System Design Specification (SDS)
för projektet
Svenska Elsparkcyklar AB

Version 1.0.0

Grupp 8

David Dahlgren (dadh22)
Melissa Johansson (mejo22)
Gloria Palm (glpa22)
Jonas Bolinder (jobf22)

Innehållsförteckning

Introduktion

En System Design Specification (SDS) ämnar att ge en beskrivning av systemet som skall byggas från många infallsvinklar. Vi väljer att följa andan i den klassiska artikeln “Architectural Blueprints—The “4+1” View Model of Software Architecture” [1]. 4+1-modellen går ut på att beskriva systemet från olika vinklar och till olika målgrupper.

De fyra vyerna av systemet, enligt den klassiska definitionen, är en logisk vy, en process-vy, en utvecklare och en fysisk vy, som tillsammans beskriver systemets arkitektur. En femte vy (1:an i 4+1) består av några utvalda användningsfall eller scenarion som illustrerar de 4 arkitekturella vyerna. Vi reducerar antalet vyer något i den här specifikationen, och bakar in process-vyn och den fysiska vyn i övriga vyer, så att vi snarare får en 2+1-vy.

Den logiska vyn beskriver systemets funktionella krav, dvs vilka tjänster systemet skall erbjuda användarna. Vi väljer att i den logiska vyn ge en högnivå beskrivning av systemet, som är tillgängligt för slutkunden, användare och andra intressenter.

Process-vyn behandlar dynamiska aspekter av systemet [2], dvs systemets delprocesser och hur de interagerar. Det kan handla om concurrency (samtidighet), distribution, prestanda och skalbarhet. I det här projektet kommer systemet endast köras i en Docker testmiljö, på en och samma dator, så vi utelämnar den här vyn, och inlemmar relevanta delar i utvecklarvyn istället.

Utvecklarvyn riktar sig mot mjukvaruutvecklaren eller programmeraren av systemet. Den kan ta upp aspekter som vilka ramverk och programmeringsspråk som används, databas, designmönster, t ex “layered”, “event-driven” eller “microservices” [3]. Också vilka API:er som finns och hur de är definierade.

Den fysiska vyn visar hur systemets processer mappas mot fysisk hårdvara. Den blir ett mindre fokus i det här designdokumentet, som i huvudsak behandlar mjukvaruaspekter, och utelämnas därför.

Slutligen scenarierna, eller användningsfallen tjänar till, som nämnts ovan, att illustrera systemets funktion genom att visa på interaktionen mellan användare och processer. De kan också vara en utgångspunkt för test och verifiering av systemet.

Bakgrund

Företaget “Svenska Elsparkcyklar AB” bedriver uthyrningsverksamhet av elsparkcyklar i tre svenska städer och planerar att utöka sin verksamhet till fler städer. De är därför i behov av ett nytt datasystem som hanterar cykeluthyrningen.

Det nya datasystemet kommer innehålla följande delar:

- Ett program som hanterar och övervakar cykeln (på, av, hastighet, begränsa hastighet, position, behöver service/laddning).
- För administratörer ett webbaserat gränssnitt för att hantera kunder, cyklar och databas.
- För kunder ett webbaserat gränssnitt för att ändra kontouppgifter, fylla på pengar, se historik över hyrda cyklar och betalningar.
- En mobilanpassad applikation eller webbplats där kunden kan hyra och återlämna cyklar, se historik över gjorda resor samt status på senaste resan.
- En funktion för att simulera hela systemet i drift för att testa dess funktioner
- Ett REST API med möjlighet att koppla in anpassade applikationer.

Bilden nedan ger en överblick över systemet i sin helhet.

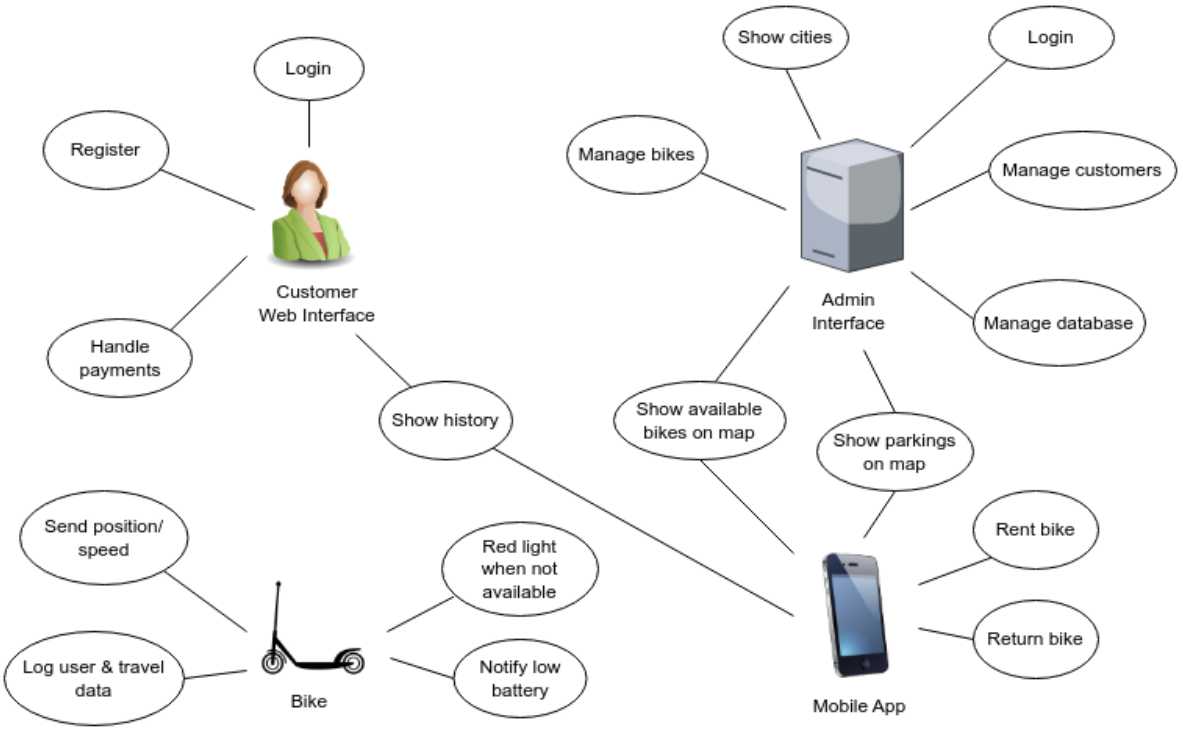


Bild 1. Systemets klienter med några användningsfall

Logisk vy – En övergripande beskrivning av systemet

Vi tänker oss att systemet har en central webbserver med en databas, och att klientapplikationer ansluter till servern via HTTP. Ett REST API [4] finns tillgängligt för att hämta och lagra data i databasen enligt följande principiella skiss.

