IP1 – Stationära enheter (IB906C)

Peter Borgstedt ([pebo6883@student.su.se](mailto:pebo6883@student.su.se)), 2020

# Obligatoriska uppgifter utförda (8):

1.1 Multitrådning obligatorisk (JNP: 3)1.1 Multitrådning obligatorisk (JNP: 3)

1.2 Paketering obligatorisk

1.3 Automatisk dokumentering obligatorisk

2.1.1 Stream sockets på klientsidan obligatorisk (JNP: 1-4, 8, 11)

2.1.2 Stream sockets på serversidan obligatorisk (JNP: 9, 11)

2.2.1 Datagram sockets med unicast obligatorisk (JNP: 12)

3.1.2 Databaskopplingar obligatorisk

3.2.1 Epost-sändning obligatorisk

# Frivilliga uppgifter utförda (8):

2.1.3 Stream sockets och XML frivillig (XML: 7-9)

2.1.4 Stream sockets och bildöverföring frivillig

2.2.2 Datagram sockets med multicast frivillig (JNP: 13)

3.1.1 Webbserverkopplingar frivillig (JNP: 5-7)

3.2.2 Epost-mottagning frivillig

4.1.1 Kryptering/dekryptering med symmetriska algoritmer frivillig (JS: 7, 9, 10, 13)

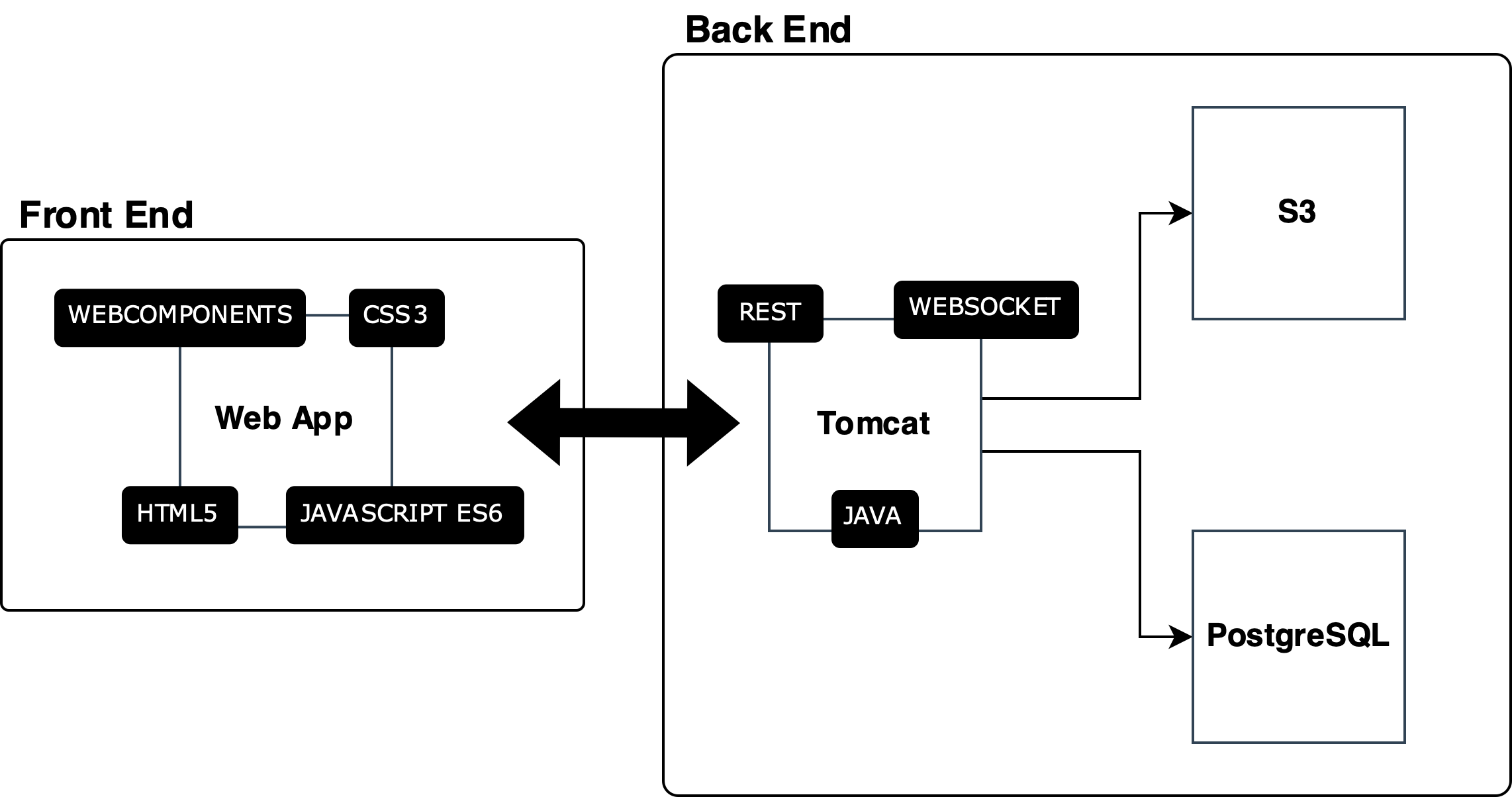
4.1.2 Signering/verifiering med asymmetriska algoritmer och nyckelpar frivillig (JS: 7, 9-12)

