Høgskulen på Vestlandet

Fakultet for teknologi, miljø- og samfunnsvitenskap

Institutt for datateknologi, elektroteknologi og realfag

Anbudsassistent - Generering av anbud med kunstig intelligens

Systemdokumentasjon

Versjon 1.1.1

REVISJONSHISTORIE

Dato	Versjon	Beskrivelse	Forfatter
12/04/24	1.0	Første utkast	Martin, Nora, Torbjørn
29/04/24	1.1	Andre utkast	Martin, Nora, Torbjørn
10/05/24	1.1.1	Formatering	Nora, Martin

INNHOLDSFORTEGNELSE

1 INNLEDNING

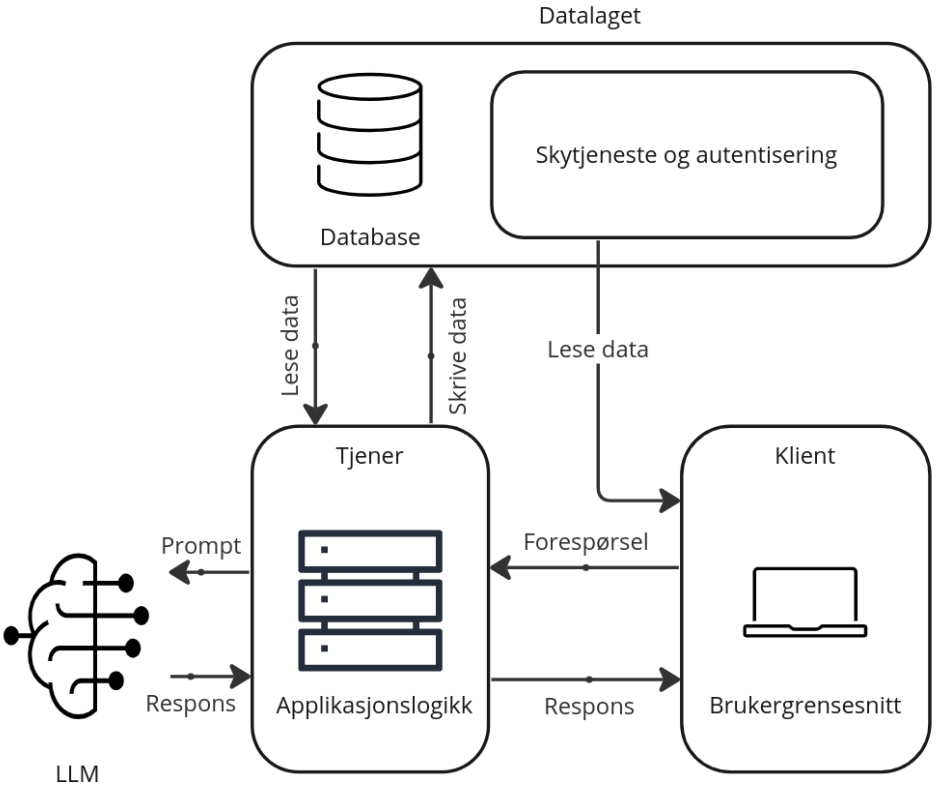
Systemdokumentasjonen går i detalj om de ulike delene av applikasjonen. Det vil virke som støttedokumentasjon for rapporten, som gir mer kontekst ved bruk av eksempler og figurer. Dette inkluderer ulike deler av applikasjonen, som klient, tjener og database og hvordan disse kommuniserer. I tillegg vil det følge med dokumentasjon om applikasjonen, som hvordan man installerer og/eller kjører den.

2 ARKITEKTUR

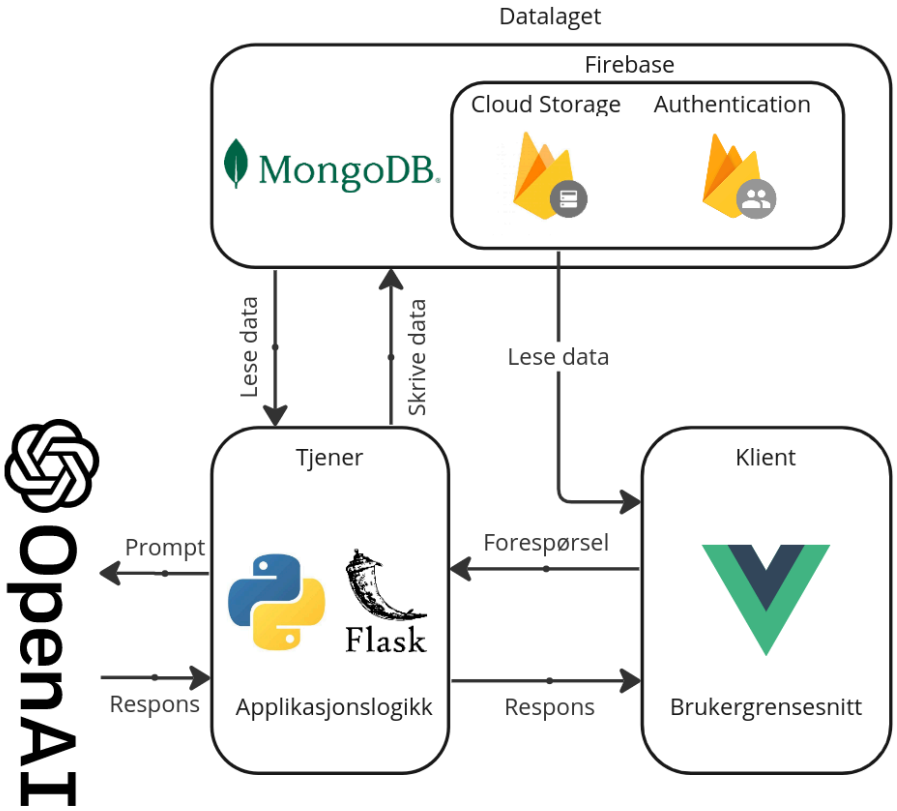
Applikasjonen bruker en klient-tjener-arkitektur med fire ulike deler.

Figur 2-1 beskriver applikasjonen på en generell måte uten produktnavn. Arkitekturen består av fire ulike hoveddeler, som kommuniserer mellom hverandre. Tjeneren styrer det meste av flyten som går mellom de ulike lagene, det eneste unntaket er at klienten kan hente data fra skytjenesten og hente brukerdata uten å gå innom tjeneren. Tjeneren i denne arkitekturen bruker et REST-API for å kommunisere med klienten.

Figur 2-2 viser arkitekturen på en mer detaljert måte med produktnavn og hvor de er plassert i systemets arkitektur.



