

# Backendudvikling, drift og distribuerede systemer

## GolfQuis backend

Gruppe 20

11-05-2020



Niklaes  
Jacobsen  
s160198



Kim  
Bossen  
s163290



Sebastian  
Stokkebro  
s170423

Github:

<https://github.com/kimsand123/Semesterprojekt>



Danmarks Tekniske Universitet  
62597 Backendudvikling, drift og distribuerede systemer  
F20

Antal ord: 3839  
Svarende til 10,5 normalsider

# Indhold

<b>1</b>	<b>Indledning</b>	<b>1</b>
<b>2</b>	<b>Guide</b>	<b>2</b>
2.1	Sådan kører / idriftsætter du vores projekt . . . . .	2
<b>3</b>	<b>Analyse</b>	<b>3</b>
3.1	Domæne . . . . .	3
3.2	Afgrænsning . . . . .	3
3.3	Use cases . . . . .	4
3.4	Krav . . . . .	4
<b>4</b>	<b>Design</b>	<b>5</b>
4.1	Skalering . . . . .	5
4.2	De elementer der udgør strukturen . . . . .	6
4.3	Service endpoints/ressourcer . . . . .	7
4.4	Brugergrænseflader . . . . .	7
4.4.1	Mobil Applikation - Flutter . . . . .	7
4.4.2	Web interfacet . . . . .	7
4.5	Flow Design . . . . .	8
4.5.1	Eksempel på Auth Flow . . . . .	8
4.5.2	Eksempel på Game Flow . . . . .	9
4.5.3	Eksempel på Admin Flow . . . . .	10
<b>5</b>	<b>Implementering</b>	<b>11</b>
5.1	Begrundelse for sprog . . . . .	11
5.2	Protokoller . . . . .	12
5.3	Implementeringsflow . . . . .	12
5.3.1	Login flow [Flutter App → Auth Service → Game Service → Database Service] . . . . .	12
5.3.2	Opret ny spiller flow [Web klient → Database Service] . . . . .	12
5.4	APX & Richardson's Modenhedsniveau . . . . .	13
5.5	Sikkerhed . . . . .	13
5.5.1	CORS . . . . .	13
5.5.2	Token & API nøgler . . . . .	13
<b>6</b>	<b>Test</b>	<b>14</b>
<b>7</b>	<b>Ansvarsområder</b>	<b>14</b>
<b>8</b>	<b>Konklusion</b>	<b>15</b>
8.1	Hvad er gået godt? . . . . .	15
8.2	Hvad er gået mindre godt? . . . . .	15
8.3	Ting der skulle have være anderledes / fremtidige iterationer . . . . .	15
<b>9</b>	<b>Bilag</b>	<b>17</b>
<b>A</b>	<b>Deployment Diagram</b>	<b>17</b>
<b>B</b>	<b>Frit billede</b>	<b>18</b>
<b>C</b>	<b>System Diagram</b>	<b>19</b>
<b>D</b>	<b>Brugergrænseflade Mobil Applikation - Flutter</b>	<b>20</b>
<b>E</b>	<b>Brugergrænseflade Web interfacet - VueJS</b>	<b>24</b>
<b>F</b>	<b>Implementering af Autorisations flow</b>	<b>26</b>
<b>G</b>	<b>Implementering af Opret ny spiller flow</b>	<b>31</b>

## 1 Indledning

Følgende projekt omhandler udviklingen af et backend-system til mobilapplikationen *GolfQuis*. *GolfQuis* er en allerede eksisterende applikation, som er tilgængelig på både Apple App Store og Google Play Store. For tiden er et større redesign af applikationen i gang, som udvikles med cross-platform frameworkt Flutter. Dette projekt har derfor til formål, at udvikle en backend til den nye redesignede applikation, så man kan spille mod sine venner og se sine profil informationer.

Backenden udvikles i Python med REST Frameworket Django, som skal bestå af en Authentication Service, Game Service og en Database Service. Udover det udvikles der klienter i hhv. Flutter og VueJS.

Formålet er at få projektet op og køre som et distribueret system, fordelt udeover tre forskellige servere.

### De kørende services:

- AuthService
  - (REST service): <http://87.61.85.141:9800/>
- GameService
  - (REST service): <http://94.130.183.32:9700/>
- DatabaseService
  - (REST service): <http://116.203.234.111:9600/>
  - \* Eller på: <https://api.dinodev.dk/>
- Administrator site
  - (VueJS App): <https://gq.stokkebro.dev/>
  - \* Bruger: "admin", Adgangskode: "password"
- Mobil applikation
  - (Appetizer): <https://shorturl.at/abHR8>
    - \* Bruger: dtu brugernavn, Adgangskode: adgangskode til javabog.dk
  - (Diawi APK): <https://webapp.diawi.com/install/MaKJ3L>
  - (Diawi IPA): <https://webapp.diawi.com/install/LVUrVA>
    - \* Kan kun installeres hvis du har adgang til DTUs Apple certifikat

## 2 Guide

### 2.1 Sådan kører / idriftsætter du vores projekt

Projektet er sat op med Docker, så det er nemt og hurtigt at starte de forskellige services. Der er en række forskellige måder at køre projektet på, to af disse er listet herunder.

#### Kør det remote

Den nemmeste måde at køre projektet på er blot, at bruge de distribuerede services med de klienter der er sat til rådighed.

- Flutter applikation (Kun Android)

Installér APK'en på en fysisk enhed eller en emulator

APK'en kan findes i 'Clients/APK' hvor der både er en debug og en release APK

Appen kan også køres på appetize.io her.

- VueJS Admin applikationen

Gå ind på følgende link <https://gq.stokkebro.dev/>

\* Bruger: "admin", Adgangskode: "password"

#### Kør det lokalt

En anden metode er at køre alle services lokalt med Docker.

##### Servere:

- Servers med Docker

Kør docker-compose.yml filen i 'Servers/' mappen.

Brug følgende kommando til at starte alle services.

`docker-compose up --build`

- AuthService kører på 127.0.0.1:9800

- GameService kører på 127.0.0.1:9700

- DatabaseServices kører på 127.0.0.1:9600

##### Klienter:

- Flutter applikation (Kun iOS)

Sæt testBackend til true i /Clients/MobileApp/lib/network/service\_constants.

Kør projektet med Flutter<sup>1</sup>. (Dette kan desværre kun køre på iOS)

- VueJS Admin applikationen

Åben projektet og kør kommandoen `npm install`<sup>2</sup>

Ændr api adresserne til de lokale adresser med de korrekte porte.

Kør derefter projektet med `npm run dev`

- Lille Python terminal klient:

Her kan du ved at skrive dine login kriterier, brugernavn og password, se den datapakke du får tilbage til klienten, som den bruger til at kommunikere med Game Service med.

Via terminalen gå til <projektet>/Clients

kør programmet Console.py ved at skrive

– pip install requirements

– python Console.py

<sup>1</sup>Installer flutter her: <https://flutter.dev/docs/get-started/install>

<sup>2</sup>Sørg for at have VueJS installeret <https://v1.vuejs.org/guide/installation.html>

### 3 Analyse

#### 3.1 Domæne

GolfQuis er et eksisterende spil, som findes til både Android og iOS.

Der er rigtigt mange mennesker der har udfordringer med reglerne i golf.

Ideen med GolfQuis er at man kan træne sig selv i golfregler, men som et spil i stedet.

Golfreglerne bliver udformet som spil-spørgsmål, som gør det sjovt at lære flere regler om golf.

I løbet af et spil svarer man på 18 spørgsmål, og får henholdsvis 2 point for korrekt, og 0 point for et forkert svar. Udover spillet, er der også mulighed for at spille, og se hvordan du klarer dig ift. dine venner, dine grupper og din klub. Der er forskellige spiltyper man kan spille:

- Single player match
- Two player match
- Group match
- Tournament

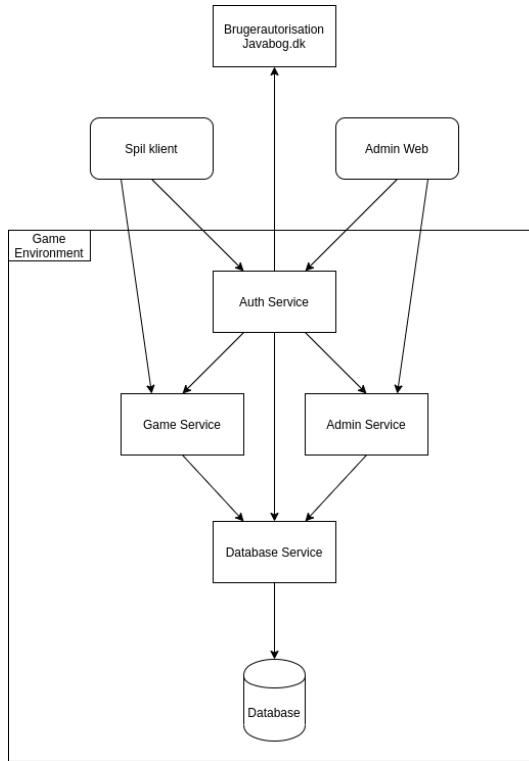
Turnering er sat op af GolfQuis' ejer Appicorn, og sættes op ift. arrangementer henover f.eks. weekender.

#### 3.2 Afgrænsning

Vi har valgt at afgrænse domænet til et system hvor man kan:

- Logge ind og blive verificeret via javabog.dk
- Spille 2 spiller spil
- Vælge at invitere en modstander blandt alle der er registreret som spillere i databasen.
- Vælge hvor lang tid der er til hvert spørgsmål.
- Spille et spil man er inviteret til.
- Sætte et spil på pause og gå videre senere
- Man kan se sin egen og sine venners highscore.