4.1.3 Signering/verifiering asymmetriska algoritmer och certifikat frivillig (JS: 7, 9-12)

# Gesällprov

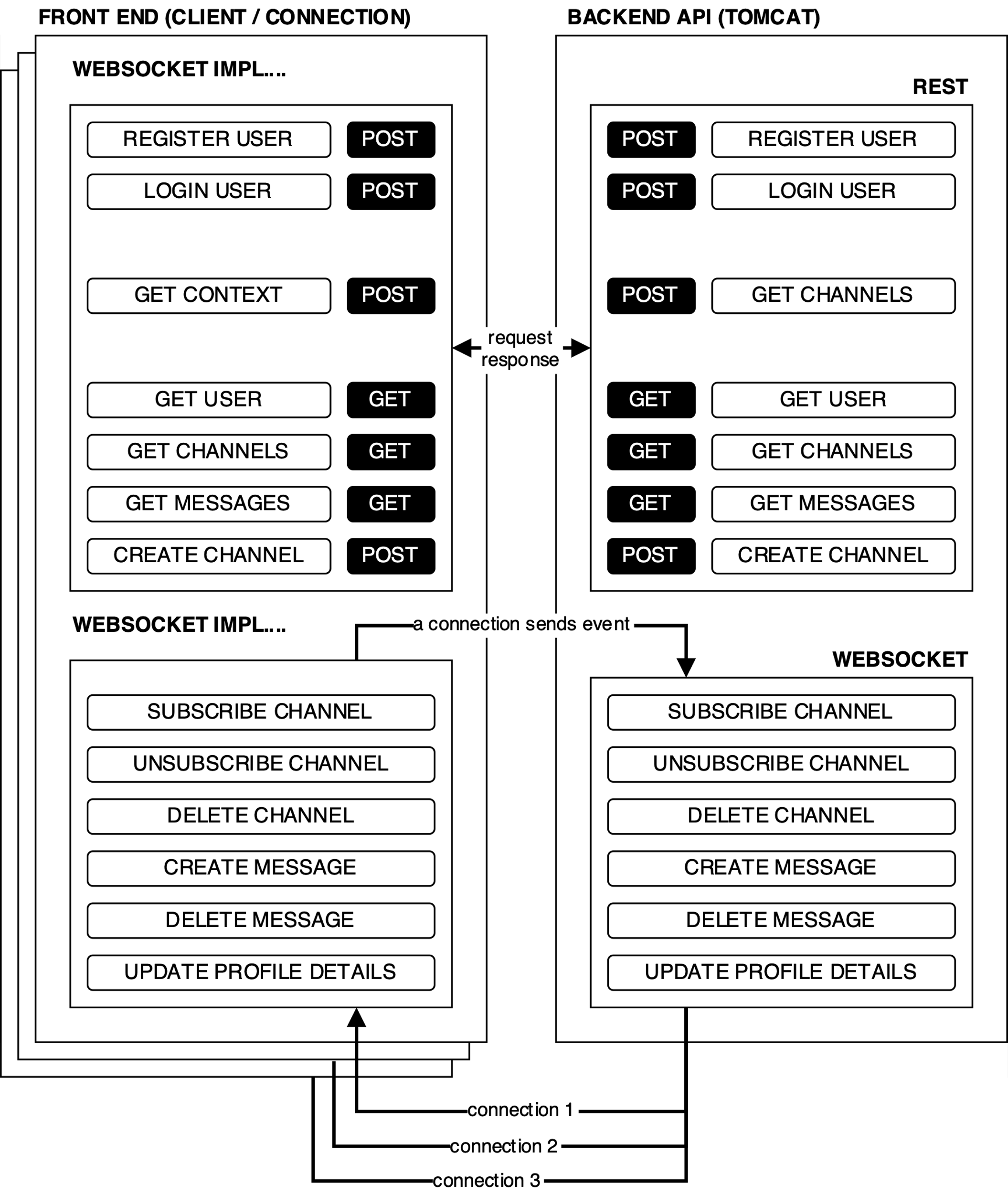
Som gesällprov har jag valt att bygga en webservice (**back-end**) för en chat med inspiration från klienter så som Teams, Slack med flera. Gesällprovet har kombinerats med kursen **IP3** (webutveckling, klientsidan) som kommer beröra implementationen för webapplikationen (**front-end**).