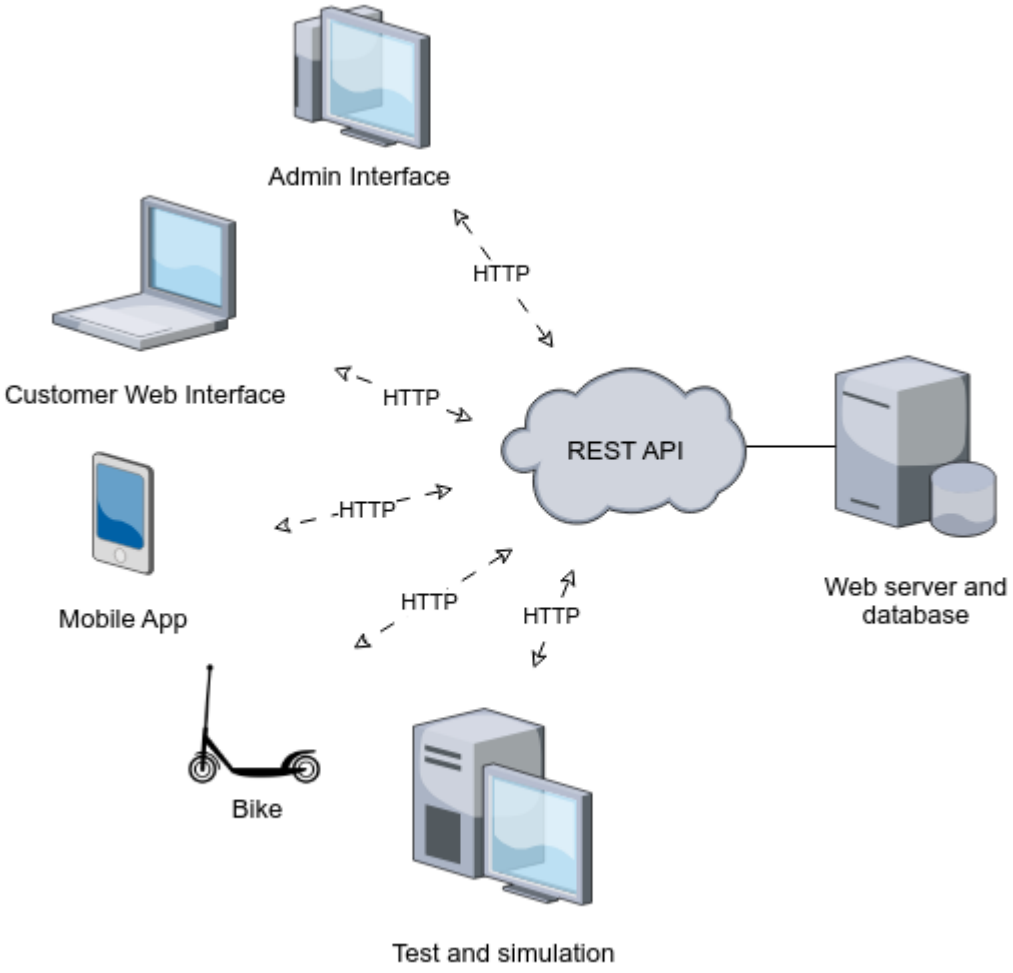


Bild 2. Översikt över systemet och hur klienterna kommunicerar med en central server via HTTP och ett REST API.

Server och databas

Webbservern använder Node.js och ramverket Express för att serva klienterna med webbsidor och med JSON-data, som svar på frågor via REST API:et. Som databas används en SQL-baserad som SQLite, alternativt MariaDB, vilket kan vara lämpligt då vi har en central instans av en databas, med en potentiellt lite högre belastning.

Klientapplikationer

För klientapplikationerna till admin och kund, samt troligen mobilappen planerar vi att använda JavaScript-ramverket React. I mån av tid finns det funderingar på att använda Apache Cordova för att skapa en native mobilapp. För cykelns app, som saknar användargränssnitt (UI), kan vi använda en enklare lösning (cykelns app behöver egentligen bara fungera i kontexten av test och simulering).

Nedan följer en beskrivning av de olika klientapplikationerna.

Kundens webbgränssnitt

Kundens webbgränssnitt tar sin början i att kunden navigerar till en publik webbplats för sparkcykelföretaget och loggar in. Därefter får kunden möjlighet att uppdatera sin personliga information och lägga till antingen pengar på sitt konto, eller att ansluta någon typ av betalningstjänst (t ex betalkort). Kunden har även en möjlighet att se en historik över sina tidigare resor och betalningar. All information hämtas från databasen via REST API:et.

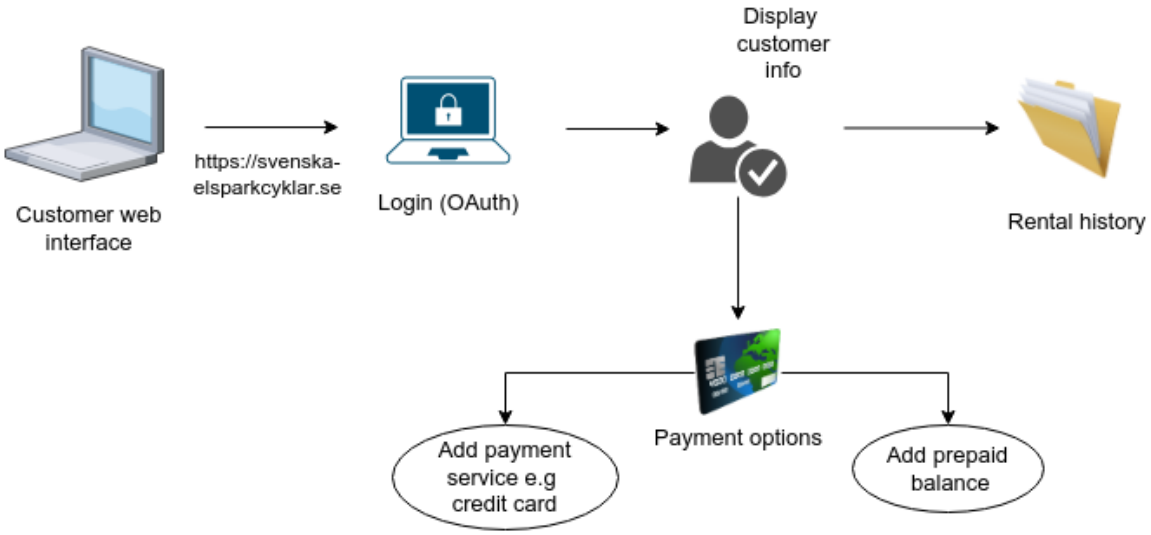


Bild 3. Bild över flödet i kundens webbgränssnitt.

Kundens mobil-app

Första gången en kund använder appen loggar den in med sina uppgifter, som hämtas från databasens kundtabell. Inne i appen har kunden möjlighet att se en karta över tillgängliga cyklar, var parkeringszoner finns, samt platser där man kan ladda cykeln. När kunden valt en cykel kan den hyra densamma. Om kunden redan hyr en cykel finns även möjligheten att lämna tillbaka cykeln.