Det skal også siges, at vi har valgt mere trivia spørgsmål i dette projekt, end de golfregel-relatede spørgsmål der er normalt.



Figur 1: Frit billede af GolfQuis (Større version i bilag B)

### 3.3 Use cases

- Spiller usecase. Spiller skal ved at logge ind på en mobilapplikation kunne invitere en ven til et spil **GolfQuis**. Når vennen har accepteret skal man kunne spille et spil. Man skal kunne se sin egen og sine venners highscore
- Admin usecase. Administrator skal, via et webinterface, kunne logge ind på systemet, og vedligeholde alle felterne i databasen.

### 3.4 Krav

Kravlisten er specificeret og efterfølgende prioriteret ved hjælp af MoSCoW metoden.

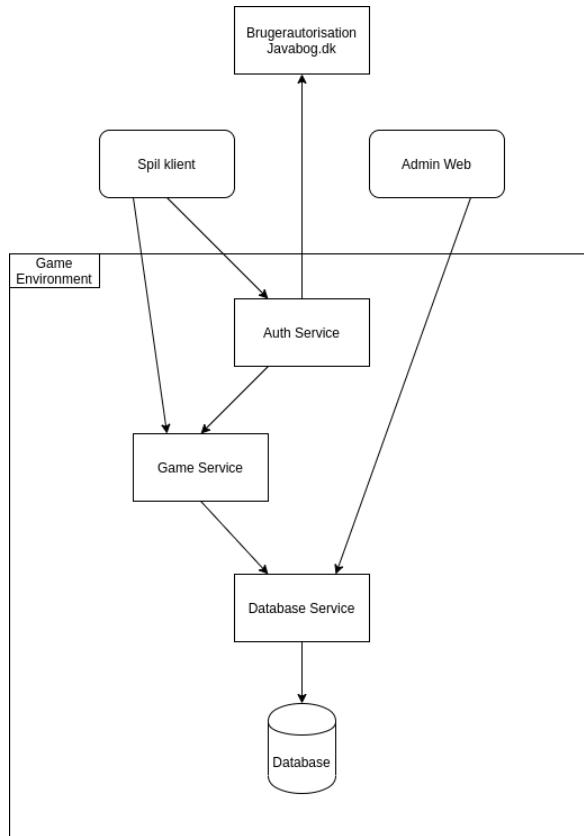
- Must have
  - Mobil app klient, web klient
  - Services kodes i Python
  - Autentificering hos javabog.dk
  - Services skal distribueres via docker på 3 geografisk separerede servere
  - Der skal hentes, sendes og ændres data med kald over nettet
  - Der skal persisteres klient data hos en eller flere servere
- Should have
  - Brug af forskellige programmeringssprog
  - Logging
- Could have
  - Kryptering af kommunikation
  - Hashing & salting af password
  - JWS (Json Web Token)

## 4 Design

Vores distribuerede system design skal bygges op med 3 services, som skal være placeret geografisk adskilt, og kunne kommunikere med hinanden over internet.

De 3 services skal være nemme og hurtige at starte og stoppe.

Vi har valgt tre i stedet for fire services som vi kan se i det tidligere frie billede, fordi vi har ladet Admin Web gå direkte ned i Database Servicen, uden om Auth Service for ikke at skulle lave 2 forskellige auth processer(prioritering af udviklings tid), og fordi den funktionalitet som Admin Web skal have er meget tæt koblet til databasen(struktur).



Figur 2: System Diagram (Større version i bilag C)

### 4.1 Skalering

Vi har i processen talt en del om skalering, og det er selvfølgelig en retorisk øvelse da vi ikke har tid og mulighed for at lave en reel skalerings øvelse. Derfor er dette en beskrivelse af en diskussion.

Vi har været optaget af at forsøge at undgå bottlenecks. Det viser sig at være svært da der på et eller andet tidspunkt altid kommer et single point of entry, og det har for os så være en opgave at definere et point of entry hvor der er så lidt trafik som muligt.

Vores single point of entry for spil klienten, er blevet til Auth Service. Dette er fordi det er den proces som kræver mindst kommunikation mellem spil klient og vores services.

Vi har skabt Auth processen således at Auth Service returnerer ip adresse og port for den Game Service som klienten skal spille på, til klienten således at denne forbinder direkte til en Game Service når den spiller, og ikke som i vores halvvejsprojekt hvor klienten på forhånd vidste hvem den skulle kommunikere med for at spille, og alt kommunikation gik igennem den service(single point of entry). Dette er gjort for at forholde sig til en skaleringsproces, hvor vi dermed eliminerer et single point of entry i selve spil processen, som vil være den proces der kræver mest trafik.

Det er også gjort for, at når klienten får ip adressen og port, så har et Ressource handler system taget beslutning omkring hvilken ressource klienten skal spille på. Dvs skaleringsprocessen som indebærer vurdering af ressource behov og allokering, er kørt til ende.

Hver service kunne tænkes at være på sin egen cluster, med en reverse proxy og en Ressource handler foran.

- Ressource handler, er noget proprietært software som via telemetri fra servicen, hele tiden vurderer om der skal spindes flere docker containere op/ned for at imødekomme det aktuelle ressource behov, og derefter udfører dette ved at spinde containere op med nye ip-adresser og/eller porte. Vurderings punkter kan være processor, ram, hd og traffik belastning, svartider, antal fejl.
- Reverse proxy, ex. NginX. som ligger og fordeler arbejdsbyrden til de etablerede containere alt efter en vedtagen algoritme, ex. Round Robin, Least Work etc.

## 4.2 De elementer der udgør strukturen

Vi laver 3 services og 2 klienter.

- **Auth Service**, som skal autentificere brugeren overfor javabog.dk, samt udstede token til Game Service og klienten.
- **Game Service**, som skal håndtere og formidle spillets spillere, invitationer og spil data ved hjælp af Database Service, og spiller rettigheder via token overfor Mobil app klient.
- **Database Service**, som skal persistere spiller, invitationer og spil, og leve til Game Service samt Web Admin klient.
- **Mobil app klient**, som spilleren bruger til at spille spillet. Her kan man:
  - Logge ind/ud
  - Se sin profil
  - Se andre spilleres profil
  - Invitere en spiller til et spil
  - Gå ind i et spil og spille
  - Pause et spil
- **Web admin klient**, som web admin bruger til at vedligeholde felterne i databasen. Her kan man:
  - Logge ind/ud
  - Fuld CRUD på alle ressourcerne i databasen.

## 4.3 Service endpoints/ressourcer

Vi har identificeret forskellige ressourcer i de forskellige services, disse er beskrevet herunder, og mere indgående i dette dokument<sup>3</sup>. Det skal siges dokumentet er ikke fuldt opdateret da det gik stærkt til sidst.

- **Auth Service**

- login/ [POST]

- **Game Service**

- players/ [GET, POST]
- players/<player\_id> [GET, PUT, DELETE]
- invites/ [GET, POST]
- invites/<invite\_id> [GET, PUT, DELETE]
- games/ [GET, POST]
- games/<game\_id>/player-status/ [PUT]
- games/<game\_id>/ [GET, PUT, DELETE]
- register\_user/ [POST]

- **Database Service**

- players/ [GET, POST]
- players/<player\_id>/ [GET, PUT, DELETE]
- invites/ [GET, POST]
- invites/<invite\_id>/ [GET, PUT, DELETE]
- games/ [GET, POST]
- games/<game\_id>/player-status [PUT]
- games/<game\_id> [GET, PUT, DELETE]
- questions/ [GET, POST]
- questions/<question\_it> [GET, PUT, DELETE]

## 4.4 Brugergrænseflader

### 4.4.1 Mobil Applikation - Flutter

Den mobile applikation er en multiplayer client, der bruger de opsatte services.

Klienten er skrevet i Flutter, og bruger rest-interface til kommunikation. Flutter er et cross-platform værktøj, og derfor kan appen køres både på iOS og Android. Som tidligere nævnt, er Sebastian og Niklaes fra gruppen i gang med at lave en ny version af den eksisterende GolfQuis App. Flutter koden er baseret på det nye design, men udstilles her, som en skrabet version med kun *two-player matches, profil* og *venner*. Derudover har man mulighed for at logge ind, men kan hverken nulstille sit password eller oprette sig som ny bruger. En fuld beskrivelse af dette ses i *bilag D*.

### 4.4.2 Web interfacet

Udstiller en administratorside, til administration af brugere, spil, spørgsmål og invitationer. Giver mulighed for CRUD direkte til databasen. Klienten er udviklet i VueJS og udstiller et administrations site. Her kan administratoren logge ind for at aflæse, oprette og modificere data direkte i databasen.

En fuld beskrivelse af dette ses i *bilag E*.