## Arkitektur och implementation



Webapplikationen (**front-end**) kommunicerar med webtjänsten (**back-end**) via en websocket och REST (HTTP). Webtjänsten innehåller all logik för att ta emot och sända anropen vidare till associerade anslutningar. Data kommer att hämtas och lagras i en **PostgreSQL**-databas. Anslutningar kommer skyddas via en signerad webtoken. Känsligt data kommer att krypteras; såsom användarens lösenord. Data kommer valideras vid anrop och eventuella felaktigheter kommer rapporteras tillbaka i svaren. Svar mot klienten kommer aldrig att innehålla ”stack-traces” eller känslig information om implementationen. Frågor (DQL) och manipulationer (DDL) mot databasen utförs med ”**prepared statements**” för ökad prestanda och skydd mot ”**SQL-injections**”. Nyckeln vid signering av webtoken roteras varje dygn. Data strömmas binärt och i portioner för att enklare hantera större mängder data (så som bilder). Data transformeras till rätt datastruktur efter att transaktionen är slutförd. **JSON** och **ByteArray** används som datastruktur från klienten, dessa transformeras till en tolkbar datastruktur i Java-lagret. Filerna lagras lokalt på web-servern (**Tomcat**) men laddas därefter upp mot en tredje-parts-tjänst (**AWS S3**) där filerna delas på en mer konstant lagringsplats – se förklaring nedan: Cloud Service (tredje-parts-tjänst)

## Dataflöden



Webapplikationen utför REST-anrop mot begärd resurs, efter utförandet stängs anslutningen. En anslutning mot en websocket upprättas när användaren har autentiserat sig och är ständigt uppkopplad tills användaren loggar ut.

Med REST så är anslutningen en till en; klient till server; en förfråga sker från klient och ett svar ges från server.

Med websocket så är anslutning **från klientens perspektiv** en till en, däremot från serverns perspektiv är det **fler till fler**. En klient skickar ett meddelande till servern, meddelandet tolkas, bearbetas och vidarebefordras till alla de uppkopplingar som associeras med (har ett intresse av) data-innehållet. **Servern tar emot meddelande från en eller fler anslutningar och sänder meddelande till en eller fler.**

Många av de interaktioner som sker i webapplikationer behövs vidarebefordras till andra klienter för att dessa ska få en uppdatering om händelsen. Det kan vara att ett meddelande har skapats eller tagits bort, att en användare har prenumererat eller avprenumererats på en kanal, att en kanal tagits bort eller där en användare har uppdaterat sina detaljer så som sin profilbild. När dessa händelser sker så skickas dem vidare till alla anslutningar som är intresserad av händelsen; användare som är i samma kanal. När någon uppdaterar sin profilbild så skickas det vidare och alla användare får därefter automatiskt bilden uppdaterad i sin vy utan att behöva göra något, meddelanden poppar upp och till och med kanaler försvinner. Event pushas ut när något förändrats vilket gör websocket väldigt kraftfull i en mer levande applikation som behöver information som är up-to-date.