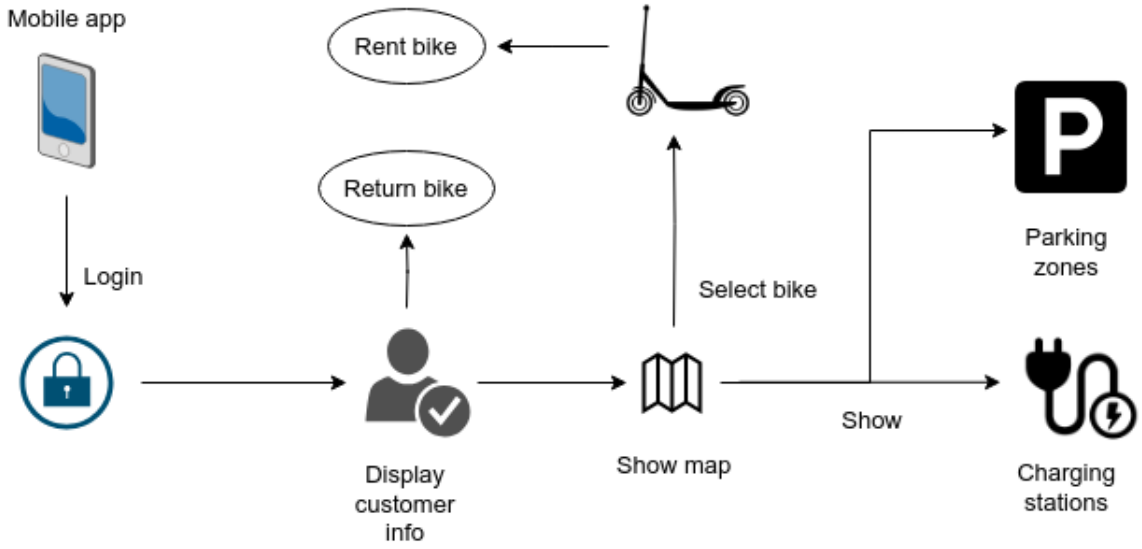


Bild 4. Bild över flödet i mobilappen.

Admins webbgränssnitt

Admins webbgränssnitt tar sin början i att man loggar in. Väl inloggad har administratören tillgång till en rad olika funktioner. Det finns möjlighet att se alla kunder och vid behov kontakta och stänga av kunder som missköter sig.

Administratören kan även välja att se alla städer och därifrån samtliga cyklar, parkeringszoner och laddstationer för varje stad. Det finns även en kartfunktion som visar var dessa befinner sig.

Går administratören in på en specifik cykel finns det möjlighet att begära underhåll. T ex om cykeln står på fri parkering och behöver flyttas till en parkeringszon, eller om den behöver laddas alternativt ges service.

Det skall gå att lägga till och ta bort städer med tillhörande laddstationer, zoner och cyklar. Möjlighet att ändra och lägga till zoner är en funktion som övervägs i mån av tid. Även hantering av tillåtna zoner och zoner med begränsad hastighet ser vi som något extra.

Vidare skall det eventuellt vara möjligt att ändra avgiften för att hyra en cykel (startavgift, minutavgift, straffavgift och eventuell rabatt).

Det skall gå att titta på loggen över alla händelser i systemet.

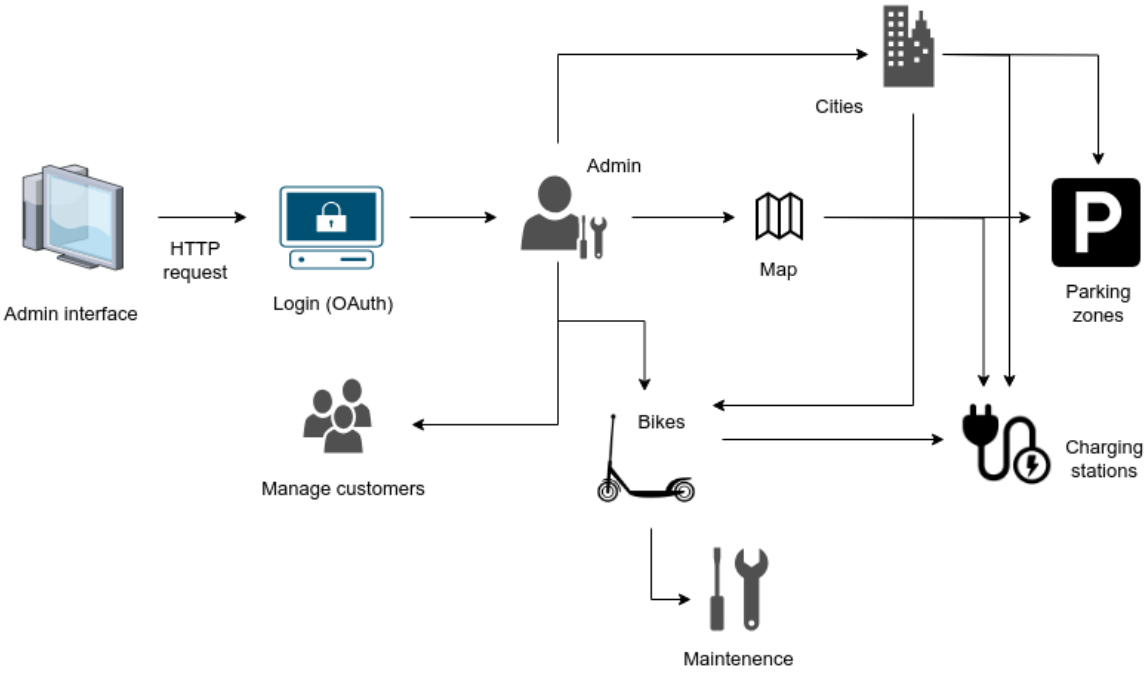


Bild 5. Bild över flödet i admins webbgränssnitt.