Figur 2-1 - Systemarkitektur generell

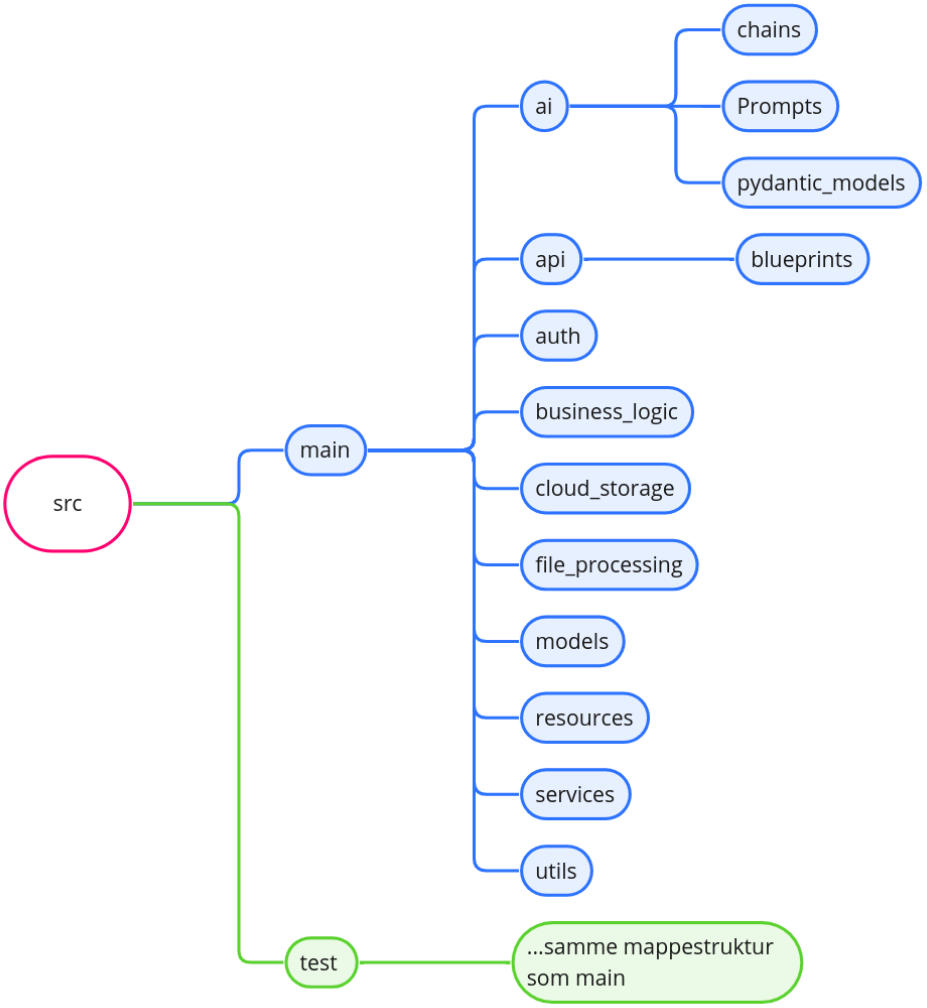


Figur 2-2 - Systemarkitektur detaljert

3 PROSJEKTSTRUKTUR

Systemet er delt opp i to ulike prosjekter, klienten og tjeneren. Dette er blitt gjort for å separere de ulike delene slik at det blir mer oversiktlig, og lettere og jobbe på de ulike delene samtidig uten å få konflikter.

3.1 Tjener



Figur 3-1 - Mappestruktur tjener

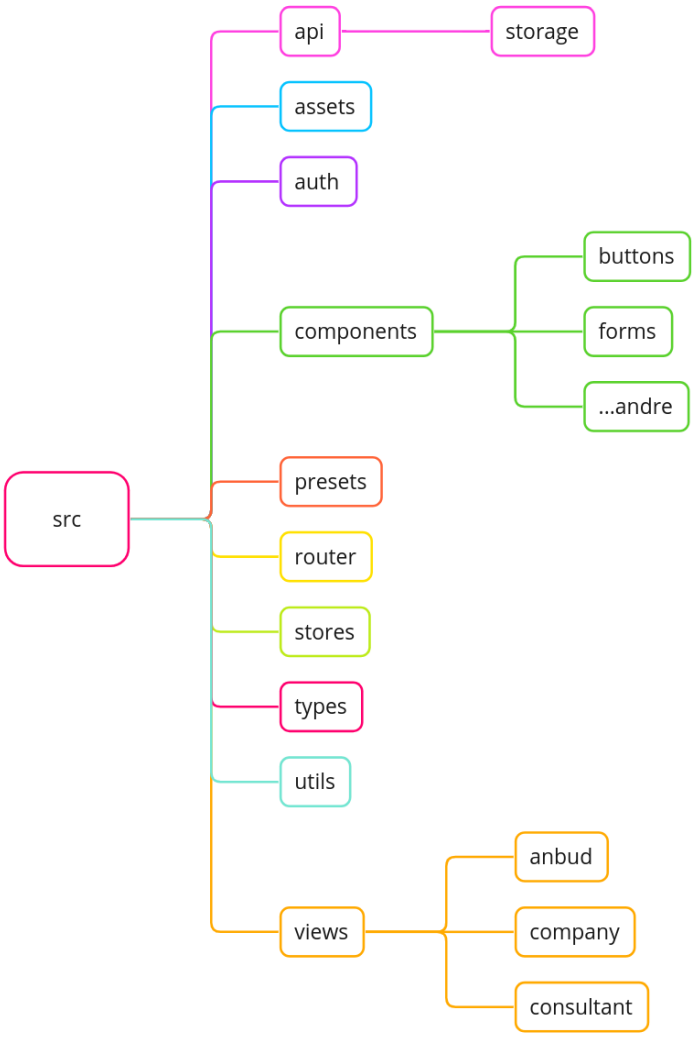
Biblioteker

Det er brukt en rekke med biblioteker og rammeverk for å hjelpe til med å lage applikasjonen. Tabell 3.1 viser de viktigste, men prosjektet inneholder en del andre biblioteker for mindre ting også.

Tabell 3-1 - Biblioteker tjener

Navn	Beskrivelse
Flask	Bibliotek for å lage enkle API-er
Pydantic	Bibliotek for å lage modeller for validering.
Flask-OpenAPI3	Bygger på Flask med autogenerering av OpenAPI dokumentasjon, samt validering med pydantic
LangChain	Inneholder mange nyttige hjelpefunksjoner for å jobbe med store språkmodeller, blant annet OpenAI sitt API.
PyMongo	Gjør det enkelt å skrive spørringer mot MongoDB, med mange ferdiglagde funksjoner.
Firebase-admin	Bibliotek for å jobbe med Firebase applikasjoner. Blant annet autentisering og skylagring.
Gunicorn	Distribuering med en WSGI tjener
Unittest	Bibliotek for enhetstesting av kode

3.2 Klient



Figur 3-2 - Mappestruktur klient

Biblioteker

Som tjeneren er klienten også bygget med diverse biblioteker og rammeverk.