---

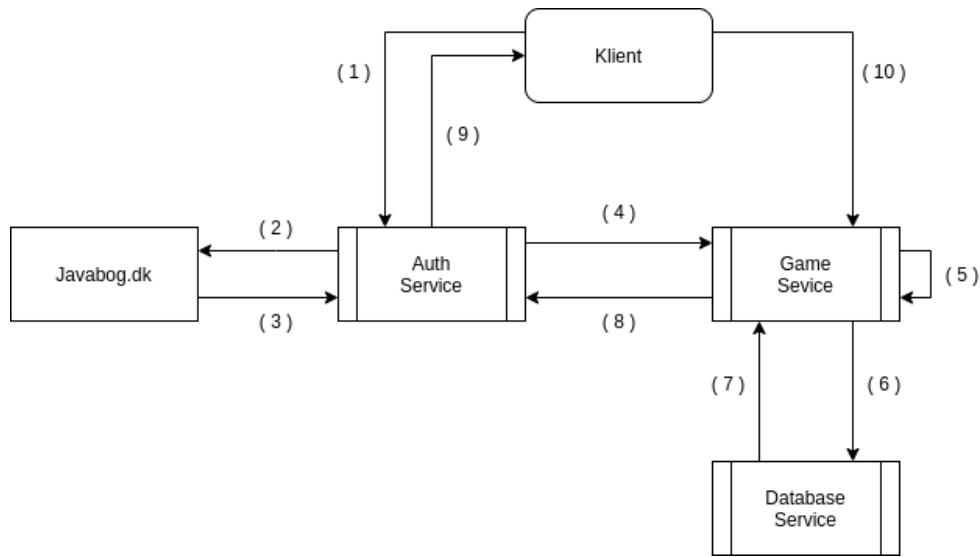
<sup>3</sup>[https://docs.google.com/document/d/1K\\_4KhTPDI-fPMkTB\\_5Xc3dukP4T34633zQ41Dphs\\_rs/edit?usp=sharing](https://docs.google.com/document/d/1K_4KhTPDI-fPMkTB_5Xc3dukP4T34633zQ41Dphs_rs/edit?usp=sharing)

## 4.5 Flow Design

### 4.5.1 Eksempel på Auth Flow

Her ses Flowet til autentifikation af en bruger.

**SK=Spil Klient, AS=Auth Service, GS=Game Service, DS=Database Service**

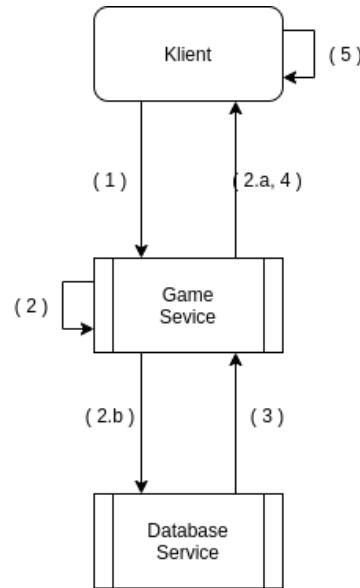


Figur 3: Auth Flow

1. **SK** logger ind med brugernavn og password
2. **AS** verificerer bruger login hos **Javabog.dk**
3. **Javabog.dk** returnerer et bruger\_objekt hvis brugeren fandtes
4. Hvis brugeren autentificeres, sender **AS** auth\_service\_access\_token, spiller\_objekt og spiller\_token til **GS**.
5. **GS** verificerer Auth\_service\_access\_token, og gemmer spiller\_token i en liste.
6. **GS** checker om bruger allerede eksisterer, ellers oprettes bruger ved hjælp af **DS**. Ved kommunikation mellem **DS** og **GS** bruges der en security\_token til at verificere **GS** overfor **DS**
7. **DS** returnerer et spiller\_objekt
8. **GS** returnerer et spiller\_objekt, **GS** ip adresse og port til **AS**
9. **AS** returnerer et spiller\_objekt, **GS** ip adresse og port, samt spiller\_token til **SK**.
10. **SK** bruger information modtaget af **AS** til at tilgå **GS**

#### 4.5.2 Eksempel på Game Flow

Her ses flowet for at vise andre spillere på en liste i mobil applikationen.  
**SK=Spil Klient, GS=Game Service, DS=Database Service**



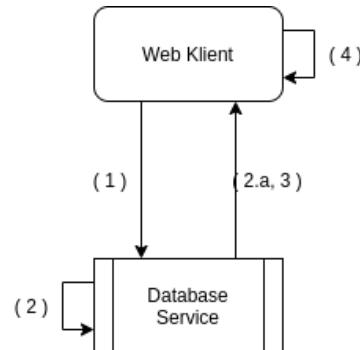
Figur 4: Hent spiller liste flow

1. **SK** beder **GS** om at få listen af spillere, authentificerer sig med spiller\_token
2. **GS** checker om spiller\_token er tilstede i listen, og at den ikke er for gammel.
  - a) Hvis nej, så send fejl tilbage til klient
  - b) Hvis ja, så refresher **GS** spiller\_token og beder **GS**, **DS** om listen af spillere, **GS** autentificerer sig overfor **DS**  
med en security\_token
3. **DS** returnerer listen af spillere til **GS**
4. **GS** returnerer listen af spillere til **SK**.
5. **SK** viser listen til brugeren

#### 4.5.3 Eksempel på Admin Flow

Her ses Flowet til en CRUD handling fra Web Admin.

**AWK=Admin Web Client, DS=Database Service**

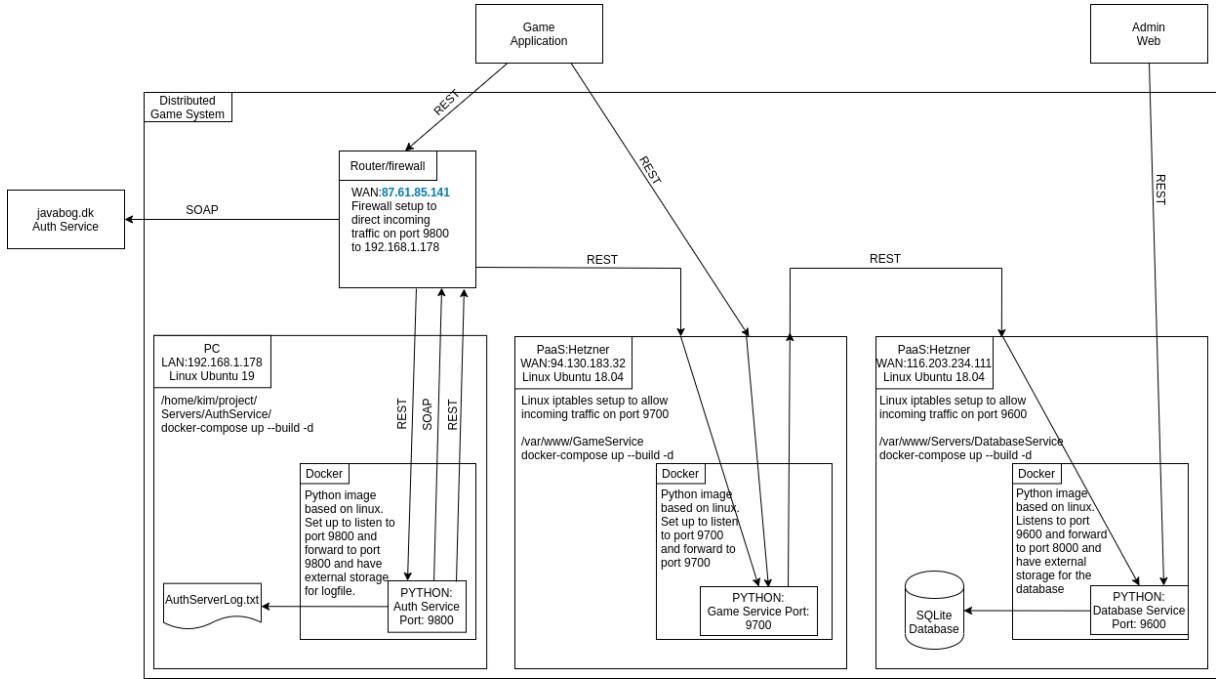


Figur 5: Auth Flow

1. **AWK** beder **DS** om at slette en spiller, ved at bruge endpointet i DS som modtager player/spiller\_id i URL, og medsender en api\_token.
2. **DS** checker om api\_token er korrekt.
  - a) Hvis nej, så send fejl tilbage til klient
  - b) Hvis ja, slet spiller i databasen
3. **DS** returnerer status kode til **AWK**
4. **AWK** opdaterer liste på baggrund af tilbagemelding fra **DS**.

## 5 Implementering

Deployment diagrammet kan ses på en fuld side i bilag A.



Figur 6: Deployment diagram

### 5.1 Begrundelse for sprog

Vi har valgt at skrive i tre sprog i følgende dele af projektet. Det har vi gjort da hvert sprog har sin fordel indenfor for deres respektive område.

- **Python**

- Servere

Vi valgte Python da vi gerne ville bruge Django REST Frameworket, til at håndtere vores endpoints og database. Django opretter en SQLite baseret database ud fra de modeller, som defineres i Database Service.

- **Flutter/Dart**

- Mobil Applikation klient

Fordi Flutter er et crossplatform udviklingsværktøj, kan vi nu køre vores mobil applikation på både IOS og Android. Flutter er skrevet i sproget Dart.

- **VueJS**

- Web Applikation klient

Vi valgte Vue til administrator web klienten, da Vue er et glimrende valg til små og simple komponent-baserede web applikationer.

## 5.2 Protokoller

Vi bruger HTTP/REST endpoints til kommunikation mellem de forskellige servere og klienter. SOAP bruges mellem Authentication servicen og javabog.dk's brugerautorisationsmodul. De forskellige protokoller bruges på følgende måde:

### SOAP(RPC)

Giver os muligheden for at kommunikere med javabog.dk's brugerautorisationsmodul, så man kan logge ind med sin studie identifikation. Brugen af SOAP i dette projekt er minimal da dette kun bruges til Authentication.

- Auth Service ↔ Javabog.dk

### HTTP/REST

Giver os muligheden for at fokusere mere på ressourcer.

I dette projekt består størstedelen af serversiden af HTTP/REST kommunikation.

- Mobil klient ↔ Auth Service/Game Service
- Auth Service ↔ Game Service
- Game Service ↔ Database Service
- Web klient ↔ Database Service

## 5.3 Implementeringsflow

### 5.3.1 Login flow [Flutter App → Auth Service → Game Service → Database Service]

Dette flow er processen der sker når man logger ind.