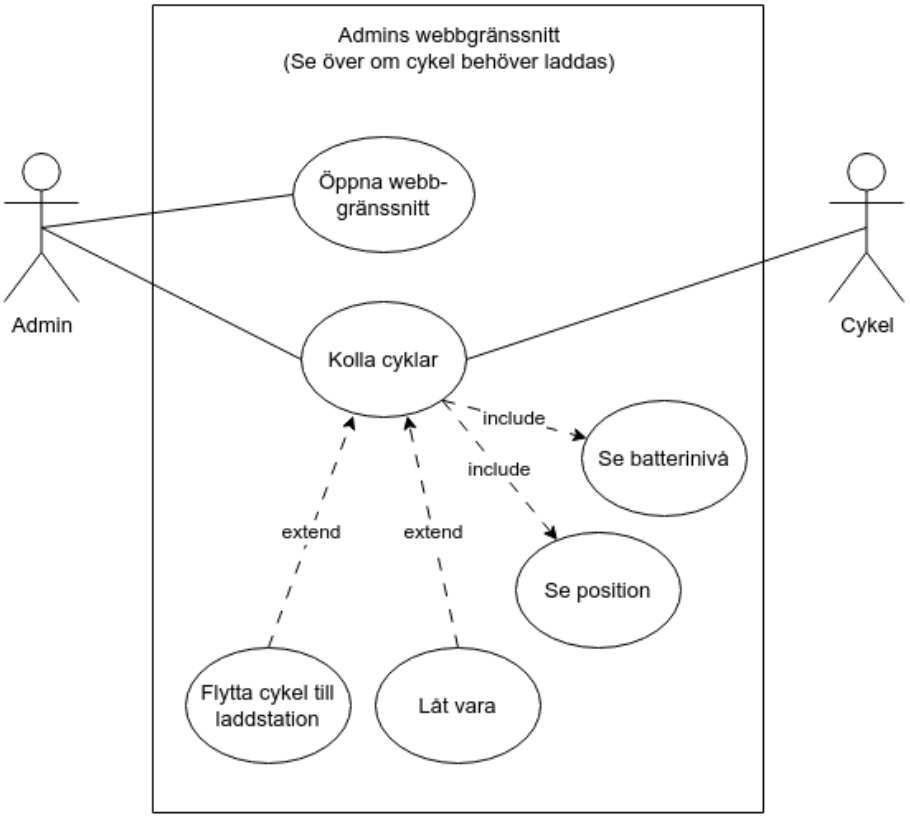
Några användningsfall (scenarier) som systemet stödjer

Att fånga det dynamiska beteendet är en viktig aspekt för att modellera ett system. Med dynamiskt beteende menar vi systemets beteende när det är i drift. Att enbart studera det statiska beteendet är ofta inte tillräckligt, snarare är dynamiskt beteende ofta viktigare för att förstå och modellera ett system.

Några exempel på användningsfallsdiagram (use-case diagrams) visas nedan för de olika klientapplikationerna. Ett diagram består av aktörer, användningsfall och deras relationer. Ett include-förhållande mellan ett basanvändningsfall och ett annat användningsfall innebär att basfallet alltid inkluderar det andra fallet. Ett extend-förhållande å andra sidan innebär att inkluderandet av det andra fallet är optionellt.

Admins webbgränssnitt

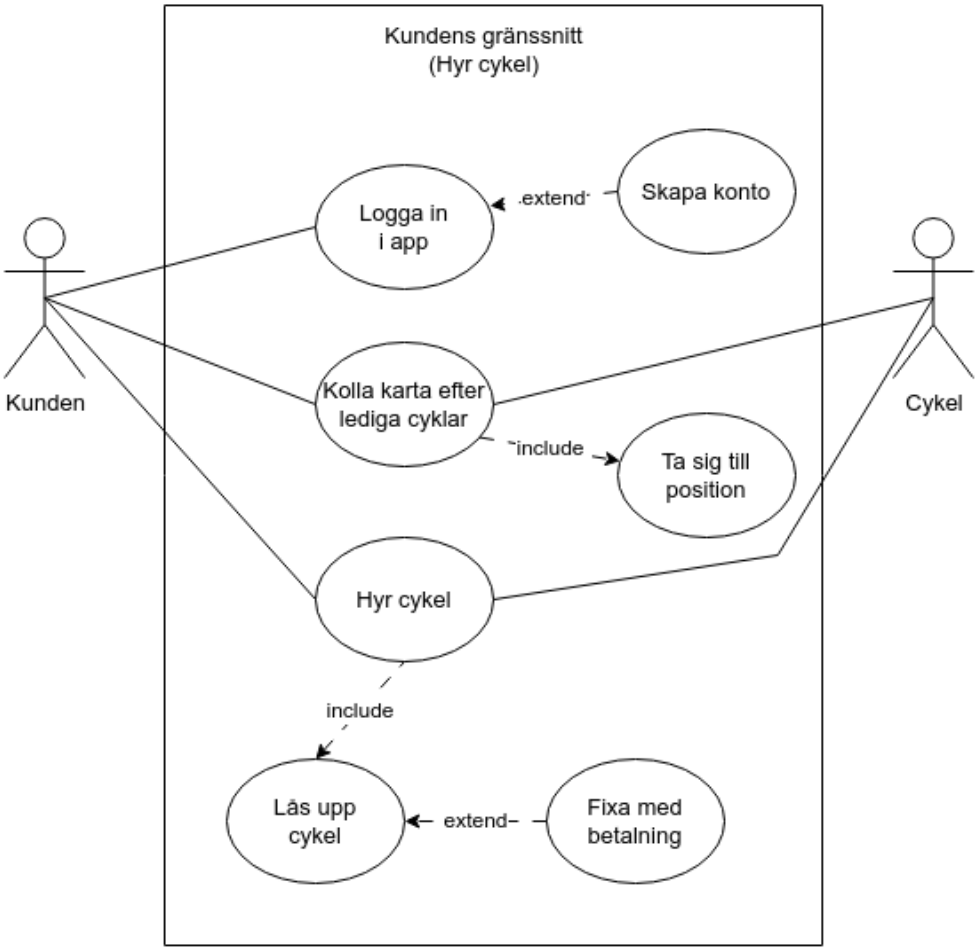
Admin loggar in i webbgränssnittet. Där får hen en överblick över alla cyklar och kan snabbt se om någon cykel behöver laddas, då cyklarna kontinuerligt skickar ut denna information. Admin kan även se cykelns position och kan begära att flytta urladdade cyklar till närmaste laddstation.



Use case-diagram 1. Användningsfallet “admin ser över vilka cyklar som behöver laddas”.