Anrop mot REST sker endast vid mer passiva data, d.v.s. data som kan hämtas vid behov.

## Kort beskrivning av några utvalda tekniker som används

### Webservice (Java)

Tillhandahåller de grundläggande mekanismer som behövs för att distribuera och köra **Java Servlets**, **Websockets**, **REST** och mycket mer. **Apache Tomcat** valdes då den är pålitligt och väldokumenterad.

### REST (REST-ful)

REST anrop sker (via HTTP) från klient till server för ”passiv” data, d.v.s. data som kan hämtas vid behov. Med REST kan klienten bland annat hämta, posta och uppdatera data.

Anrop mot REST utförs via HTTP och kan därmed använda de metoder som protokollet stödjer; GET, POST, UPDATE, PATCH, DELETE med flera.

Det finns vissa REST-resurser som är skyddade, för dessa behövs en webtoken anges i huvudet för anropet (även kallad **request-header**), denna header måste namnges **Authorization**. För att få en token behövs ett anrop via REST med användarens inloggningsuppgifter, om uppgifterna stämmer så returneras en token.

För att en websocket kan upprättas så behövs en giltig webtoken, vilket anges som en parameter i adressen.

### Websocket

Varje klient kommer att ansluta sig mot en websocket vilket möjliggör ständigt mottagande och skickande av data; allt från text, bilder och objekt (event).

Hur länge en websocket kommer att vara vid liv är beroende på konfiguration men också huruvida ping och pong meddelanden skickas; ett ping skickas från servern för att se om klienten svarar, när klienten får meddelandet så skickas ett pong tillbaka. Denna del behövs implementeras på serversidan men däremot inte på klientsidan då stödet redan finns via HTML5 (Websocket API).

Den inre hanteringen av händelser behöver ta i hänsyn till ”concurrency”, d v.s. trådhantering. Detta är viktigt då händelser sker samtidigt från olika uppkopplingar och det kan enkelt uppkomma fel där data inte skrivs eller läses in rätt, eller ännu värre där dead-lock uppkommer.

### Webtoken

För en säker kommunikation mellan server och klient så krävs en autentisering, d.v.s. en användare behöver först registrera sig för att nyttja chat-klienten och därefter logga in. Vid en lyckad inloggning får användaren en webtoken som kan användas för att upprätta en anslutning via websocket eller anropa vissa skyddade REST-resurser. Registrering och inloggning sker via REST och dessa kräver ingen webtoken.

Notera att utöver webtoken så utförs även transaktioner via SSL/TLS både för HTTP (**https**) och websocket (**wss**).

### Databas

All data sparas i en **PostgreSQL** databas. En relationsdatabas stödjer **SQL** vilket ger möjlighet till flexibla frågor med hög prestanda, medans en **dokumentations** eller **NOSQL** databas skulle vara alldeles för långsam.

**PostgreSQL** är en väldokumenterad databas med fri licensanvändning vilket avgjorde valet.

A screenshot of a cell phone

Description automatically generated

Databas-modellen består av fyra tabeller med lösa relationer till varandra, d.v.s. inga främmande nycklar. Tabellen ”**users**” innehåller alla användare och refereras i alla ”entiteter” som associeras med användaren; de kanaler som användaren skapar, de prenumerationer på kanaler som användaren valt, de meddelanden som användaren skriver. Tabellen ”**channels**” innehåller alla kanaler. En kanal är associerad till den användaren som skapade kanalen men även mot de prenumerationer och meddelanden som sedan skapas för den. Tabellen ”**messages**” innehåller alla meddelanden och är associerad mot en användare och kanal. Tabellen ”**subscriptions**” innehåller prenumerationer mot kanaler som användare har valt.

