop alla tabeller insåg vi att det var svårt att undvika SQL-tänk. Eftersom att det var svårt att komma överens om en struktur på Mongodb och två av oss inte hade tidigare erfarenhet av denna typ så övergavs det första valet. Alltså hamnade slutliga valet på en SQL-databas och mer specifikt på MySQL på grund av tidigare erfarenheter.

Hosting av databas

Att hosta databasen på en extern server var önskvärt då man redan i utvecklingsstadiet kunde jobba med samma data och ha en viss del kommunikation mellan delsystemen. Att alla jobbade mot en instans databasen underlättade även vid uppdateringar av den. Då kunde databasen helt enkelt byggas om på en plats istället för att alla i gruppen behövde ta emot ny SQL-kod och kör denna mot sin egna instans av databasen. Det här hjälpte även med att undvika problem som kan uppstå ifall någon råkade utveckla mot en äldre version.

Azure

Ett första försök till hosting utfördes med hjälp av Azure. Det här övergavs dock då gratisverktyget som erbjöds för hosting av en databas inte var byggt för MySQL som vi redan hade valt. Det verktyg som fanns tillgängligt för MySQL var otydligt med sitt pris och trots krediten som hade erbjudits av Azure var det otydligt om detta räckte hela vägen till Januari-leveransen.

AWS RDS

Efter att ha övergett Azure försöktes det istället med Amazon Web Services och deras Relational Database Service. Det enda problemet som stöttes på här var att registreringen avbröts mitt i vilket

orsakade en bugg. Den här buggen förhindrade kontot från att fulländas. Detta löstes enkelt genom att använda en annan mail-adress. AWS RDS gjorde det enkelt att starta upp en ny databas-service och gratisversionen som erbjöds kunde användas i upp till ett år.

Skapande av databas

För skapande och vidare kommunikation med databasen användes verktyget MySQL Workbench. I Workbench etablerades det först en uppkoppling mot databasen och därefter ritades alla tabeller med hjälp av EER-diagram. Diagramet sparades för framtida förändringar och utfrån denna genererade Workbench automatiskt kod som användes för att bygga upp databasen. Koden sparades undan i en egen fil och strukturerades på ett sådant sätt att den kunde köras gång på gång utan felmeddelande. Det här förenklade återskapning av databasen vid eventuella uppdateringar.

API

GraphQL vs REST

Det första beslutet som skulle tas angående API:et var ifall vi ville användas oss av GraphQL eller utveckla ett vanligt REST-API. Efter att ha forskat en del om GraphQL övervägdes fördelarna. Den största fördelen var förenklingen av hur man hämtar data som klient. Genom att lägga extra tid på API:et och därmed förenkla alla förfrågningar som skulle göras från klienterna kunde arbetet fördelas mer jämnt. En annan fördel var GraphiQL som gjorde det lätt att testa sina förfrågningar i förväg för att se resultatet direkt. Den sista fördelen som var avgörande var att GraphQL kunde ses som ett “plus i kanten”.

Hosting av API

För att underlätta vid uppdateringar och undvika extrajobbet som skulle komma med att behöva få igång en egen instans av API:et valdes det att även hosta detta hos en extern tjänst.

Azure och AWS

Både Azure och AWS erbjuder tjänster för att publicera en webbapp i en virtuell miljö. Azure hade här samma problem som tidigare. Oklarhet kring vad som var gratis och ifall betaltjänsterna skulle va billiga nog för att inte överskrida krediten. En gratis tjänst prövades men att publicera API:et tog oacceptabelt lång tid och att försöka göra förfrågningar när den väl var publicerad är inte ens värt att nämna.
AWS hade däremot återigen ett bättre erbjudande av gratis tjänster. Här lyckades en virtuell miljö skapas och API:et publicerades dit. Det gick även att starta igång allt men att få tillgång för att göra förfrågningar blev ett stort problem. Det följdes många guider för att öppna portar för åtkomst till API:et men efter att ha lagt ner en hel del tid övergavs även AWS. Smidigheten med att publicera till en extern tjänst övervägdes mot vikten av att få klart API:et så fort som möjligt. Att de skulle bli klart snarast var av större vikt då alla andra delar i systemet förlitade sig på detta.