Tabell 3-2 - Biblioteker klient

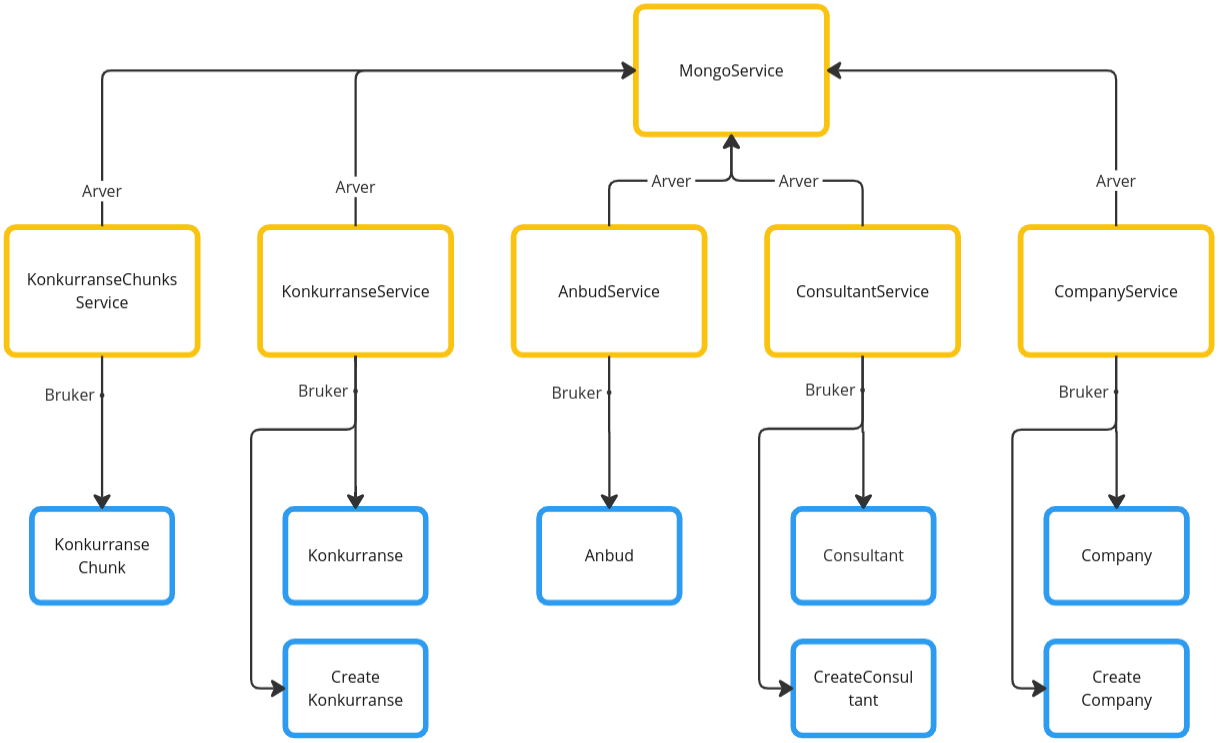
Navn	Beskrivelse
TailwindCSS	CSS bibliotek for å enkelt lage moderne design
TypeScript	En typesikker variant av JavaScript, som kompileres ned til vanlig JavaScript.
Vue.js	Rammeverk for å lage nettsider ved å lage og bruke komponenter
Pinia	Et bibliotek til Vue som gjør det lett å jobbe med tilstander mellom ulike komponenter.
Vue-router	Bibliotek for å bytte mellom ulike komponenter når man navigerer mellom ulike sider.
Firebase	Bibliotek for å jobbe mot firebase i klienten. Inneholder blant annet brukerautentisering og skylagring.
PrimeVue	Et komponentbibliotek med mange ferdiglagde komponenter.
PrimeIcons	Ikonbibliotek med mange ulike ikoner, som virker bra med PrimeVue komponentene.
Vitest	Bibliotek for enhetstesting av kode.

4 KLASSEDIAGRAM

4.1 Tjener

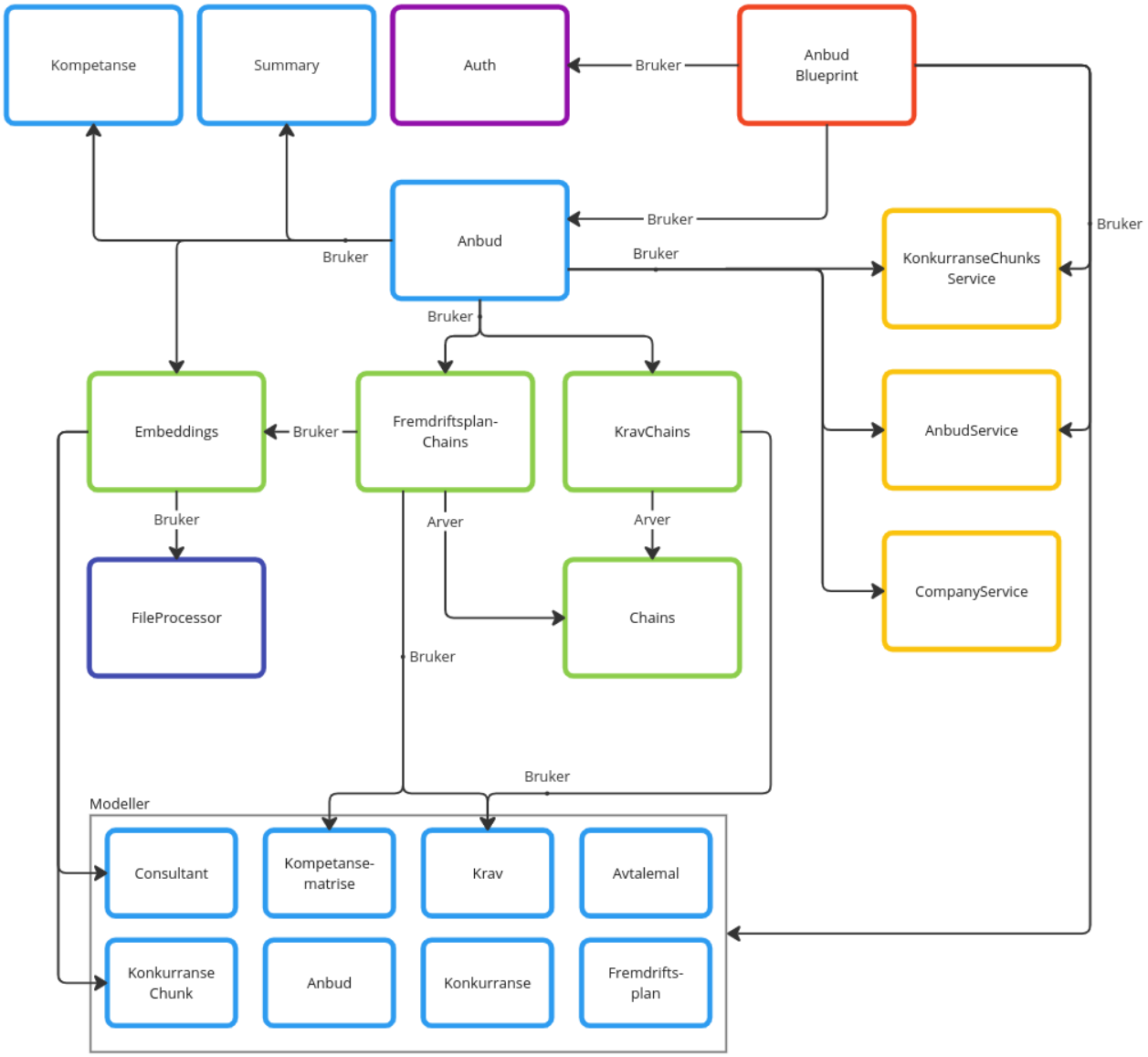
For å få en oversikt over forholdet mellom klasser på tjenersiden, må flere diagrammer til.

Det første diagrammet (Figur 4-1) viser Service-klassene som utfører CRUD-operasjoner mot databasen. Hver Service-klasse arver MongoService, som inneholder grunnleggende metoder for uthenting, oppdatering, lagring og sletting av data. Modell-klasser blir brukt for å sørge for at data er på riktig format. For å forenkle de videre diagrammene er ikke MongoService eller Modell-klassene som Service-klassene bruker, tatt med.