Login flowet fra Flutter App til DatabaseService

- Flutter App forbinder til Auth Service
- Auth Service autentificerer med `javabog.dk`
- Auth Service forbinder til Game Service
  - Her sendes brugerinformationer, en genereret token og en API token
- Game Service forbinder til Database Service
  - Her oprettes eller hentes brugeren fra databasen
- Database Service returnerer brugerobjekt til Game Service
- Game Service returnerer data til Auth Service
  - Herunder egen ip og port, samt svar fra Database Service
- Auth Service returnerer data til Flutter app
  - Svaret fra Game Service

Det fulde implementerede flow med kode kan ses i *bilag F*.

### 5.3.2 Opret ny spiller flow [Web klient → Database Service]

Dette flow er processen der sker når man opretter en bruger fra administrator sitet.

- Web klient forbinder til DatabaseService.
- Der sendes et post request med en bruger som JSON-objekt.
- Database service åbner op for databasen og tilføjer brugerobjektet.
- Database service returnerer et JSON respons med statuskode og data.
- Brugeren oprettes hvis Database service returnerer statuskode 201.

Det fulde implementerede flow med kode kan ses i *bilag G*.

## 5.4 APX & Richardson's Modenhedsniveau

Der er lagt stor fokus på at have et højt APX niveau gennem hele projektet. Dette har været vigtigt for brugen af API'en, både for andre udviklere, men også for den interne udvikling af produktet. Oplever man et problem med API'en, har det været en høj prioritet, at man får et meningsfuldt respons tilbage med den korrekte HTTP fejlkode. Med fejlkoden gives en begrundelse på hvorfor problemet opstod, samt hjælp til hvordan man kan løse problemet.

I forhold til Richardson's Modenhedsmodel, når vores API et niveau på to.

API'en opfylder ikke HATEOAS princippet, men benytter dog flere forskellige URI'er til forskellige ressourcer, samt gør brug af de forskellige HTTP verber. Her er et eksempel på et HTTP respons, hvis man forsøger at tilgå en forkert sti.

---

```

1  {
2      "requested-url": "[GET] /players",
3      "error": "You have requested a wrong path.",
4      "players endpoint": "/players/",
5      "games endpoint": "/games/",
6      "invites endpoint": "/invites/",
7      "questions endpoint": "/questions/"
8 }
```

---

Som det kan ses ovenfor, gives der et respons med de tilgængelige endpoints, så man nemt og hurtigt kan spore fejlen og tilgå det rigtige endpoint.

## 5.5 Sikkerhed

Projektet indeholder ikke de mest avanceret sikkerhedsforanstaltninger, der er dog tænkt over hvordan sikkerhedsniveauet kan hæves en smule. Herunder er beskrevet to sikkerhedsudfordringer vi er stødt på.

### 5.5.1 CORS

Ved udviklingen af web applikationen stødte vi på sikkerhedspolitikken `same-origin policy`, som forhindrer cross-site request forgery hvor en hacker kan udnytte session cookies til at autentificerer sig på andres profiler. Der er flere forskellige måder at løse CORS fejlen, vi har valgt at tillade CORS fra to bestemte domæner på serverens side. Til dette bruges følgende linje.

---

```

1 CORS_ORIGIN_WHITELIST = [
2     'https://gq.stokkebro.dev',
3     'http://localhost:8080'
4 ]
```

---

Det ene domæne der skal tillades er selvfølgelig `https://gq.stokkebro.dev`, da dette er det kørende distribuerede system. Den anden url der skal tillades er `http://localhost:8080` til udviklingen af sitet, når det køres på en lokal server.

### 5.5.2 Token & API nøgler

Vi bruger tokens/API nøgler til at håndtere autentificering og autorisation i projektet.

Hvis en bruger bliver verificeret hos javabog.dk, laver vi en player\_token som er en tilfældigt genereret streng, som bliver returneret til klienten og sendt til Game Service som gemmer denne i en liste. Hver gang en klient forsøger at lave et restkald, skal denne player\_token være med i autorisations headeren, ellers vil metoden ikke udføre sin kode, og returnerer http fejlkode 401 Unauthorized. Denne player\_token vedligeholdes af en metode, som i et kald forholder sig til om player\_token eksisterer i Game Service og som derfor må bruge de udstillede endpoints, om den er udløbet(mere end 30 minutter gammel), og hvis den ikke er for gammel, så opfriskes token sådan den får et nyt tidsstempel og er valid i endnu 30 minutter. Ellers sendes en http fejlkode 401 Unauthorized tilbage

De eneste endpoints hvor man ikke skal bruge player\_tokens er

1. Login hos Auth Service, for det skal alle kunne tilgå.
2. Register\_user hos Game Service, for det er et endpoint der ikke bruges af klienten, men af Auth Service
3. Alle endpoints hos Database Service da det kun er andre services eller admin modulet der tilgår dem.

For at undgå at en person udenfor systemet, kan bruge de endpoints der beskrives i punkt 2-3, har vi introduceret API nøgler. De er det samme som en token, men det er hemmelige nøgler som kun vores system kender, og som ikke afsløres i kommunikationen med spil klienten. Vi har ikke lavet sikkerhed omkring kommunikation mellem Admin Web og Database Service.

For punkt 2 bliver nøglen overført som en del af body, og for punkt 3 bliver nøglen overført som en Authorization header. Den eneste grund til forskellen, er at vi ikke har fået rettet en gammel metode (Punkt 2) til at bruge den rigtige metode (Punkt 3) før vi frøs projektudviklingen.

## 6 Test

Vi har brugertestet systemet via applikationerne som en blackbox, hvor vi har kørt de forskellige applikationer og set om de opfører sig som de skal.

Derudover har vi brugt web admin applikationen til at kigge på data i databasen for at se de ændrede sig som vi forventede de skulle gøre.

Vi har skrevet en test af active\_player\_list i Game Service, som undersøger om metoderne i den klasse virker som de skal. Desværre kunne vi ikke nemt finde ud af hvordan vi lavede Unit tests i Python, derfor blev det en klasse som vi kaldte når man kaldte login endpointet, som testede active\_player\_list.

## 7 Ansvarsområder

Navn / Produkt	Kim (s163290)	Niklaes (s160198)	Sebastian (s170423)
Auth Service	100%		
Game Service	50%	10%	40%
Database Service		100%	
Flutter App		75%	25%
Web applikation			100%
Containerization		100%	

## 8 Konklusion

Vi kan konkludere, at vi har udviklet et distribueret system, der opfylder de prioriterede krav. Et distribueret system der kommunikerer på tværs af geografisk placering, på tværs af enheder, programmeringssprog og protokoller. Systemet er spundet op i docker containere således at idriftsættelse er hurtigt, enkelt og nemt tilgængeligt, og som opfylder behovet i den afgrænsede opgave. Det distribuerede system kan understøtte en autorisations, spil og vedligeholdelses proces som bliver udført på de udviklede klient applikationer. Vi har lavet et proof of concept, med essentielle komponenter og strukturer, som man kan arbejde videre på i forhold til at videreudvikle applikationen/systemet til marked.

### 8.1 Hvad er gået godt?

Vores APX er ret god, vi synes vi giver en fin feedback til programmøren/brugeren af vores API.

Vi mener at vores struktur er ret god. Der er nogle enkelte ting som vi også beskriver nedenunder, men i det store hele er vi ret glade for den måde vi har struktureret det på også i forhold til en skaleringsdiskussion.

Vores Admin website giver et godt overblik, og er en nem måde at håndtere database entiteten på.

Vores idriftsættelsesmetode er rigtig god, der er blevet skrevet nogle rigtig gode docker-compose.yml og docker filer som gør det at starte en Service til et spørgsmål om bare at skrive `sudo docker-compose up --build -d`. Nemmere kan det vist ikke være.

Vores samarbejde inden Corona og vores evne til at komme godt rundt omkring emnet, har været virkelig god. Vi har haft en masse gode diskussioner inden vi blev smidt i karantæne.

Vi har fået lavet en god prototype som man kan arbejde videre på i forhold til afsætning til marked.

### 8.2 Hvad er gået mindre godt?

Det har været svært at samarbejde udelukkende over discord. En kreativ proces og vidensdeling er på ingen måde så umiddelbar og tilgængelig som hvis man sidder sammen. Kommunikationen er meget anderledes når man kan se hinanden, i forhold til når man ikke kan. Det er gået lidt ud over den røde tråd, og vi har brugt uforholdsmaessigt meget tid på at få tingene til at hænge sammen, selvom vi har brugt en god del tid på at definere tingene først.

### 8.3 Ting der skulle have være anderledes / fremtidige iterationer

I vores design skulle vi have undgået at involvere Game Service i håndteringen af en bruger i login processen. Det skulle have været Auth Service som talte direkte med Database Service omkring om en bruger eksisterede eller ej, og den deraffølgende oprettelse. I vores umiddelbare situation betyder det ikke noget, men i forhold til skalering i et system med hundredetusindvis af brugere, så betyder login processen ret meget i forhold til trafik og computeringsressourcer.

Vi ville gerne have været endnu mere konsistent med feedback (APX) således vi havde en ensartet exception håndtering og feedback til brugeren og programmøren. Dette har vi dog været udfordret med at koordinere pga. Corona krise, hvor vi ikke har kunnet mødes for at designe og brainstorme sammen.