Docker

Slutligen bestämdes det att Docker skulle användas. En image byggdes med Node som grund och denna publicerades på Docker Hub. Därmed kunde alla gruppmedlemmar med enkelhet hämta hem imagen och

starta igång API:et. Här dök dock ett problem upp som hade undvikits ifall det var publicerat hos en extern tjänst. Det var ofta man trodde att den senaste versionen kördes när det egentligen var en gammal image man inte hade raderat. Däremot var fördelen att man själv kunde välja version utifrån sina egna tillfälliga behov.

Förbättringar

En förbättring som hade gjort stor skillnad under utveckling har att göra med problemet med gamla images som tidigare nämndes. Här hade man enkelt kunnat lösa problemet genom att skriva ut vilken version som kördes i terminalen samtidigt som man skrev ut porten som användes. En annan förbättring hade varit att dela upp all kod i mindre filer. Det stöttes på mycket problem när en uppkoppling mot databasen skulle etableras. Kopplingen behövde vara klar innan GraphQL försökte använda sig av den. Efter att ha bråkat med async, await och en del försök med in sync samlades istället all GraphQL kod i samma fil där uppkopplingen skapades. Återigen vägdes tiden som krävdes för förbättring mot vikten av att blir klar så fort som möjligt. Att bli klar fort var viktigast.

Oauth

Oauth var efter lite googling relativt lätt att förstå. Det första man behövde göra var att registrera sitt projekt på github för oauth med en callback address att skickas till. Som callback användes localhost:666/github/callback som hämtar token och skickar en vidare till route /success. Här började det bli problem för att det är flera olika klienter som använder OAuth och behöver därför kunna skicka till flera olika sidor. Hade det bara vart en sida som behövde OAuth så hade man därifrån statiskt kunnat redirecta till samma länk.
Jag kollade då på ifall det gick att skapa en array med länkar för att skicka till en speciell sida beroende på vart man kom ifrån. Det gick inte att se vart man kom ifrån, men det gick däremot att skicka med parameter callback, till url för github. Den skickades då sedan med till /github/callback när verifieringen blivit gjord. Med hjälp av det var det möjligt att se vart man kom ifrån. Jag kunde således skapa en array för länkar till respektive sida. Det dök dock då upp nya problem iom att jag inte visste vilken localhost jag skulle skicka tillbaka till och vilken route på den localhost. Det blev också mer manuell uppdatering än önskat om det skulle läggas till fler delar i systemet.

Slutgiltig lösning

Lösningen blev då att skicka med url, i callbackparameter. Där uppstod ett problem med att skickar med en länk med hash-routing så bröts länken efter #. Det löstes genom att man istället för #! Skickade med enbart ! så kompleterades routen med # innan man skickades vidare. För att sedan hämta data så använder man sig av användar-id vilket fås av github. Med det som parameter anropar man route /data på oauth klienten, vilket returnerar användaren.

Cykel

Vi behövde revidera utformningen av cykeln för att få den att fungera tillsammans med simuleringen vilket var huvudmålet för hela programmet

Version 1 - Cordova

Vi valde först att skapa en app med hjälp av cordova för att kunna utnyttja dess plugin för att hämta hastighet och position. Det för att underlätta dess arbete. Allt fungerade som tänkt fram till dess att vi skulle börja använda den i simulering. Då vi kom fram till att vi behövde skapa en dockerimage för varje cykel, då varje cykel behövde en egen port. Utöver det, behövde då varje cykel skapas en service för i vår docker-compose fil och porten för varje cykel behövdes öppnas för portforwarding från docker till windows systemet. För att möjliggöra kommunikation mellan cyklar och övriga delar av systemet. Vi kom fram till att det var en metod som skulle fungera för ett par cyklar, men inte vara hållbart för att skapa 1000 cyklar. Därför skrotades den iden.