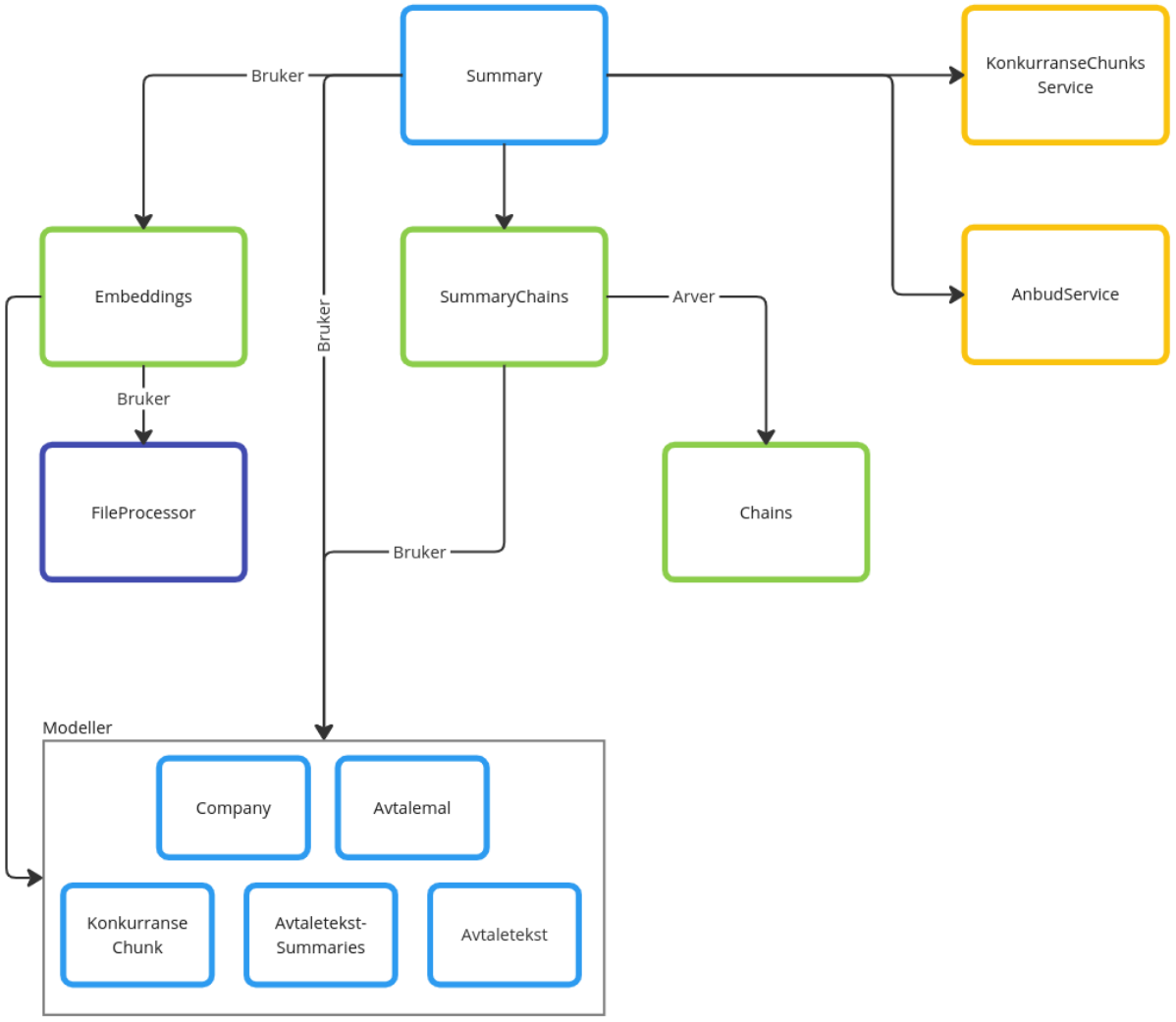


Figur 4-1 - Service-klassene arver "MongoService", og bruker Modell-klasser for å få inn data på riktig format

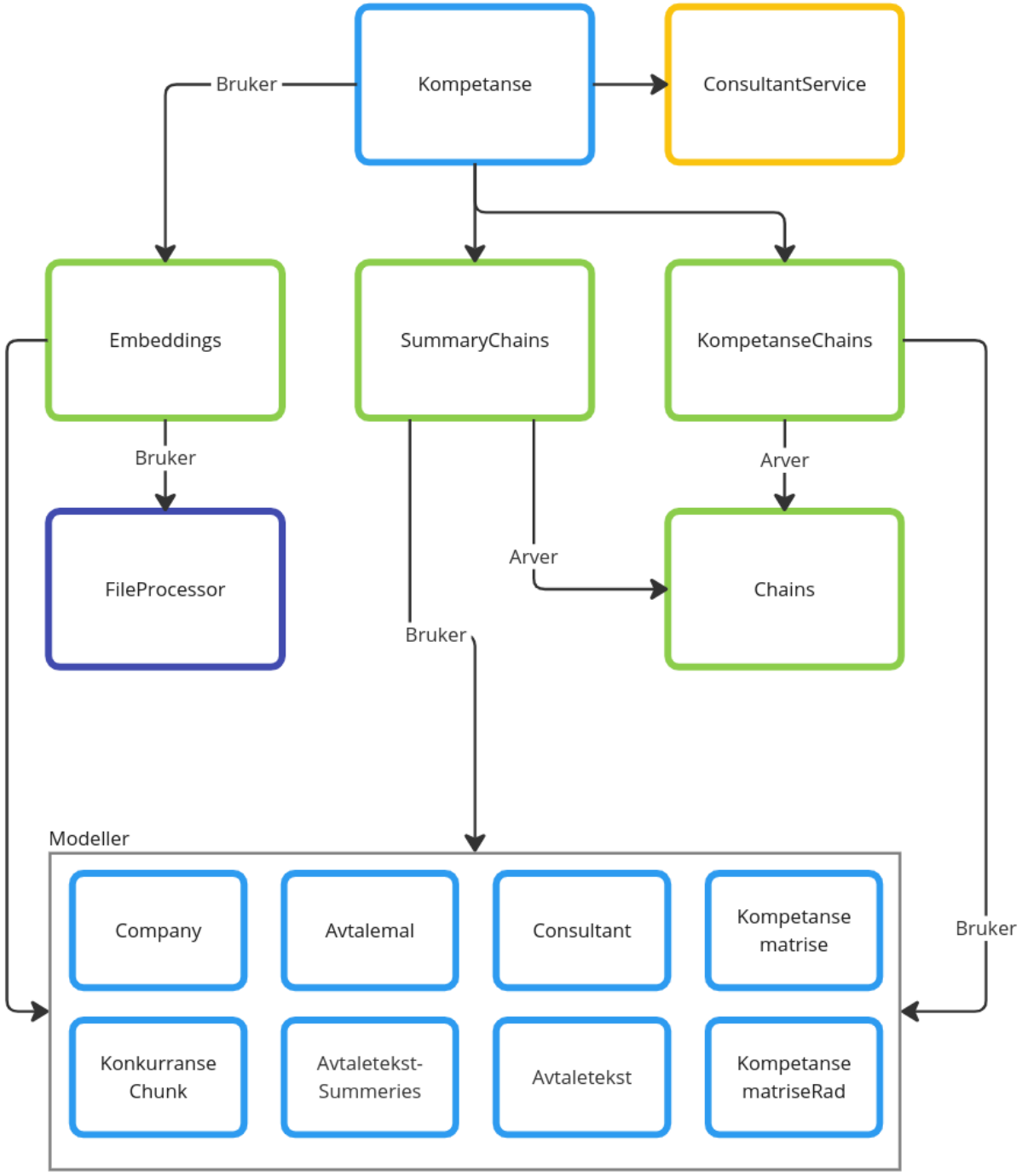
"AnbudBlueprint" håndterer endepunktene som angår oppsummering og anbud. En business-logic-klasse Anbud er benyttet som et mellomledd mellom service-klasser og chains-klasser. Modeller blir brukt for riktig formatering av data.