Kundens gränssnitt

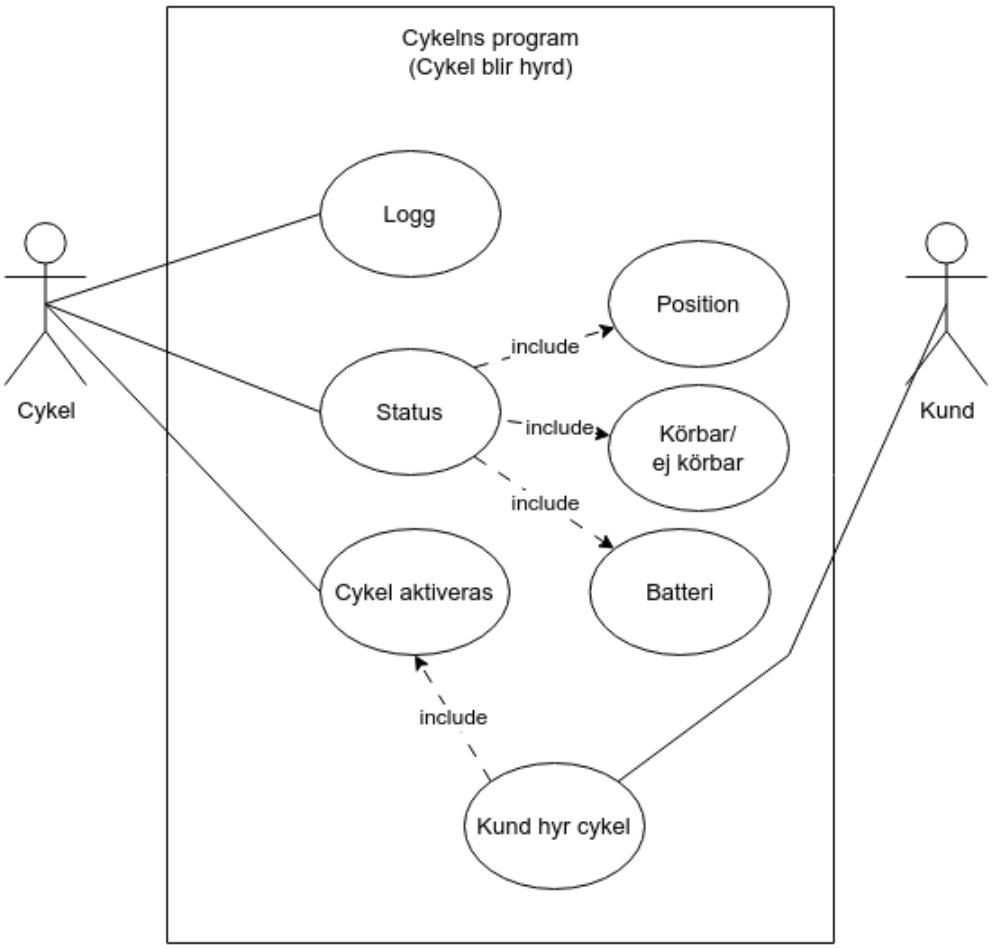
Kunden loggar in i sin app, via existerande konto eller skapar ett nytt. Med en karta kan kunden lätt se vilka cyklar som finns tillgängliga och var de befinner sig, via information som cykelns program sänder ut. Kunden tar sig till en position där en ledig cykel finns och låser med hjälp av appen upp cykeln. Om kunden inte har betalningen uppsatt på sitt konto behöver detta fixas. Kunden kan sedan börja cykla.



Use case-diagram 2. Användningsfallet “kunden hyr en cykel via app”.

Cykelns program

Cykelns program skickar kontinuerligt ut en logg över resor och skickar även ut sin status. Där ingår dess position, om den är körbar eller avställd, samt om batterinivå. Cykeln aktiveras för körning när en kund har aktiverat den.



Use case-diagram 3. Användningsfallet att cykeln blir hyrd ur cykelns perspektiv.

Utvecklarvy – Arkitektur, metoder och tekniker

En grundidé i designen av systemet är att vi skall använda oss av beprövade tekniker, och hålla oss till enkla och okomplicerade lösningar. Detta är framförallt för att tiden för utveckling av systemet är begränsad till några få veckor, och också för att utvecklingsteamets erfarenhet är begränsad.

Alltså konstruerar vi en lösning med en central webbserver och databas. Tanken är att vi bygger ett REST API, och att samtliga klienter använder sig av det för att hämta och lagra data i databasen.

Backend med webbserver och databas

En webbserver med Node.js och ramverket Express används för att serva klienter med både vanliga webbsidor i HTML och med JSON-data, som svar på frågor via REST API:et. Andra statiska resurser som bilder, JavaScript-filer och CSS servas förstås också.

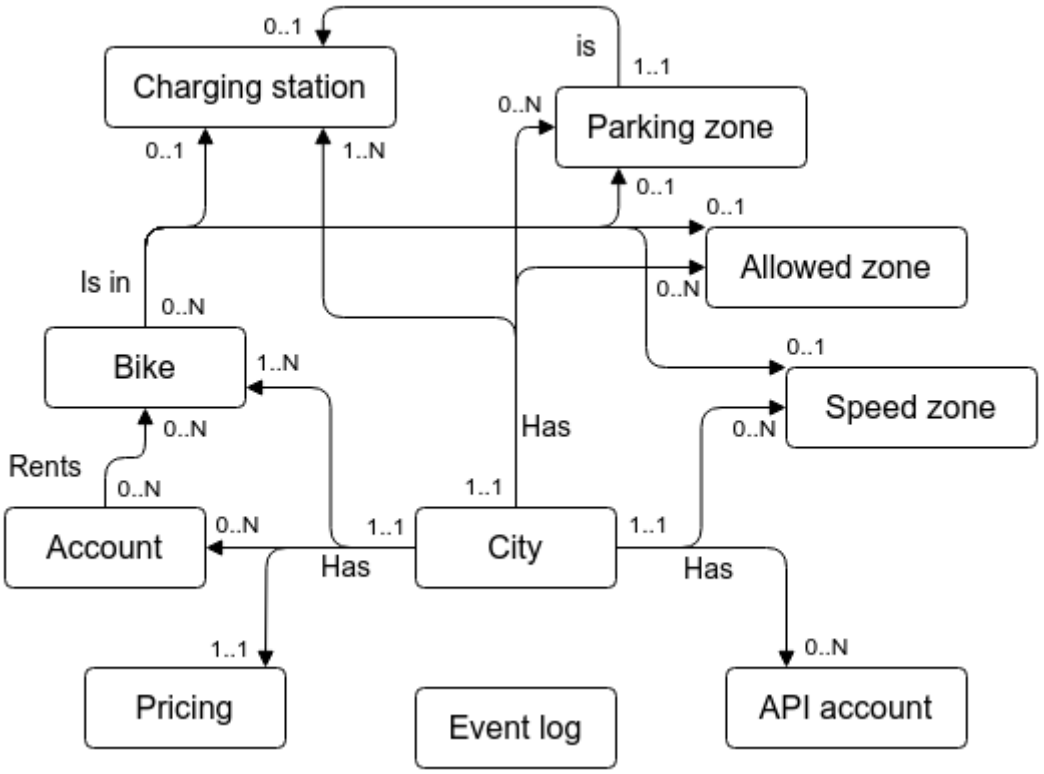
Som databas används lättviktiga SQLite, alternativt MariaDB. Ett annat alternativ hade varit att använda en dokumentbaserad databas som MongoDB. En sådan passar bra framförallt i en distribuerad miljö, med databasen är utspridd över flera datacenter. I vårt fall har vi en central databas, med en potentiellt högre belastning, vilket kan tala för att använda en SQL-databas. Om vi ser behov av att använda så kallade lagrade procedurer, så är MariaDB lämplig, annars kan vi klara oss med den enklare filbaserade SQLite.