Version 2 - Klass

Vi kom då istället fram till att vi behövde skapa samtliga cyklar oavsett antal från samma fil. Vi valde i slutändan att skapa en klass, vilket gjorde det enkelt att skapa ett visst antal cyklar, då det bara var att skapa en ny instans av cykelklassen. Vi gick dock då miste om cordovas plugin för att få hastighet. Vi hade dock möjlighet att beräkna det med hjälp av två punkter och radien på jorden. Utöver det så behövdes setInterval användas för att kunna skicka och ta emot data med bestämda intervall. Det som återstod var något sätt att manuellt ändra cykelns position istället för att förlita sig på var enheten var för att kunna utnyttja samma klass till simuleringen. Det löstes genom en metod för att överskrida gps och istället manuellt skicka in nya positioner.

Kunden

App

Appen gjordes med hjälp av Cordova och Mithril samt leaflet för kartorna. Här var inga större problem utan skapandet av appen flöt på bra med hjälp av kunskaperna från webapp-kursen. Det gjordes även en studie på två olika qr-läsare, vilket publicerades på discord för övriga gruppers läsning. Den valda QR-scanner implementerades även för att kunna scanna cyklars QR-koder vid hyrning. All fungerade som önskat fram till automatisering av tester och skapande av docker filer. Mer om det efter genomgång av webbklienten.

Webklient

Webbklienten skapades från början med hjälp av enbart Mithril och leaflet för kartor. Vilket fungerade som tänkt dock saknades då kommando likt cordova run browser för att starta webbapplikationen. Först kollades på Mithril-app vilket var Mithrils sätt att få igång en server. Däremot föll valet även för webbklienten på Cordova för att skapa enhetlighet, mellan delarna då samtliga grafiska delar då kör Cordova. Efter att strukturerat om filerna för att passa Cordovas struktur, flöt även här arbetet på enligt plan fram till automatiserade tester och docker.

Docker

Tidigare när arbete med docker och skapande av Dockerfiler har strukturen kunnat se ut som nedan:

FROM node:14
RUN apt-get update && apt-get install
WORKDIR <mapp-namn>
COPY package.json och filer som behövs för att kunna köra klienten
RUN npm install
ENTRYPOINT ["npm"]
CMD ["start"]
Det fungerade inte för applikationer som var skapade med hjälp av Cordova. Det fick då upp att det inte hittades kommando Cordova run browser, vilket är kommandot som startar en Cordova-klient i webbläsaren. Då testades att kopiera med mapparna som Cordova skapade när man initierar ett Cordova-projekt. Resultatet blev fortfarande detsamma. Nästa steg blev då att i dockermiljön installera cordova, innan dess att filer kopieras eller RUN npm install körs. Resultatet blev fortfarande detsamma. Efter ytterligare felsökning så blev domen att containern inte visste att det var ett Cordova projekt trots att alla Cordova mappar kopierades över. För att komma runt det, skapades därför ett Cordova projekt i Containern innan dess att kopieras eller RUN npm install körs. Det tillsammans med att browser lades till som plattform för Cordova gjorde att det slutligen gick att köra dockerfilen som önskat.

Tester

Den andra delen som skapade problem för kundens delar var när testerna skulle automatiseras, eller köras över huvud taget.

Export av moduler

För att kunna komma åt en modul från andra moduler så använde vi oss i stor utsträckning av Export default i slutet av JavaScript-modulerna. Det skapade det skapade problem då det inte är helt kompatibelt med webbläsaren utan mer med Node. Vi testade då att övergå till Module.exports, men det gav då andra fel att filer inte sågs som moduler av testverktygen. Så det letades vidare efter lösningar. Efter att hittat plugin för via @babel, så kunde testerna hantera Default Export, men den hade istället problem med att hitta, window, screen och andra globala element, då testerna inte körs i en visuell miljö. Även där fanns hjälp via @Babel och det kombinerat med bibliotek från jsdom, gjorde att testerna tillslut kunde köras lokalt.