Figur 4-2 - AnbudBlueprint

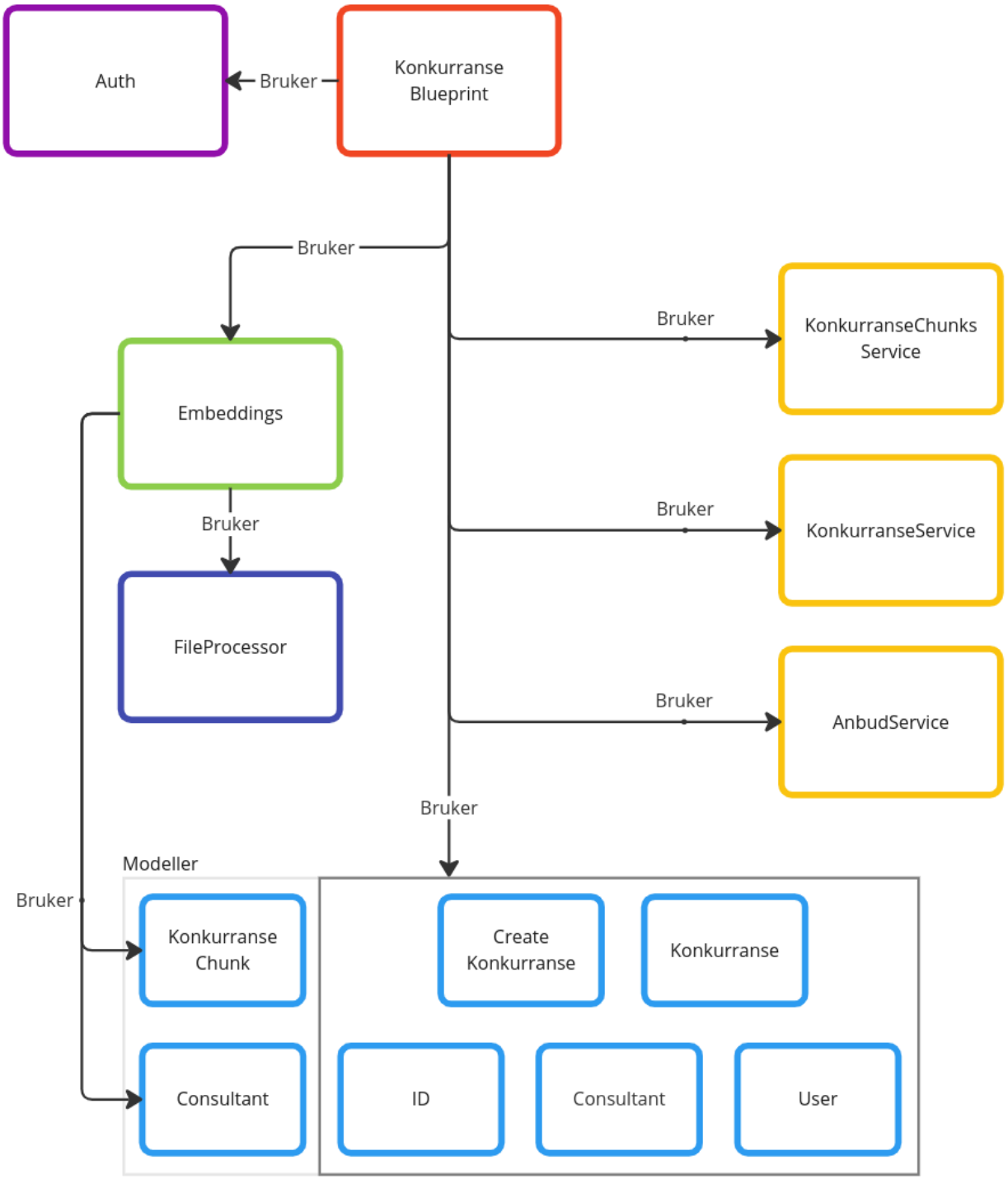


Figur 4-3 - Summary



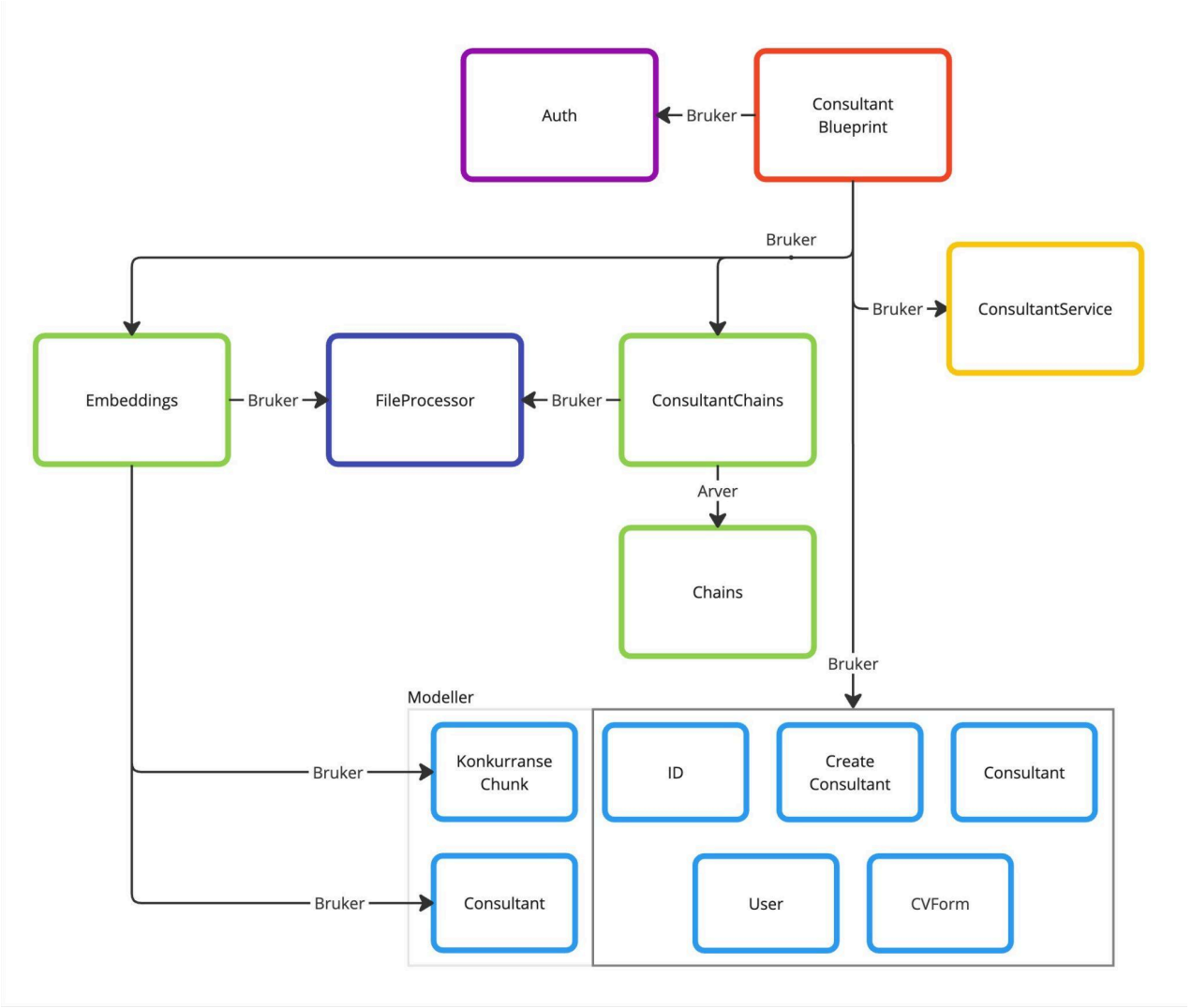
Figur 4-4 - Kompetanse

"KonkurranseBlueprint" inneholder endepunkter som har med oppretting, endring og sletting av konkurranser å gjøre, samt filopplasting.