Vores player\_token burde have været en JWT. Det blev det ikke fordi vi havde implementeret det meste af logikken omkring dette inden vi havde haft forelæsningen, og vi prioriterede derfor ikke at lave det om. Vores player\_token er en UUID automatisk genereret token, som bliver genereret på baggrund af bl.a. det tidspunkt som den bliver lavet på. Dette gør den forudsigelig, og man ville formentligt kunne gætte sig til en ny token ved at studere hvorledes UUID fungerer og begynde at gætte sig frem. Dette er vi opmærksomme på, men som proof of concept mener vi vores løsning er udemærket. En JWT ville selvfølgeligt have været en bedre løsning da der i én token er indeholdt alt den information som man skal bruge, og den er beskyttet af en meget stærk algoritme. Vi ville stadig have lavet vores token hjælperutine

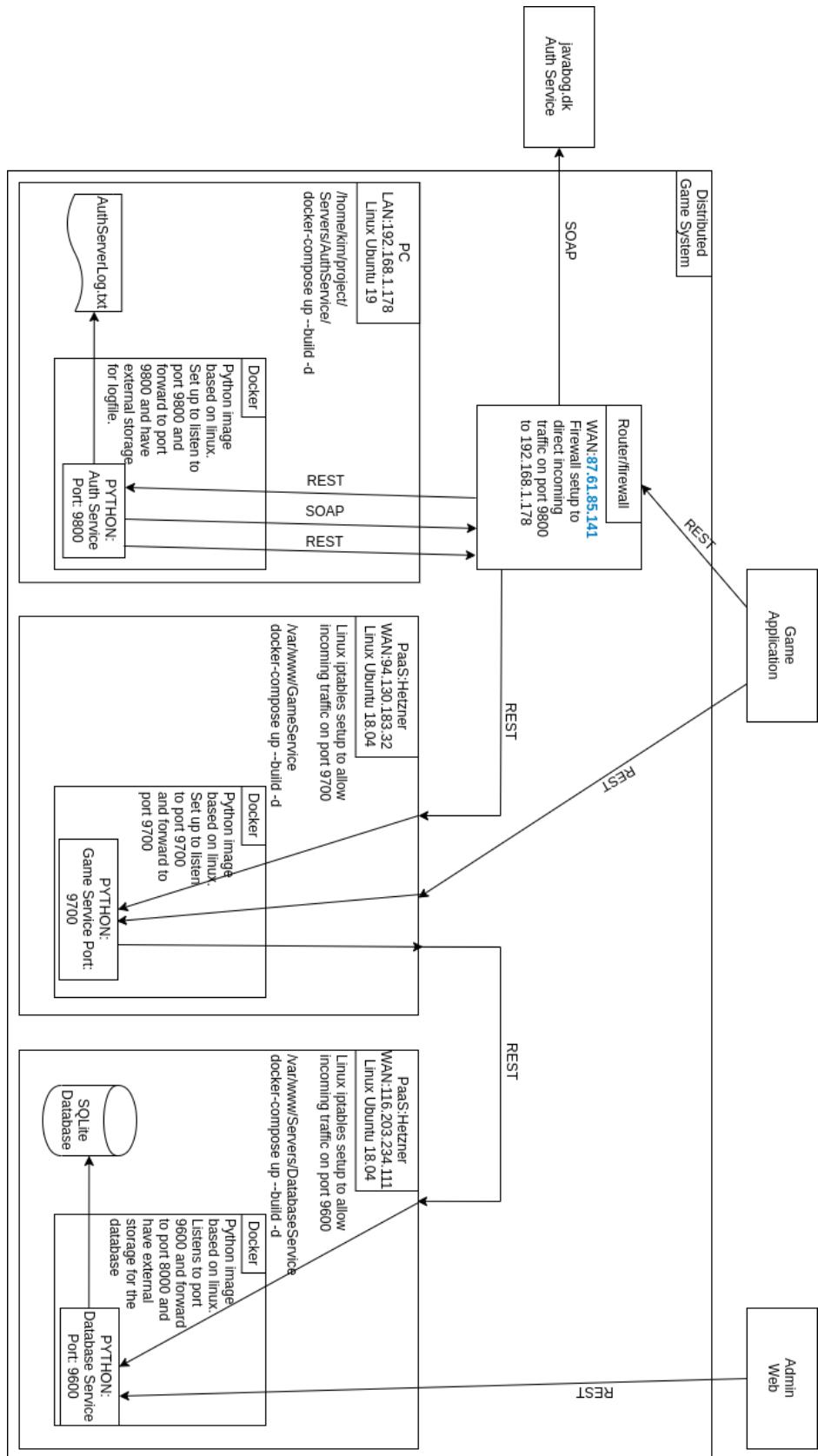
nogenlunde som den allerede er lavet.

Vi følger ikke standarden for autorisations header nemlig at den skal postfixes med "Bearer ", fordi vi først ret sent kom i gang med at implementere den rigtige måde at håndtere token på. Derfor blev det ikke gjort helt korrekt.

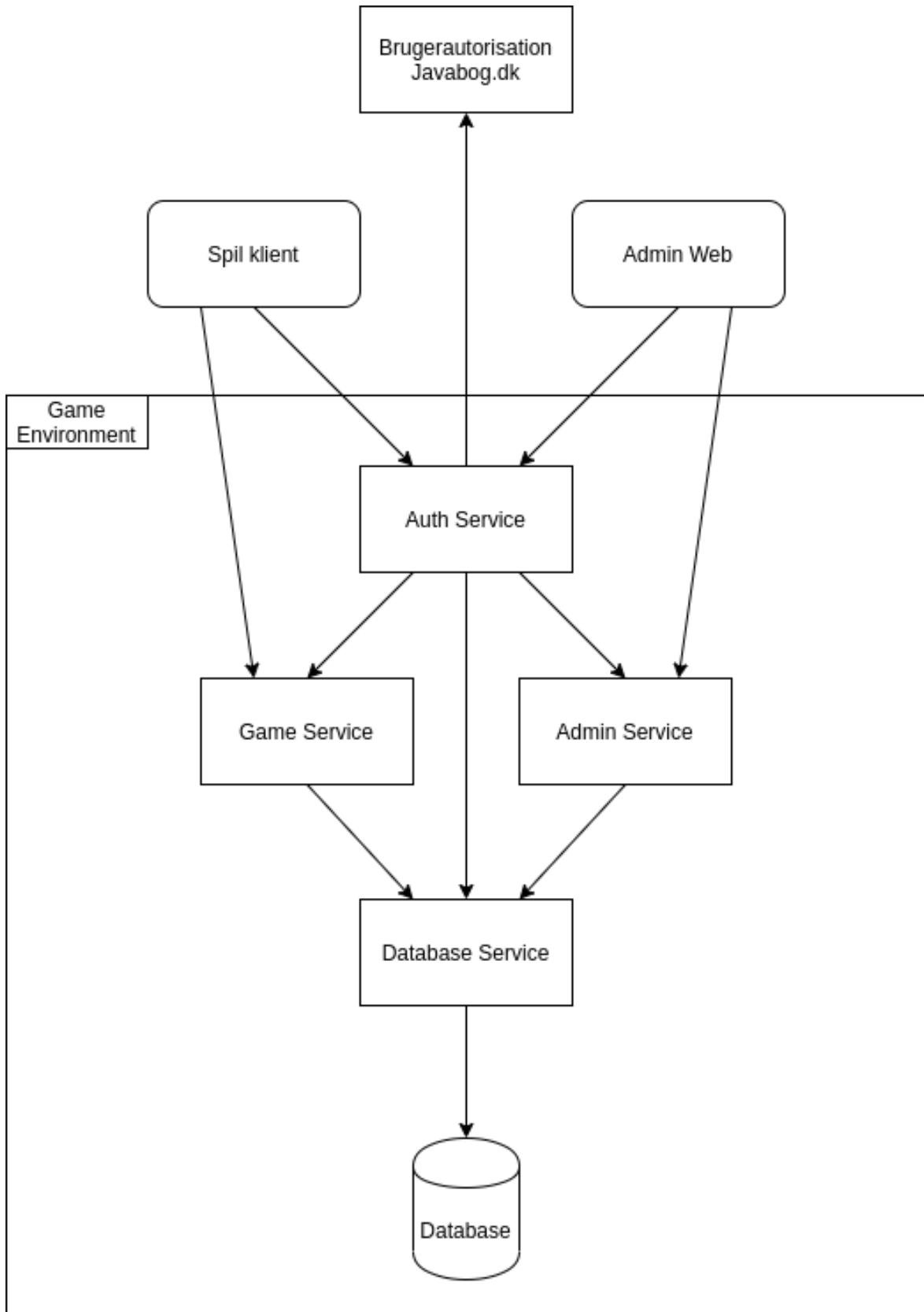
Vi kunne sagtens have testet mere, skrevet unit tests, lavet monkey tests, bruger tests med andre personer, men vi blev presset for tid, og har måttet allokere alt vores tid til at udvikle, og få de forskellige services til at arbejde rigtigt sammen.