Konceptuell modellering av databasen

Ett försök har gjorts att skapa en konceptuell modell av databasen enligt en flerstegsmodell eller “kokbok” [5]. Först beskriver man i ett textstycke vad databasen skall klara av att hantera utifrån kraven i kravspecifikationen. Därefter kan man identifiera entiteter som substantiv i den beskrivande texten. Följande entiteter framträdde då (med en engelsk beteckning):

- City
- Bike (scooter är mera korrekt, men ett längre ord)
- Charging station
- Parking zone
- Allowed zone
- Speed zone
- Account (eller user, customer)
- Pricing (start fee, per minute, extra, discount)
- API account
- Event log

Därefter försöker man hitta relationer (verben) mellan entiteterna, till exempel genom att skapa/rita en tabell. Nästa steg är att rita ett så kallat ER-diagram (Entity-Relationship diagram). Ett sådant med kardinalitet utskriven kan ta sig ut enligt följande:



ER-diagram 1. ER-diagram för databasen med kardinalitet.

Av diagrammet följer att “City” är en som “spindel i nätet”, och har en relation till i stort sett alla övriga entiteter.

Sista steget i den konceptuella modelleringen är att ange attributen till entiteterna, dvs fälten eller kolumnerna i en SQL-databas. Det kan bli lite oöverskådligt att rita ut i ett ER-diagram, speciellt om det är många attribut, så det görs inte här. Dessa kommer att framgå vid en fortsatt logisk eller fysisk modellering av databasen.

Några kommentarer och funderingar till den konceptuella modellen är som följer:

- Med “Account” ovan avses kundens konto. Detta kan möjligen vara det samma för alla städer, om en kund hyr cyklar i olika städer. Det kan också behövas konton för de anställda på sparkcykelföretaget. Sådana inloggningskonton är något helt annat än vad som avses med “Account” ovan.
- En “Charging station” kan ses som ett specialfall av en “Parking zone". Man skulle kunna ha ett attribut “is charging station” för att särskilja dem.
- En “Speed zone” kan ses som ett specialfall av en “Allowed zone” med ett särskiljande attribut. I en första implementering av systemet utelämnar vi sannolikt hastighetszoner och även tillåtna zoner, eftersom de kan vara lite knepiga att få till.
- Med “API account” menas konton till externa appar, som skall kunna ansluta till systemet. Denna funktion utelämnar vi också i en första version.

REST API

Klienterna kommunicerar som nämnts med servern via ett REST API, för att kunna lagra och hämta information från databasen. Ett REST API definieras av så kallade endpoints (URL:er) och vilka metoder som används (GET, POST, PUT och DELETE). API:et är versionshanterat genom att till exempel v1/ läggs till URL:en, detta är utelämnat i tabellen nedan.

Vi har “city” som håller staden och detaljer som vilka parkeringar, laddstationer och zoner som finns där. “city/{id}/station” visar detaljer om de olika parkeringarna och laddstationerna – som vilka cyklar som befinner sig där och om platsen är fullsatt med cyklar och/eller har lediga cyklar att hyra, samt koordinater och id samt typ (parkering/laddstation). Information om zonerna är till exempel var zonen finns och om det finns hastighetsbegränsningar där.

Sedan kommer “bike” som innehåller detaljer om cyklarna och dess position, batteri, hastighet och om den är körbar eller uthyrd. Det finns också en “bike/{id}/rent” som sköter uthyrning av cyklarna. Där skickas användarens id in som JSON via HTTP-bodyn istället för i url:en och visar vem som hyrt cykeln. Detaljer som “rent-id”, var cykeln hyrdes från, planerad destination, vilken kund som hyrt cykeln, om uthyrningen är aktiv, start- och slutdatum finns där. Samma endpoint kan sedan användas för att avsluta uthyrningen genom att uppdatera om uthyrningen är aktiv. Genom att inte göra delete när en cykel lämnas tillbaka sparas informationen om uthyrningen och kan användas för att få fram uthyrningshistorik.

Vi har också “user” som innehåller kunddetaljer som id, namn, betalningsinformation och om kunden hyr en cykel för tillfället.

endpoints	GET	POST	PUT	DELETE
/city	Hämta detaljer om alla städer	Lägga till ny stad	-	-
/city/{id}	Hämta detaljer om staden	-	Uppdatera stad	
/city/{id}/station	Hämta detaljer om alla stadens parkeringar och laddstationer	Lägga till ny parkering eller laddstation	-	-
/city/{id}/station/{station-id}	Hämta detaljer om parkering eller laddstation	-	Uppdatera parkering eller laddstation	Ta bort parkering eller laddstation
/city/{id}/zone	Hämta alla stadens zoner	Lägga till zon	-	-
/city/{id}/zone/{zone-id}	Hämta zon	-	Uppdatera zon	Ta bort zon
/city/{id}/bike	Hämta alla cyklar i staden	Lägga till cykel i staden	-	Ta bort cykel från staden

/bike	Hämta detaljer om alla cyklar	Lägga till cykel i systemet	-	-
/bike/{id}	Hämta detaljer om cykeln	-	Uppdatera cykel	-
/bike/{id}/rent	Hämta detaljer om uthyrning	Skicka in {user id} som JSON i HTTP-body för att lägga till uthyrning	Uppdatera cykelns uthyrning	-
/user	Hämta detaljer om alla kunder	Lägga till kund	-	-
/user/{id}	Hämta detaljer om kund	-	Uppdatera kunddetaljer	Ta bort kund

Tabell 1. REST API (ej komplett)

Klientapplikationer

Som tidigare beskrivits kommer klientapplikationerna bestå av ett tredelat koncept, där kunden har dels ett webbgränssnitt dels en mobil-applikation samt ett webbaserat gränssnitt för administratörer.

Kundens mobil-app

En applikation i grund och botten skapad i Javascript, eventuellt med stöd av ramverket React, alternativt React Native eller Cordova. De två sistnämnda kan användas om man istället föredrar en fristående app framför en mer traditionellt responsiv webbplats. Exakt vilken approach vi kommer använda oss av görs efter att vi slutfört vår riskanalys och bedömt hur lång tid de olika teknikerna kan uppskattas ta.

Applikationen kommer använda någon form av karttjänst, till exempel leaflet och OpenStreetMap för att placera ut cyklar, laddstationer och parkeringszoner på kartan. Dessa hämtas via ovan beskrivet REST-API. Kundens egna position på kartan hämtas via GPS. Med hjälp av detta kan kunden enkelt se hur långt det är till varje cykel.

För att hyra en cykel anger kunden cykelns id, alternativt scannar cykelns QR-kod varpå cykeln låses upp och är redo för användning. Kunden har sedan möjlighet att när som helst via appen avsluta resan och cykeln låses åter. Vid avslut dras pengar från kundens saldo, alternativt läggs till på kundens månadsfaktura, beroende på vad kunden valt för typ av betalning.