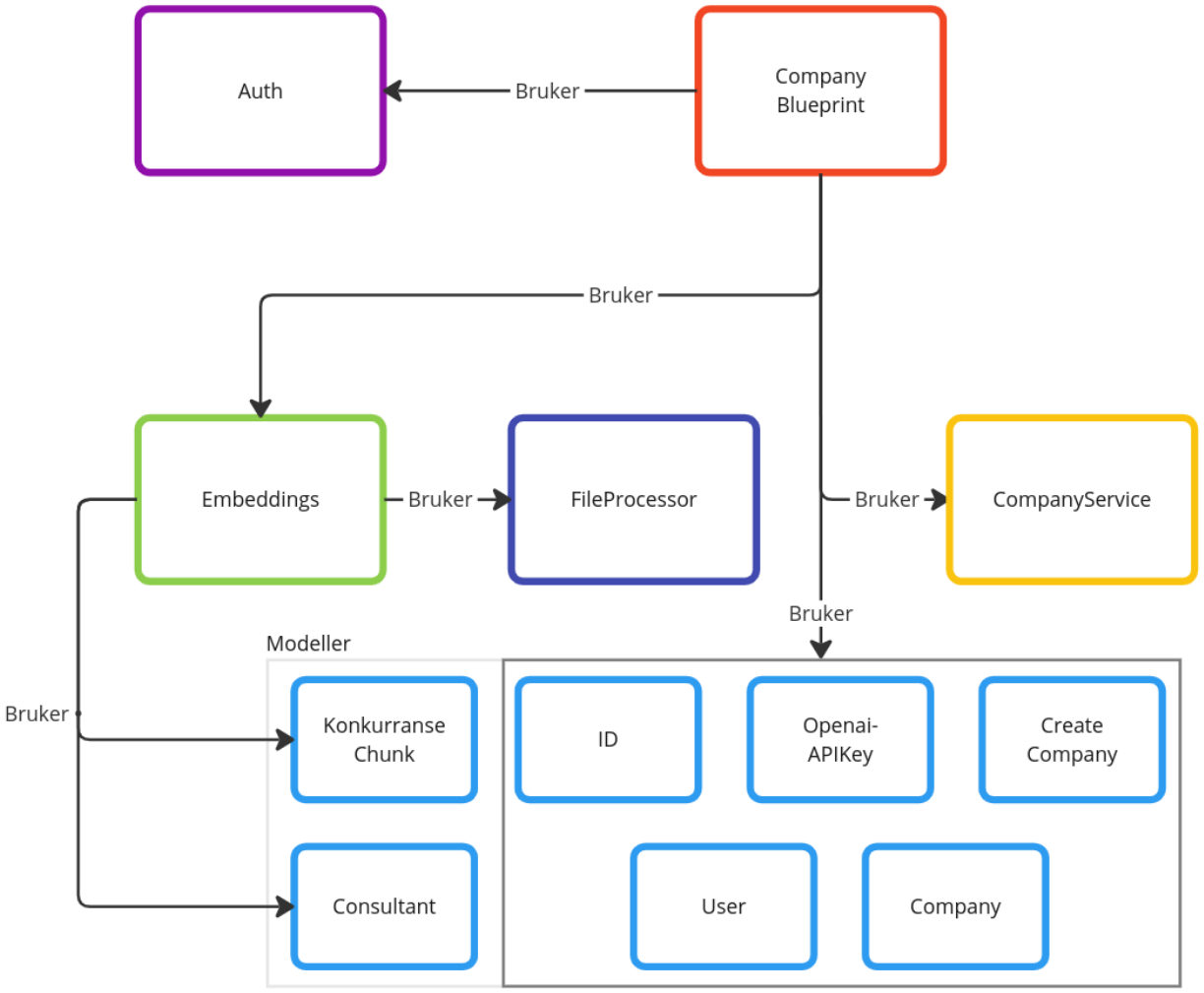
Figur 4-5 - KonkurranseBlueprint

“ConsultantBlueprint” inneholder endepunkter for oppretting, endring og sletting av konsulenter i en bedrift, samt opplasting av CV.



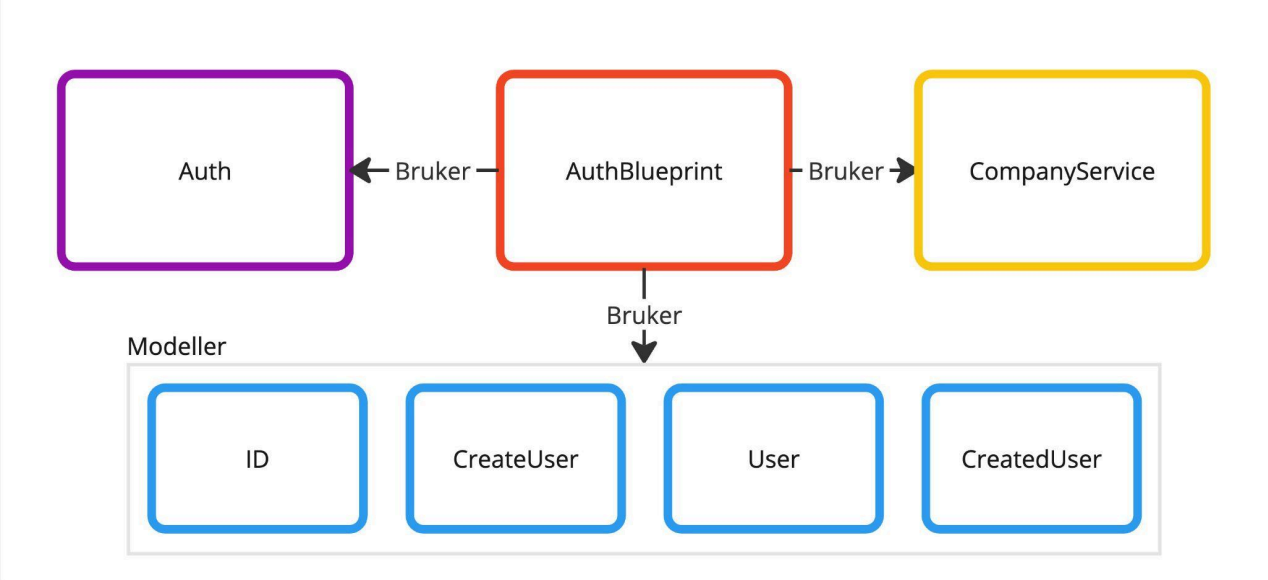
Figur 4-6 - ConsultantBlueprint

CompanyBlueprint inneholder endepunkter for oppretting, endring og sletting av bedrifter i systemet.



Figur 4-7 - CompanyBlueprint

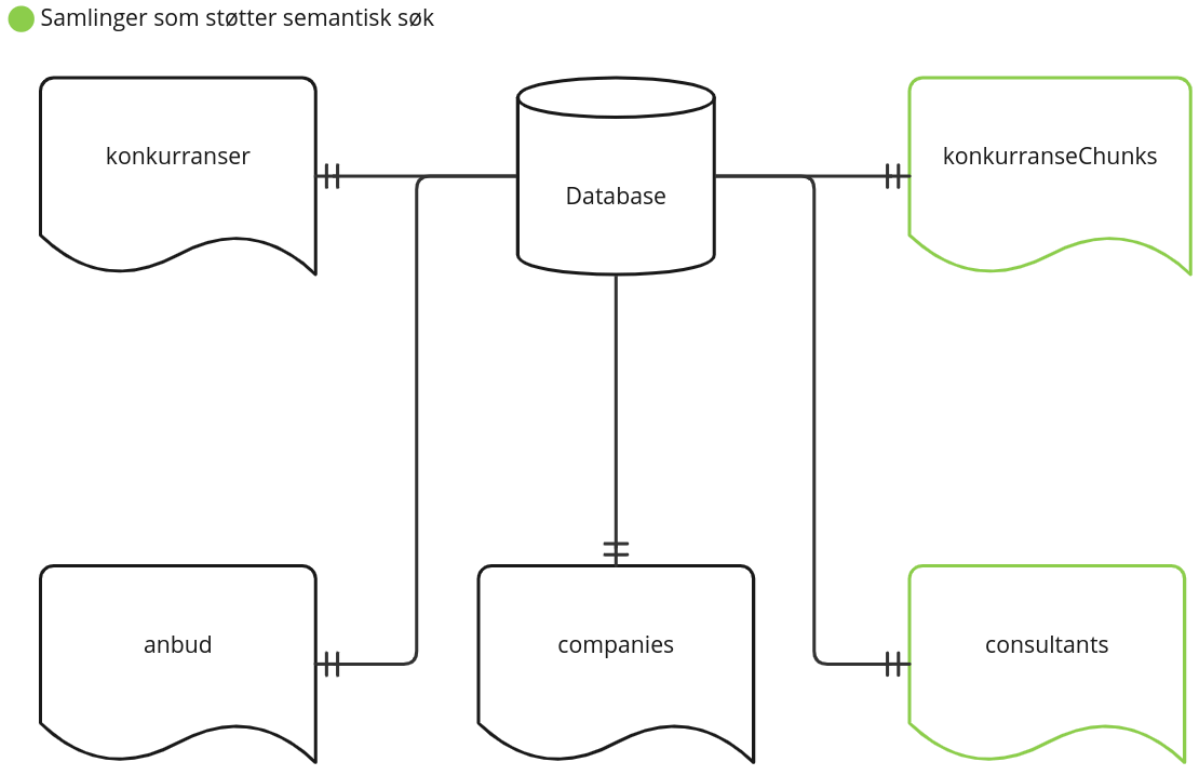
AuthBlueprint inneholder endepunkter for oppretting, endring og sletting av brukere i systemet.