9 Bilag

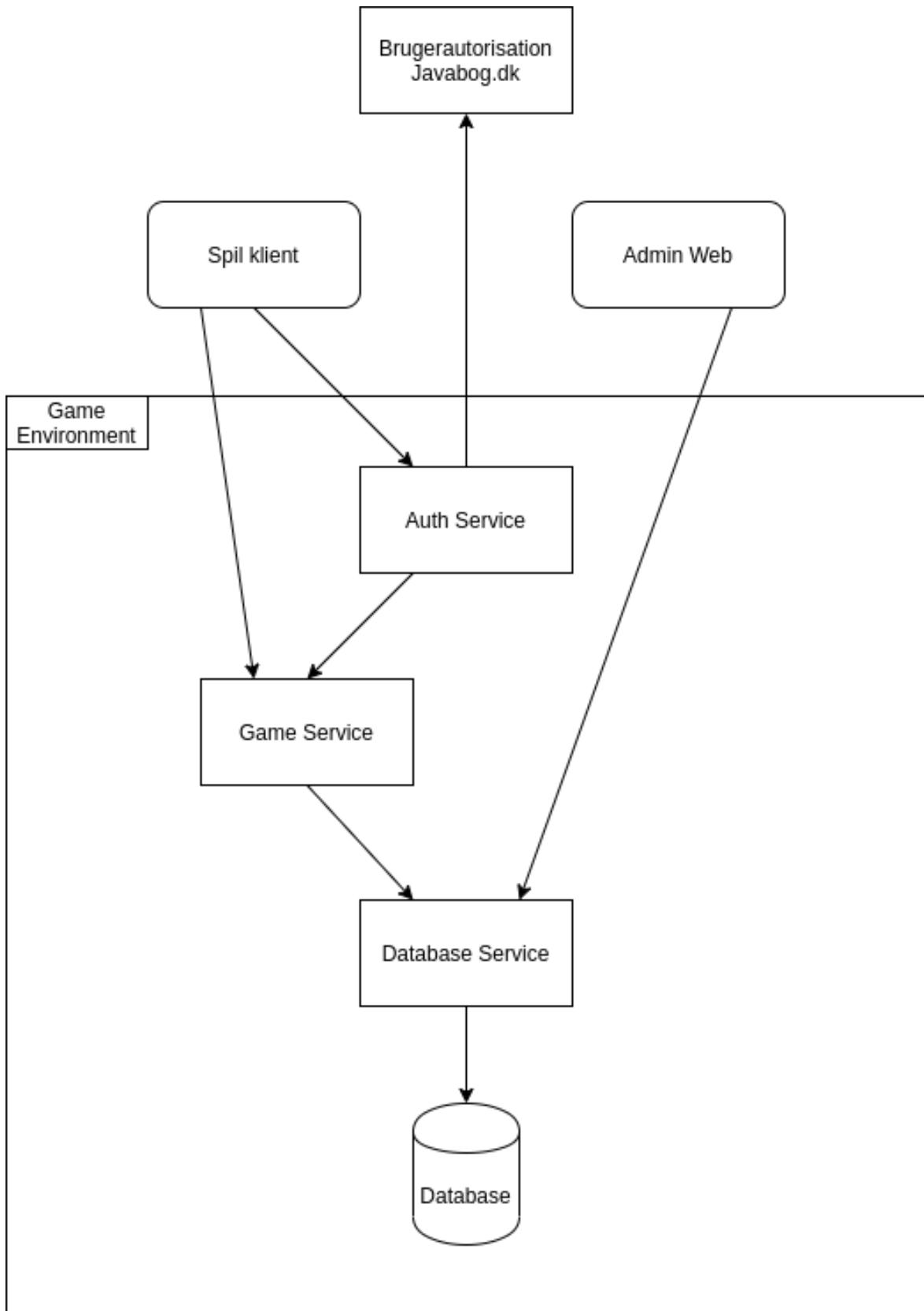
## A Deployment Diagram



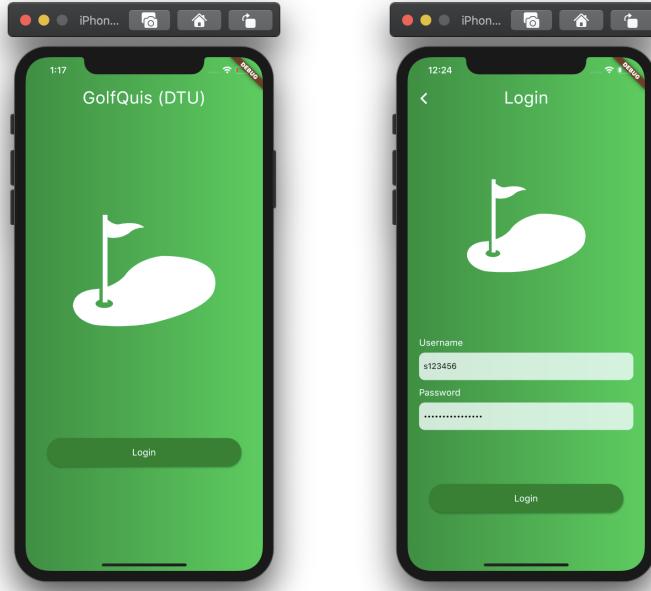
## B Frit billede



## C System Diagram

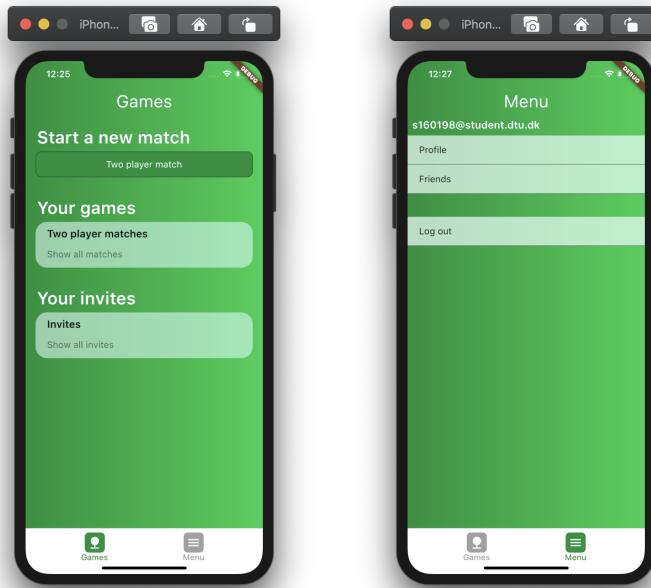


## D Brugergrænseflade Mobil Applikation - Flutter



Figur 7: Login

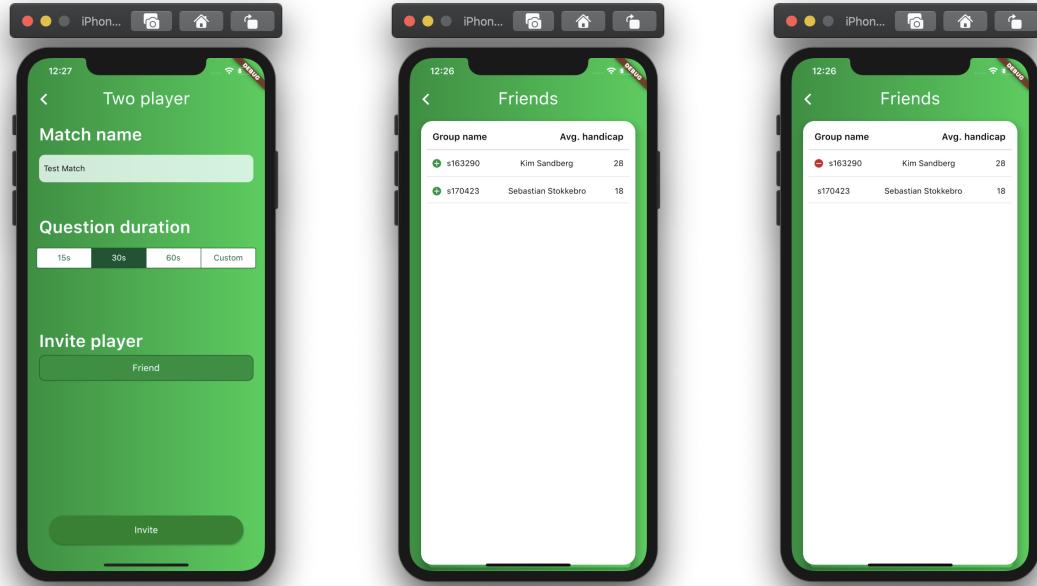
På Figur 7 ses de første skærme man bliver mødt med i appen. Her er der, i denne version, kun mulighed for at logge ind.



Figur 8: Hovedmenu

På Figur 8 ses de skærme man støder på efter login.

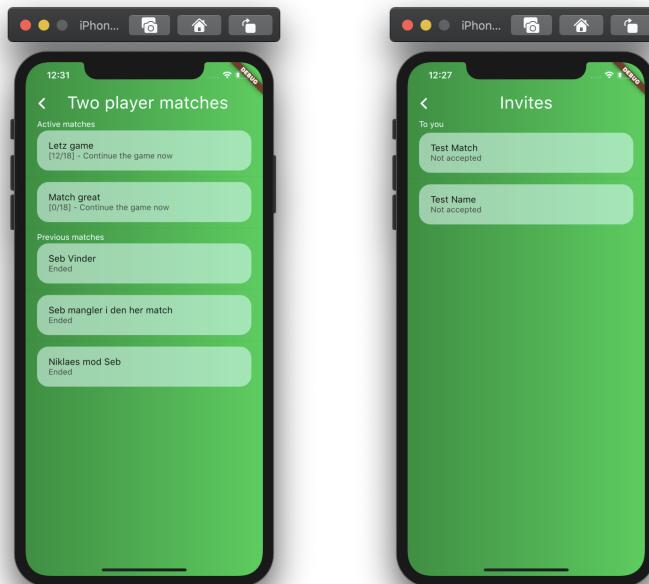
Under **Games** er der genveje til at invitere nye mennesker, se eksisterende spil, og se invitationer. Ved **Menu** findes der genveje til at se ens egen profil, sine venner, og mulighed for at logge ud.



Figur 9: Spil

På Figur 9 kan man se oprettelsen af et nyt invite.

Andet skærmbillede er igennem ”Friend” knappen fra første skærmbillede. Sidste skærmbillede er efter et klik på Kim, for at invitere ham.