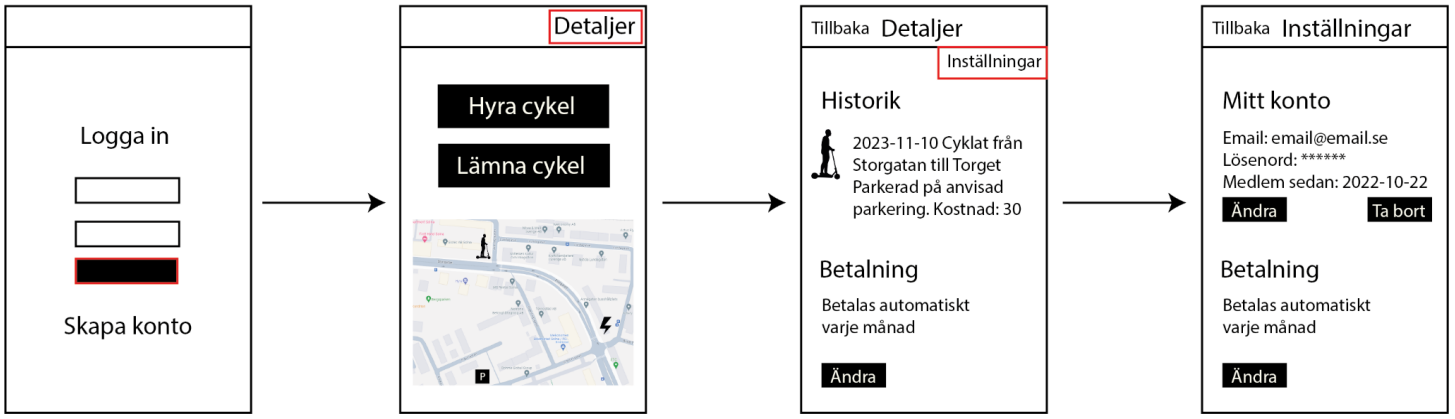


Bild 6. Frontend för kundens mobil-app.

Kundens webbgränssnitt

Webbgränssnittet för kunden kommer att göras på liknande sätt, med Javascript och React. Det kunden kommer att kunna göra via detta gränssnitt är dock mindre än för mobil-appen, då kunden här istället bara kan se sina detaljer och hyrhistorik. Kunden kan också uppdatera dessa detaljer, som betalningsinformation, ändra lösenord eller ta bort sitt konto. Även detta kopplas till REST-API:et.

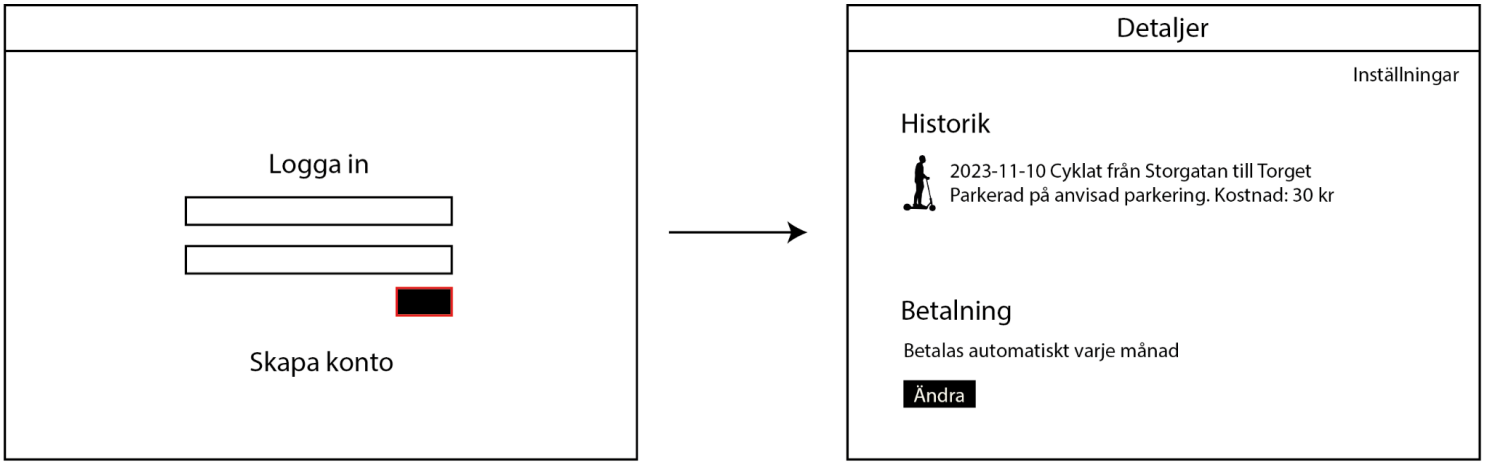


Bild 7. Frontend för kundens webbgränssnitt.

Enligt kravspecifikationen skall inloggning ske med hjälp av OAuth. Detta försöker vi uppfylla i mån av tid och kompetens, annars får en enklare lösning användas.

Admins webbgränssnitt

Även webbgränssnittet för admin kommer att göras i Javascript och React. Admin ser som i mobil-appen för kund en karta som görs i till exempel leaflet och OpenStreetMap. Från den kan admin kolla olika laddstationer, parkeringar och cyklar genom att klicka på dessa. Det går till exempel att se hur många cyklar som är placerade på en laddstation.

Admin kan hantera cyklar och kunder, till exempel begära laddning av en cykel eller hantera en kunds betalning.