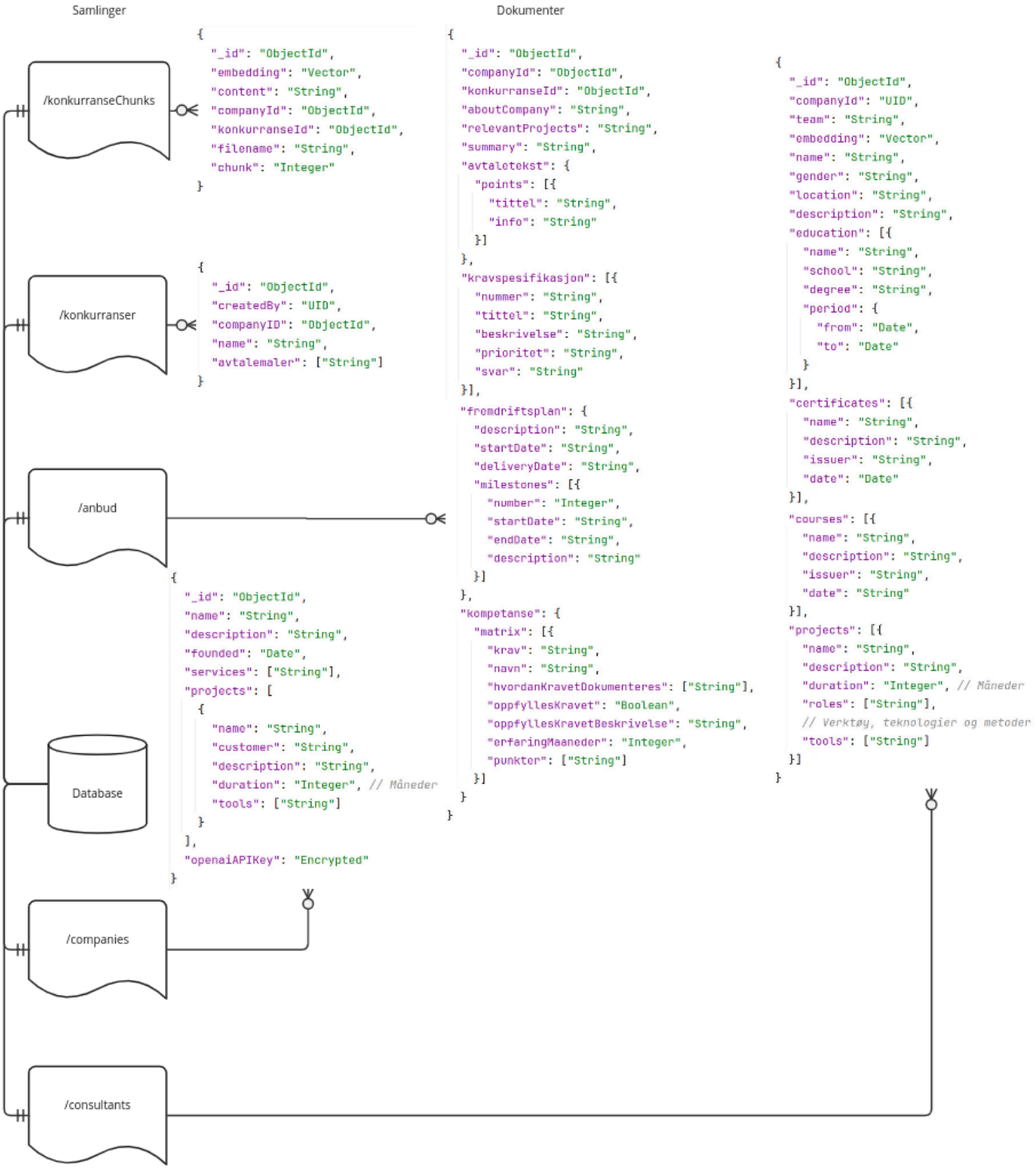
Figur 4-8 - AuthBlueprint

5 DATABASEMODELL

Figur 5-1 beskriver hvordan MongoDB databasen er bygd opp. MongoDB er en dokumentdatabase som er bygd opp slik at hver bruker kan sette opp flere databaser, hvor hver database kan ha flere samlinger og hver samling kan ha flere dokumenter. Disse dokumentene inneholder data på JSON struktur, som er lett å jobbe med og veldig fleksibelt. Denne databasen støtter også lagring av embeddinger og semantisk søking på disse. Figur 5-2 viser innholdet i disse samlingene, med navn på feltene og hvilken type de har.