Automatisering av tester

Nästa steg blev att automatisera testerna, första valet föll på Travis och Scrutinizer för att det inom dessa fanns kunskaper sedan tidigare kurser. Däremot uppstod problem med Travis, då vi använt det i flera kurser tidigare och det finns en gräns för hur många tester som går att köra. Då tack vare tips på discord öppnades ögonen för Github Actions, vilket kunde köras när en ny verision pushades till Github. Det fungerade smidigt att sätta upp så när som gällande miljövariabler. Miljövariablerna sattes upp som dolda i gitrepot och hämtades sedan med hjälp av secrets.<variabelnamnn> däremot var det inte tydligt beskrivet var dessa skulle hämtas ut, det första som försöktes var att sätta dessa som ett steg i arbetsflödet. Det problem som då uppstod var att variablerna inte hittades utanför det steget. Efter mycket felsökande hittades att dessa skulle placeras innan jobb stegen startades under en egen del med namn env. Tack vare discord fick vi även tips om Codecov för att se hur bra kodtäckning ens tester har. Det gick enkelt att integrera inom samma Github action.

Simulering

Vi hade från start svårt att få en bild över hur simuleringen skulle fungera, det kan även vara en anledning till att vår Cykelklient behövde göras om från grunden. När simuleringsprogrammet skapades, gjordes en upptäckt gällande att cykelns konstruktion behövde vara asynkron, vilket Klassers inom JavaScript inte stödjer i sin konstruktionsmetod. Däremot är det möjligt att returnera önskat inom konstruktionen. Därmed gick det att göra det som returnerades av konstruktionen asynkront för att gå runt denna begränsning. Nästa problem var att programmet blev väldigt långsamt när 1000 cyklar skulle skapas samtidigt när programmet startades. Det löstes att cykeln inte skapades förens det var dag att hyra den. Vilket gav bättre flytt och gjorde att själva simuleringen fungerade.
Ett annat problem som istället uppstod var att eftersom Node/Express, som API-körs med hanterar ett anrop åt gången, blev hämtningen av data för andra delar av systemet väldigt långsam med flera minuter långa hämtningstider. Det berodde på att ett anrop för att uppdatera cykeln tog ca 100-200ms, när det var som flest anrop per sekund var det nästan 20 anrop per sekund. Det innebär att det behövs 2-4s av tid för att klara varje sekunds anrop. Det i sin tur gör att kön för att utföra anrop hela tiden växer. Det var något vi även upplevde genom att det ju längre tid som gick desto längre tid tog det att hämta data, för en annan del av systemet. För att lösa det valde vi att skapa en extra api-klient för att hantera simuleringens anrop. Det gav resultatet att Simuleringens kö fortfarande växer men att det inte påverkar övriga delar av systemet. Vårt argument för att se det som en fungerande lösning, är att i ett riktigt system sker inte anropen linjärt utan flera anrop hanteras samtidigt, det simulerar vi med hjälp av två portar.

Docker

För docker var det två problem vi behövde lösa dels att simuleringen inte ville prata med andra delar av systemet, samt att simuleringen inte startar per automatik när man kör docker-compose up.

Kommunikation inom systemet

Det första problemet var att simuleringen inte ville prata med sin API-client, det var inget som skapade problem för någon annan del av systemet. Det kopplades till att simuleringen själv inte kördes på någon port. För att lösa det så användes links i konfigurationen av simuleringens docker-compose service. Då användes server:1338 istället för localhost:1338 för att prata med API.