### Cloud Service (tredje-parts-tjänst)

För att lagra filer används **AWS Simple Storage**. Det går alldeles utmärkt att dela filer via en Tomcat, vilket var min första lösning, däremot inte en optimal sådan. Filer bör lagras på en dedikerad tjänst som kan tillhandahålla alla de mekanismer som behövs; så som tillgång och säkerhet. Detta är den officiella förklaringen. Den inofficiella är att från början använde jag Heroku som värd, men eftersom tjänsten är gratis så slängdes alla filer efter 24timmar då den Tomcat som kördes gick ner i ”dvala”, i själva verket slängdes hela containern. Jag gick därefter över till AWS EC2 som kör en Tomcat på en Linux-maskin tillsammans med PostgreSQL. Dock sparade jag denna del då den inte kändes så dum eftersom filer bör sparas på en extern filserver för att minska belastning på servern, samt fick jag med en tredjeparts-integration i backenden.

## Monitorering

Med Tomat ingår loggning vilket möjliggör monitorering av fel och händelser. Jag har valt att använda mig av **Log4j2** istället för en **System.out** för att möjliggöra filtrering, d.v.s. det går att ange vilket ”nivå” man vill visa, om det är information (**info**), debugging (**debug**), varningar (**warn**) eller fel (**error**). En bra monitorering och översikt underlättar vid utveckling och felsökningar.

A screenshot from a bottle

Description automatically generated

## Säkerhet

Implementation har tagit hänsyn till ett flertal säkerhetsaspekter, vissa av dem har redan nämnds kort ovan. Den främsta är autentisering och webtoken. Denna token går att nyttja under begränsad tid. Den har ingen ”**refresh-token**” i sig (då jag inte hann implementera det), användaren behöver därmed logga in på nytt när den har löpt ut.

En token signeras med en nyckel, denna nyckel genereras när Tomcat startar en session, den kommer därefter även att **roteras** varje dygn, d.v.s. efter ett tag så kommer alla nya webtokens att signeras med en ny nyckel vilket kommer ogiltigförklara alla befintliga tokens – som leder till att användare behöver logga in på nytt; vilket kan lösas med en ”**refresh-tokens**” som då kommer att generera en ny token signerad med den nya nyckeln.

Säkerhetsåtgärder har även applicerats vid databas-transaktioner där alla mutering utförs via **prepared statements** vilket undanröjer **SQL-injektioner.**

Autentisering utförs även vid ”**handshake**” när en websocket upprättas, samt vid REST-anrop där en ”**interceptor**” validerar token före resursen anropas.

Tomcat har även konfigurerats med TLS/SSL; **HTTPS** och **WSS** (websocket).

## Summering

Det har varit ett jättekul projekt och speciellt eftersom man kunna kombinera den tillsammans med IP3, vilket gjorde att jag kunde bygga något litet extra. Det blev sommarens stora sysselsättning tillsammans med de övriga uppgifter som kursen innehöll. Tungt och stundvis omfattande men roligt och skönt nu i efterhand att se vad man har lyckas åstadkomma.

Jag har kunnat laborera med många tekniker både från kursen och utanför. Trots en del förkunskaper har jag behövt läsa på mycket vilket breddat mig en hel del. Vid kombinationsgesäll så får man lite mer verklighet på hur dessa tekniker kan användas med varandra, vilket jag tycker var motiverande och nyttigt att se.

Jag har aldrig byggt micro-tjänster i Tomcat tidigare, så det blev kul att testa på.

Det var många andra bitar som jag hade velat få på plats, bland annat datagram och strömmande av video via UDP. Jag ville också ha ”refresh-tokens” på plats men fick strippa dessa tankar för att få saker och ting klart.

Jag hoppas att mitt arbete blir uppskattat och kanske presenterar någon nytt som ingen ännu har presenterat i sin gesäll.

Övrig teknisk dokumentation finns detaljerat i koden, bl.a. hur jag resonerat och eventuella källor.