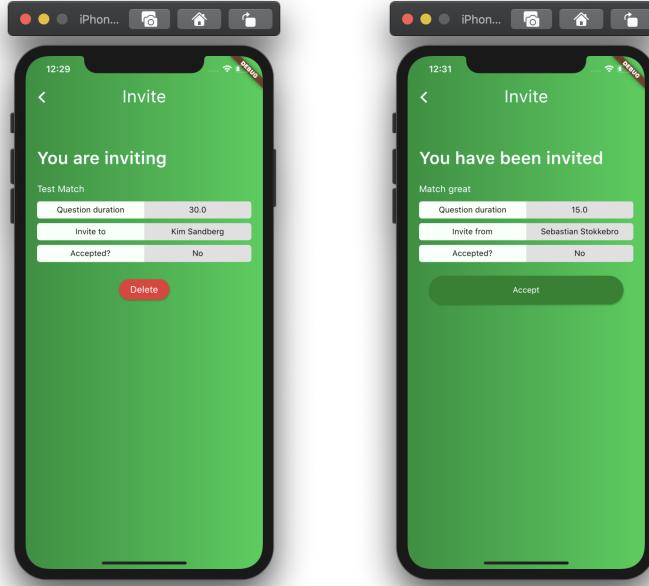


Figur 10: Spil og invitations liste

På Figur 10 ses spil og invitationer.

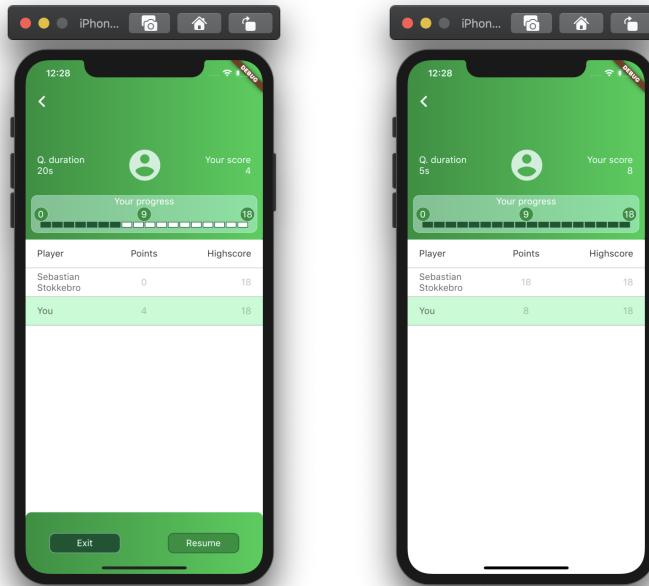
Under **Two player matches** ses alle eksisterende spil.

Ved **Invites** ses alle invitationer, som ikke er accepteret. Det er både dem du har sendt, og dem der er sendt til dig.



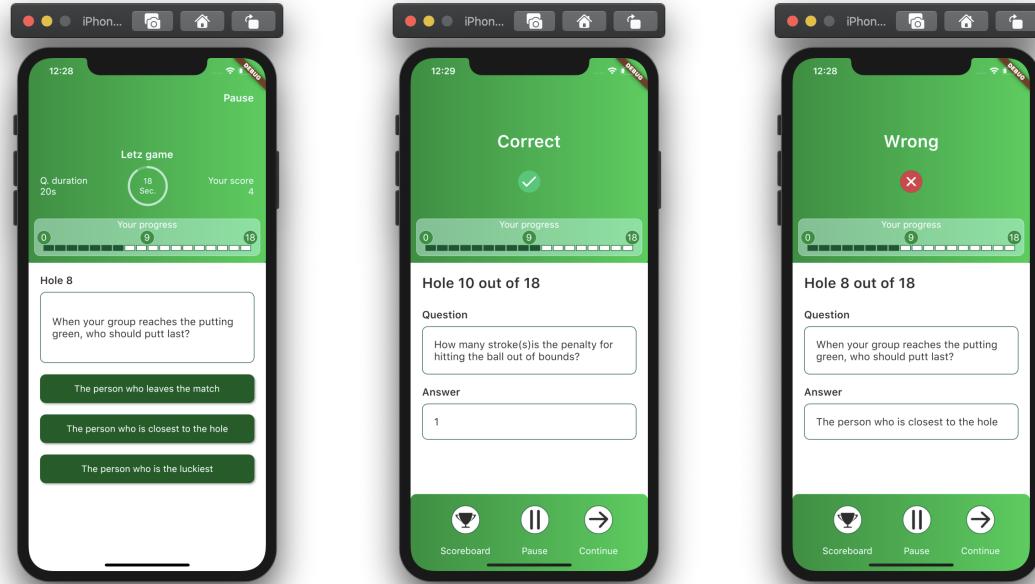
Figur 11: Invites

På Figur 11 ses et enkelt invite, til venstre et invite du har sendt, og til højre et invite der er sendt til dig. Efter et klik på ”Accept knappen”(Højre billede) vil man blive ført til spil listen, hvor det nye spil vil ligge.



Figur 12: Spil

På Figur 12 ses en skærm der vises når man klikker på det enkelte spil i spil listen. Til venstre ses et spil der stadig er i gang, og til højre et spil, som er sluttet.

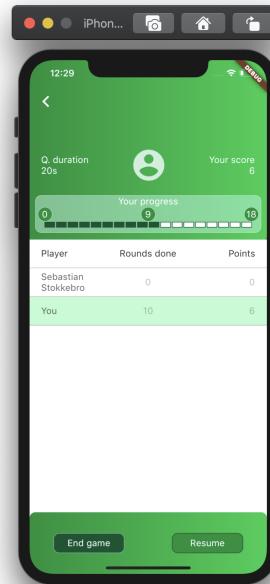


Figur 13: Spillet i action

På Figur 13 ses spillet i aktion.

Undervejs vil man blive spurgt spørgsmål, som man svarer med svar-knapperne.

Herefter vil man blive ført til enten billede 2 eller 3, hvor man vil få at vide om det var korrekt eller ej. Man får 2 points pr. rigtig runde, og 0 for en forkert runde.



Figur 14: Scoreboard

På Figur 14 ses scoreboard for spillet.

Efter de 18 runder er spillet, vil man blive ført til scoreboard. Der er også mulighed for at se på dette scoreboard undervejs i spillet.

## E Brugergrænseflade Web interfacet - VueJS

Når man logger ind som administrator vil man blive mødt af følgende skærm. Dette er oversigten over brugerne hvor der er mulighed for at tilføje, ændre og slette en bruger.

ID	First Name	Last Name	Study Programme	Username	Email	High Score		
3	Niklaes Dino Robbin	Jacobsen	Softwaretek.	s160198	s160198@student.dtu.dk	18		
4	Kim Sandberg	Bossen	Softwaretek.	s163290	s163290@student.dtu.dk	28		
5	Sebastian Stokkebro	Sørensen	Softwaretek.	s170423	s170423@student.dtu.dk	18		

Figur 15: Brugere viewet fra administrator siden

I venstre side er der en navigationsbar. Her kan man navigere mellem brugere, spil, spørgsmål og invitationer. I bunden af navigationsbaren er der mulighed for at logge ud. I det viste eksempel tages der blot udgangspunkt i *brugere-viewet* da de andre views er helt identiske.

For at slette noget i tabellen skal der blot klikkes på det røde skraldespandsikon til højre i rækken. Ønsker man at redigere i rækken klikker man på blyantsikonet og får følgende view.

The screenshot shows a table titled "Users" with columns: ID, First Name, Last Name, Study Programme, Username, Email, and High Score. There are three rows of data:

ID	First Name	Last Name	Study Programme	Username	Email	High Score
3	Niklaes Dino Rob	Jacobsen	Softwaretek.	s160198	s160198@student.dtu.dk	18
4	Kim Sandberg	Bosse	Softwaretek.	s163290	s163290@student.dtu.dk	28
5	Sebastian Stokkebro	Sørensen	Softwaretek.	s170423	s170423@student.dtu.dk	18

A vertical sidebar on the left contains icons for users, game, help, add, and back.

Figur 16: Når en række skal redigeres skiftes cellerne ud med et input felt hvor administratoren kan ændre værdierne i disse

For at tilføje en bruger trykker man på *add* knappen øverste i venstre hjørne, hvor man sendes til *tilføj-viewet*. Her mødes man af en tabel række hvor man kan indtaste de informationer, der skal tilføjes til en tabel.

The screenshot shows a form titled "Add User" with fields for First Name, Last Name, Study Programme, Username, Email, and High Score. All fields are currently empty. A green "+" button is located to the right of the form.

A vertical sidebar on the left contains icons for users, game, help, add, and back.

Figur 17: Her kan administratoren tilføje en bruger

## F Implementering af Autorisations flow

### Flutter App

I Flutter appen sendes et HTTP post kald igennem `http4`, en native pakke i dart. Der sendes headers med, som defineres på linje 1, og en json string som defineres på 5 og 6. Der defineres samtidigt en Uri, som indeholder en url konstant, som er "87.61.85.141:9800" i den live version. Her bliver der tilføjet `/login/`, som er stien for login. Til sidst bruges et bibliotek, udgivet af Google.dev, som hedder `Retry5`. Denne pakke hjælper os til at gentage kaldet, hvis der blot er en `SocketException` eller en `TimeoutException`. `TimeoutException` sker når de 5 sekunder, som er defineret på linje 12 udløber. `Retry` sørger for at kaldet forsøges 8 gange, med en pause imellem dem alle, som stiger for hver gang.

---

```

1 class AuthService {
2     static Future login(String username, String password) async {
3     ...
4     ...
5     Map<String, String> headers = {
6         HttpHeaders.acceptEncodingHeader: "application/json",
7     };
8
9     var jsonMap = {"username": username, "password": password};
10    String json = jsonEncode(jsonMap);
11
12    Uri uri = Uri.http(ServiceConstants.baseAuthUrl, "/login/");
13
14    return retry(
15        () => http.post(uri, headers: headers, body: json)
16        .timeout(Duration(seconds: 5))
17        ...
18        ...
19        retryIf: (e) {
20            if (e is SocketException || e is TimeoutException) {
21                return true;
22            } else {
23                return false;
24            }
25        },
26    );

```

---

<sup>4</sup><https://pub.dev/packages/http>

<sup>5</sup><https://pub.dev/packages/retry>

### Auth Service

Vi bruger Django<sup>6</sup> til at opsætte REST endpoints i Auth Service.