Förhindra automatiskt start av simulering

För att man inte alltid behöver köra igång en simulering när man kör docker-compose up, så utnyttjades något som heter profiles, vilket specificerar vilka profiler en service startas med, för simuleringen specificerades simulation som profil, för att starta den kör man då antingen docker-compose run simulation eller docker-compose –profiles simulation up

Admin

Från krav till färdigt system

Kravspecifikationen för den administrativa webbklienten var att presenterar städer, cyklar, laddstationer/parkeringar och kunder, samt göra det möjligt att förflytta cyklar till laddstationer eller parkeringar. Det skulle även finnas en karta med en översikt över positionen för cyklarna och städernas laddstationer.

Vår design/arkitektur över webbklienten innefattar en sida med inloggning som leder till en översikt över samtliga städer i systemet. Efter val av stad presenteras karta som visar laddstationer, parkeringar och cyklarnas position i staden. Från den här sidan ville vi att det skulle finnas en meny med val att gå till sida för kunder, parkeringar och cyklar. Sidan för kunder skulle visa en vy över de kunder som är registrerade i systemet, sidan för parkeringar skull visa en lista över parkeringar och de parkerade cyklarna. Sidan för cyklarna skulle visa en sida för cyklarna och deras batteristatus, för att sedan kunna göra det möjligt att välja cykel och därefter genomföra en flytt av cykel till önskad parkering/laddstation.

Den ursprungliga idén över designen/arkitekturen för webbklienten förverkligades och hölls vid under genomförandet av sidan. Dock blev det justeringar för menyvalet för parkeringar och cyklarna, då istället för att ha menyval till dessa sidor är det möjligt att välja cykel eller parkering direkt via ikonerna på kartan för att göra interaktionen med kartan mer intressant. På kartan visas även ytterligare information om cyklarna och parkeringar som exempelvis batteri, antal parkerade cyklar, laddstation m.m.

Utmaningar och tekniska lösningar

Det ramverk och programmeringsspråk som valdes till den administrativa webbklienten var JavaScript med Mithril.

Enligt kravspecifikationen skulle det finnas en karta som markerar positioner för cyklarna och även de godkända parkeringsplatserna, för att lösa det här valde vi att använda Leaflet. Vi hade en önskan om att det skulle vara möjligt att se hur cyklarna rör på sig tillsammans med simuleringen, vilket vi löste genom att skapa ett tidsintervall med ’setInterval()’ så att cyklarnas position i kartan uppdateras var tionde sekund och därmed kan man följa cyklarnas rörelse. För att det inte ska upplevas som att hela sidan uppdateras valde vi att lösa det här genom att jobba med lager för kartan, så vi placerade cyklarnas markering i ett lager som vi sen gjorde tidsintervallet för, därav behålls kartan och parkeringsplatserna medans cyklarna uppdateras.

För att komma åt innehållet inne på administrationens webbklient skapades applikationen Oauth av Jesper Stolt som applicerades på bl.a webbklienten och gjorde det möjligt att välja GitHub som inloggningssystem för sidan.

För att lyckas få all information med cyklar, parkeringar, kunder m.m från databasen användes Axios som hanterar anropen till databasen via GraphQL. Till en början var det en utmaning att hantera anropen till databasen via Axios och GraphQL, men genom forskning och kommunikation mellan gruppen lyckades vi lösa anropen och hämta informationen från databasen.

Sammanfattning

Det slutliga resultatet kom mycket nära den bild vi hade av systemet i början av kursen. Den största skillnaden mellan slutresultatet och den initiala bilden var hur cykelprogrammet fungerar. Det gick från att vara ett program skapat med hjälp av Cordova för användning av plugins för position och hastighet till att vara en mycket enklare klass med ett fåtal metoder. Tack vare god kommunikation och en vilja att hjälpa överkom vi flera problem som var svåra att hantera själva. Däremot hade mer frekventa uppdateringar om varje delsystems tillstånd kunnat möjliggöra en tidigare leverans av systemet redan i December. Nu är dock kraven uppfyllda och allt fungerar som förväntat. Systemet är alltså redo för leverans.