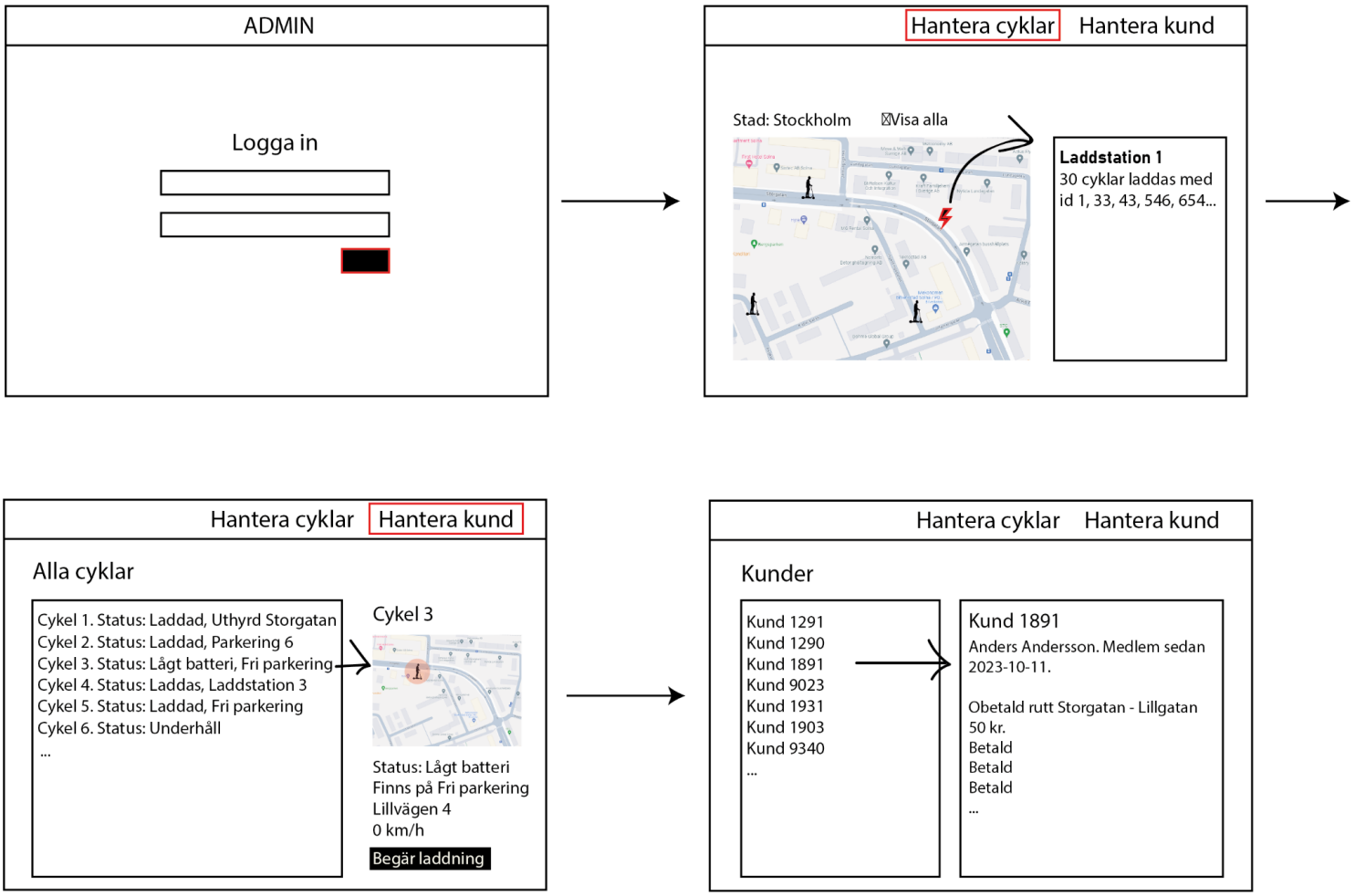


Bild 8. Frontend för admins webbgränssnitt.

Test och simulering

Enhetstester och kodvalidering automatiseras med ett CI/CD-flöde (continuous integration and delivery) med hjälp av GitHub Actions.

Systemet som en helhet simuleras med hjälp av en simulerings-app, som belastar systemet på ett realistiskt sätt genom simulera ~10^3 stycken samtidiga kunder som hyr cyklar och åker runt i städerna. En möjlighet kan vara att slumpmässigt generera start- och slutpunkter för cyklarna, och låta dessa färdas längs en rät linje mellan punkterna, och med ett lämpligt intervall rapportera sin position till servern via REST API:et. Ett annat alternativ kan vara att låta cyklarna röra sig i cirkulära banor, eller någon annan enkel geometrisk form, eller så varierar man färdsätten.

Tekniken för simuleringsappen får växa fram under utvecklingens gång, men troligen blir det en JavaScript-baserad klientapplikation med ett enkelt webb-gränssnitt.

Testmiljön kommer att använda Docker-containrar för att köra olika delar av systemet, typiskt med en container för webb- och REST-servern. Om vi använder en client-server-anpassad SQL-server, t ex MariaDB, så kan det vara lämpligt att köra den i en separat container. Om vi istället använder en filbaserad som SQLite, så ligger den i samma container som webbservern.

Klienterna kommer att köras i en webbläsare, som till exempel Chrome, Firefox eller Safari. Detta kan också gälla för simuleringsappen, om vi väljer att göra den JavaScript-baserad, med ett webbgränssnitt, annars kan man tänka sig att köra simuleringsappen baserad på Node.js utan grafiskt användargränssnitt. I så fall kunde man köra appen i en egen Docker-container.

En reflektion är att hela systemet simuleras på en fysisk utvecklardator, så systemets upplevda prestanda är beroende av datorns prestanda. Det kan vara viktigt att tänka på när vi demonstrerar systemet för slutkunden.

Appendix A. Definitioner av några begrepp

Systemdesign är en process att definiera arkitektur, moduler, gränssnitt och data för ett system för att tillfredsställa specifika krav [6]. Systemdesign kan ses som tillämpningen av systemteori på produktutveckling.

Designmetoder:

1) **Arkitekturdesign**: För att beskriva systemets vyer, modeller, beteende och struktur.

2) **Logisk design**: För att representera dataflödet, in- och utdata från systemet. Exempel: ER Diagrams.

3) **Fysisk design**: Definierat som a) Hur användare lägger till information till systemet och hur systemet representerar information tillbaka till användaren. b) Hur data modelleras och lagras i systemet. c) Hur data rör sig genom systemet, hur data valideras, säkras och/eller transformeras när de strömmar genom och ut ur systemet.

Appendix B. Exempel på fysisk modellering av en del av databasen (ej slutgiltig)

ADMIN

```
CREATE TABLE ADMIN ( EMAIL VARCHAR2(20),
PASSWORD VARCHAR2(15));
```

ACCOUNT

```
CREATE TABLE ACCOUNT ( FNAME VARCHAR2(15),
LNAME VARCHAR2(15),
EMAIL VARCHAR2(20) PRIMARY KEY,
PASSWORD VARCHAR2(15),
PHONE BIGINT(12));
```

TERMINAL

```
CREATE TABLE TERMINAL ( TERM_ID INT(5) PRIMARY KEY,
TERMINAL_NAME VARCHAR2(15),
NO_OF_BIKES INT(2));
```

BIKES

```
CREATE TABLE BIKE ( BIKE_ID INT(5) PRIMARY KEY,
BIKE_NAME VARCHAR2(10),
MODEL YEAR, COLOR VARCHAR2(10),
BIKE_TYPE VARCHAR2(8),
PRICE INT(4),
TERMINAL_ID REFERENCES TERMINAL(TERMINAL_ID) ON DELETE CASCADE,
AVAIL INT (1));
```

TRANSACTION

```
CREATE TABLE TRANSACTION (EMAIL REFERENCES USER(EMAIL) ONDELETE
ON DELETE CASCADE,
BIKE_ID REFERENCES BIKE(BIKE_ID) ON DELETE CASCADE,
START_TIME DATETIME,
END_TIME DATETIME);
```

REFERENSER

[1] **Architectural Blueprints - The “4+1” View Model of Software Architecture**,
https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf
(Besökt 2023-11-17)

[2] **4+1 architectural view model,**
https://en.wikipedia.org/wiki/4%2B1_architectural_view_model
(Besökt 2023-11-17)

[3] **Software Architecture Patterns**,
https://www.oreilly.com/content/software-architecture-patterns/
(Besökt 2023-11-18)

[4] **How to design a RESTful API architecture from a human-language spec,**
https://www.oreilly.com/content/how-to-design-a-restful-api-architecture-from-a-human-language-spec/
(Besökt 2023-11-19)

[5] **Kokbok för databasmodellering,**
https://dbwebb.se/kunskap/kokbok-for-databasmodellering
(Besökt 2023-11-19)

[6] **What is Systems design**,
https://economictimes.indiatimes.com/definition/systems-design
(Besökt 2023-11-10)