#### Parsing af request

Til at starte med bliver brugernavn og password taget ud af det request der sendes.

---

```

1 @api_view(['POST'])
2 def login(request):
3     ...
4     ...
5     decoded = request.body.decode('utf-8')
6     response = json.loads(decoded)
7     username = response['username']
8     password = response['password']

```

---

#### Forbindelse til javabog

Her ses forbindelsen der sker til javabog.dk's brugerautentifikation, som sker i samme login metode. Der bruges Client til SOAP kaldet, som er fra biblioteket zeep<sup>7</sup>. Efterfølgende defineres en dictionary (Python's map-funktionalitet), som indeholder spillerens oplysninger. Hvis enten at javabog.dk's brugerautentifikation giver fejl tilbage, eller at der er skrevet forkert brugernavn/password, sendes dette tilbage til Flutter App.

---

```

1 @api_view(['POST'])
2 def login(request):
3     ...
4     ...
5     # Create a soap client object
6     url = 'http://javabog.dk:9901/brugeradmin?wsdl'
7     ...
8     ...
9     client = Client(url)
10    # Get user and create token
11    response = client.service.hentBruger(username, password)
12
13    # create the userobject for GameService
14    user_object = {
15        'player':
16            {
17                "username": response["brugernavn"],
18                "first_name": response["fornavn"],
19                "last_name": response["efternavn"],
20                "email": response["email"],
21                "study_programme": response["studeretning"],
22                "high_score": 0
23            }
24    }
25    # create the user_token. This should be exchanged for a creation of a JWT.
26    user_token = str(uuid.uuid1())
27
28    # Register user at gameservice and get gameservice ip and port
29    game_service_ip, game_service_port, player = register_user_with_game_service(
30                                            AUTH_SERVICE_ACCESS_KEY,
31                                            user_token,
32                                            user_object, logfile)

```

---

<sup>6</sup><https://pypi.org/project/Django/>

<sup>7</sup><https://pypi.org/project/zeep/>

### Forbindelse til Game Service

Her ses det POST kald der skabes til Game Service. Fra svaret bruges Game Service ip, Game Service port og spilleren fra det respons der kommer tilbage. Der bruges en miljøvariabel "DC\_GS", til at definere ip til Game Service. Dette bruges i sammenhæng med de docker opsætninger vi har sat op, som gør det nemt at rette det et sted. Der sendes samtidigt en service\_key, som tjekkes i Game Service.

---

```

1 def register_user_with_game_service(service_key, user_token, user_object, logfile):
2     compose_env_url = os.getenv('DC_GS')
3
4     ...
5     URL = "http://" + str(compose_env_url) + ":9700/register_user/"
6
7     body_data = {
8         "service_key": service_key,
9         "user_token": user_token,
10        "user_object": user_object
11    }
12
13    ...
14
15    # Connect to gameservice for registration of the user in our db,
16    # and setting the user_token.
17    r = requests.post(url=URL, json=body_data)
18
19    #json decode the response
20    data = json.loads(r.content.decode("UTF-8").__str__())
21
22    # extract the needed data from the response
23    game_service_ip = data["game_service_ip"]
24    game_service_port = data['game_service_port']
      player = data['player']

```

---

### Respons til Flutter App

Her sendes det respons der kommer fra Game Service videre til Flutter App. Der sendes samtidigt også ip og port til gameservice videre til flutter app, som gemmes og bruges til fremtidige kald.

---

```

1 @api_view(['POST'])
2 def login(request):
3
4     ...
5
6     # If gameservice responds with proper data,
7     # return the userobject, usertoken and gameservice ip and port to client.
8     user_object['user_token'] = user_token
9     user_object['game_service_ip'] = game_service_ip
10    user_object['game_service_port'] = game_service_port
11    user_object['player'] = player
12
13    ...
14
15    return JsonResponse(user_object, status=status.HTTP_200_OK)

```

---

## Game Service

Vi bruger Django<sup>8</sup> til at oprette REST endpoints i Game Service.

### Forbind til Database Service

Til at starte med tjekkes der for den service-key der sendes via body fra Auth Service. Det er kun i dette endpoint i Game Service, hvor der sendes en service-key.

Der bruges i metoden ”check\_or\_add\_user(user)”, linje 4, en metode der kaldes ”connection\_service”. Denne metode er også vist nedenunder, og forbinder til databaseservice-ip med headers og body. Metoden ”check\_or\_add\_user(user)” sørger for at lave en GET på den enkelte bruger til at starte med. Hvis brugeren findes vil der blot blive sendt brugeren tilbage til AuthService. Hvis brugeren ikke findes, vil der blive oprettet en ny bruger i Database Service.

---

```

1 @api_view(['POST'])
2 def register_user(request):
3     ...
4     ...
5     if req_json['service_key'] == AUTH_SERVICE_ACCESS_KEY:
6         if player['id'] not in token_player_list:
7             ...
8             ...
9         player = check_or_add_user(req_json['user_object'])

```

---

```

1 def check_or_add_user(user):
2     ...
3     ...
4     response_data = connection_service("/players/" + dtu_username + "/", None, None, "GET")
5
6     data = json.loads(response_data.content)
7     ...
8     ...
9     player = data['player']
10    return player

```

---

```

1 def connection_service(endpoint_url, body_data, query_params, method):
2     build_url = database_service_url() + endpoint_url
3     ...
4     ...
5     if method == "POST":
6         if body_data is not None:
7             r = requests.post(url=build_url + str(params), json=body_data, headers=headers)
8         else:
9             r = requests.get(url=build_url + str(params), headers=headers)
10        ...
11        ...
12    return r

```

---

<sup>8</sup><https://pypi.org/project/Django/>

### Send respons tilbage

Her ses det data der så sendes tilbage til Auth Service efter en successfuld forbindelse til Database Service.

---

```

1 @api_view(['POST'])
2 def register_user(request):
3     ...
4     ...
5     game_service_ip = os.getenv('DC_GS_SELF')
6
7     if (game_service_ip is None):
8         game_service_ip = "127.0.0.1"
9     game_service_port = "9700"
10    response = {"game_service_ip": game_service_ip, "game_service_port": game_service_port,
11                "player": player}
12    return Response(response, status=status.HTTP_200_OK)

```

---

### Database Service

Vi bruger Django<sup>9</sup> til at opsætte REST endpoints i Database Service. Der ses herunder det eksempel hvor spiller ikke eksisterer, og der derfor skal oprettes en ny spiller. Ved metoden ”PlayerDatabase.create\_return\_serialized(json\_body)”sker oprettelsen af brugeren udfra det request der kommer fra Game Service. Der returneres et JsonResponse med den nye bruger, og HTTP-koden 201.

---

```

1 def __players_post(request):
2     ...
3     ...
4     json_body = json.loads(request.body)
5
6     # Create a database entry
7     return_data = PlayerDatabase.create_return_serialized(json_body)
8
9     # if it returns a string, send a missing property json back
10    if isinstance(return_data, str):
11        return missing_property_in_json(request, return_data, CORRECT_PLAYER_JSON)
12
13    # Prepare jsonResponse data
14    json_data = {
15        'requested-url': '[' + request.method + '] ' + request.get_full_path(),
16        'message': 'You have posted a new player',
17        'player': return_data
18    }
19    return JsonResponse(data=json_data, status=status.HTTP_201_CREATED, safe=False, encoder=Django

```

---

<sup>9</sup><https://pypi.org/project/Django/>

## G Implementering af Opret ny spiller flow

Opret ny spiller flow [Web klient → Database Service].

Dette flow er processen der sker når man opretter en bruger fra administrator sitet.

### Web klient

Web klienten bruger JavaScripts `fetch` metode til at håndtere api kald. I denne skal der gives en api url, samt informationer omkring hvilken HTTP metode der bruges, headers og eventuel body data. Følgende kode viser det post request der laves ved oprettelse af en bruger.

---

```

1   fetch(api_players, {
2       method: 'POST',
3       body: payload,
4       headers: auth_header
5   })
6   .then(response => {
7       if(response.status === 201) {
8           showModal('Player successfully added!')
9           fields.forEach(field => {
10               field.value = ''
11           })
12       } else {
13           showModal('Something went wrong...')
14       }
15   })
16   .catch(error => {
17       showModal('Something went wrong...'))
18   })

```

---

Som det kan ses håndteres fejlkoder ikke på administrator siden, der vises blot et modal vindue som fortæller at noget gik galt, skulle der ske en fejl. Efter dette kald sker der følgende i Database Service.

### Database Service

Vi bruger Django<sup>10</sup> til at opsætte REST endpoints i Database Service. I følgende snippet ses hvordan `_players_post` metoden modtager en payload og opretter en instans af dette som et bruger objekt i `sqlite` databasen.

---

```

1 def __players_post(request):
2     print_origin(request, 'Players')
3     ...
4     ...
5     # Create a database entry
6     return_data = PlayerDatabase.create_return_serialized(json_body)
7
8     # if it returns a string, send a missing property json back
9     if isinstance(return_data, str):
10         return missing_property_in_json(request, return_data, CORRECT_PLAYER_JSON)
11
12     # Prepare jsonResponse data
13     json_data = {
14         'requested-url': '[' + request.method + ']' + request.get_full_path(),
15         'message': 'You have posted a new player',
16         'player': return_data
17     }
18     return JsonResponse(data=json_data, status=status.HTTP_201_CREATED, safe=False,
19     encoder=DjangoJSONEncoder)

```

---

<sup>10</sup><https://pypi.org/project/Django/>