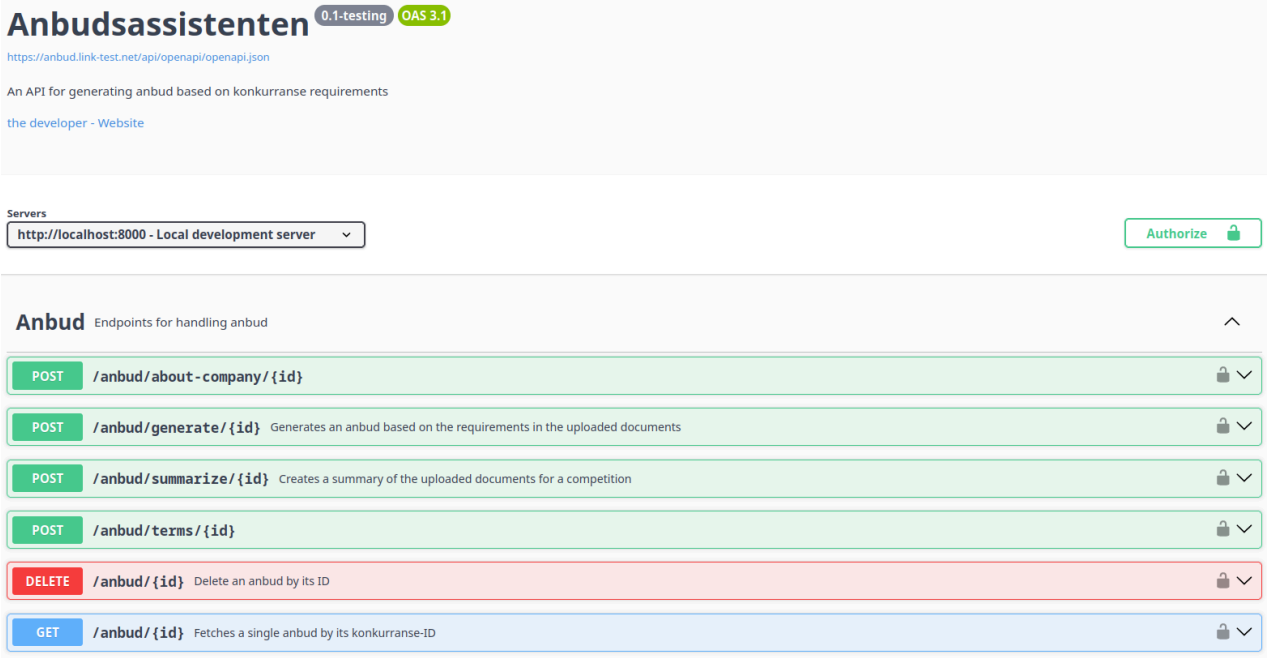
Figur 5-1 - MongoDB samlinger



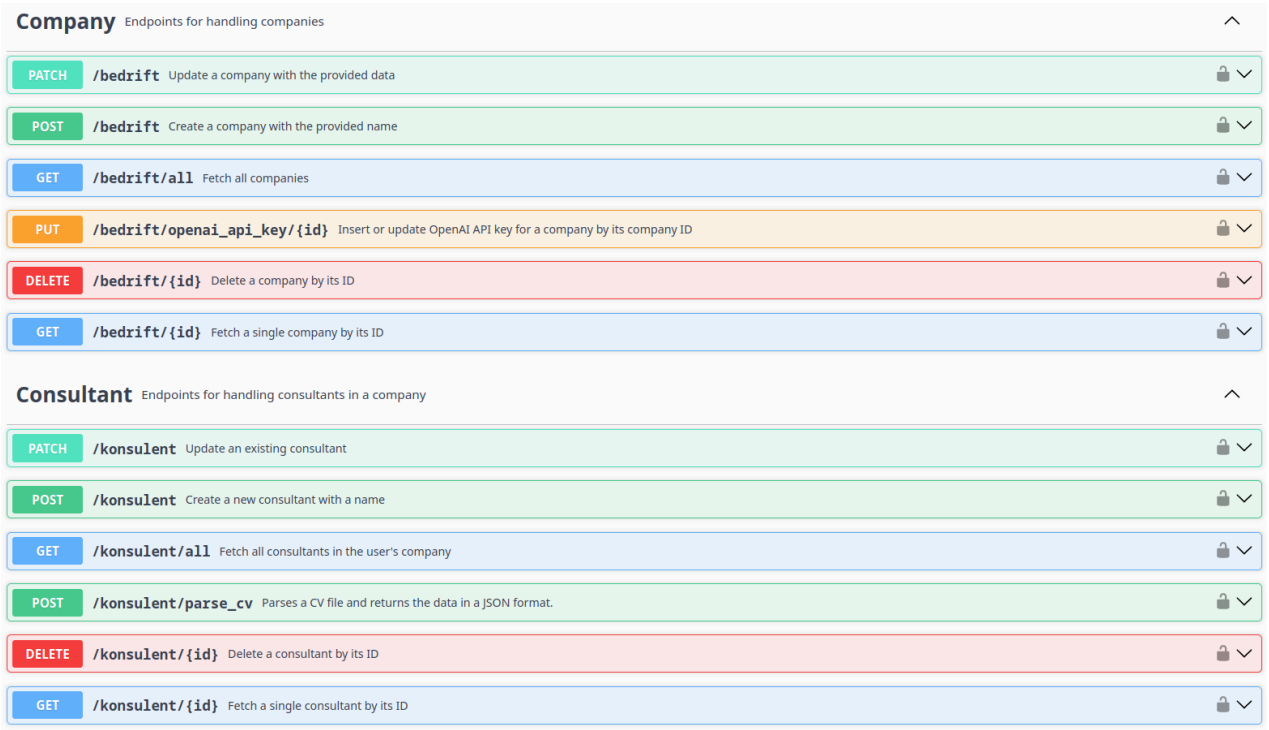
Figur 5-2 - Databasemodell med innhold i dokumentene

6 SERVER-TJENESTER

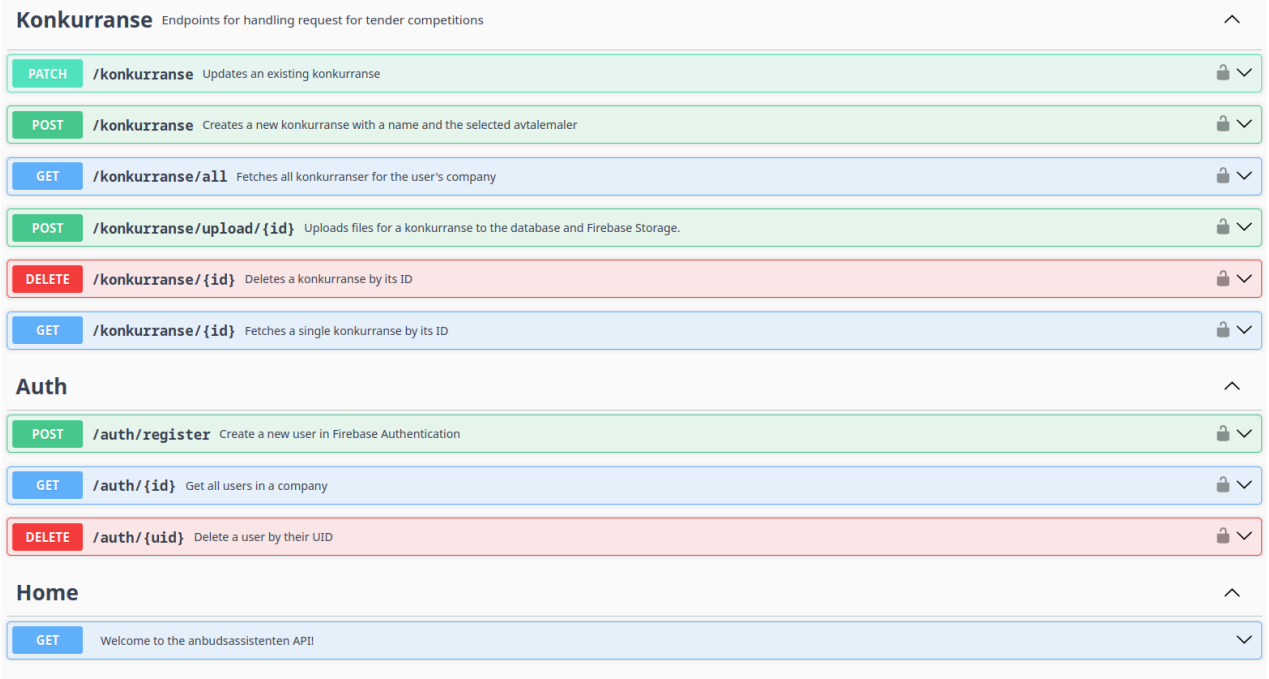
Tjeneren autogenerer OpenAPI dokumentasjon som kan vises på flere ulike formater. OpenAPI er en godt brukt standard for å vise og distribuere dokumentasjon på API-er (The Linux Foundation, 2024). Ved å bruke pakken flask-openapi3, så vil denne dokumentasjonen bli automatisk generert ut ifra definisjonen av hvert endepunkt. Ikke all info kan hentes fra endepunktet direkte, så noe må defineres for hånd, for eksempel formatet data blir returnert på og hvilke HTTP statuskoder som kan oppstå. Figur 6-1 til 6-3 viser en oversikt over alle endepunktene med HTTP metode, URL til endepunktet og en kort beskrivelse. Ved å trykke på en, får man en detaljert visning som vist i figur 6-4, med eksempler på bruk og informasjon om mulige responser. Figur 6-5 viser en del av typeoversikten, som viser hvordan typene er bygd opp og aktuelle verdier.



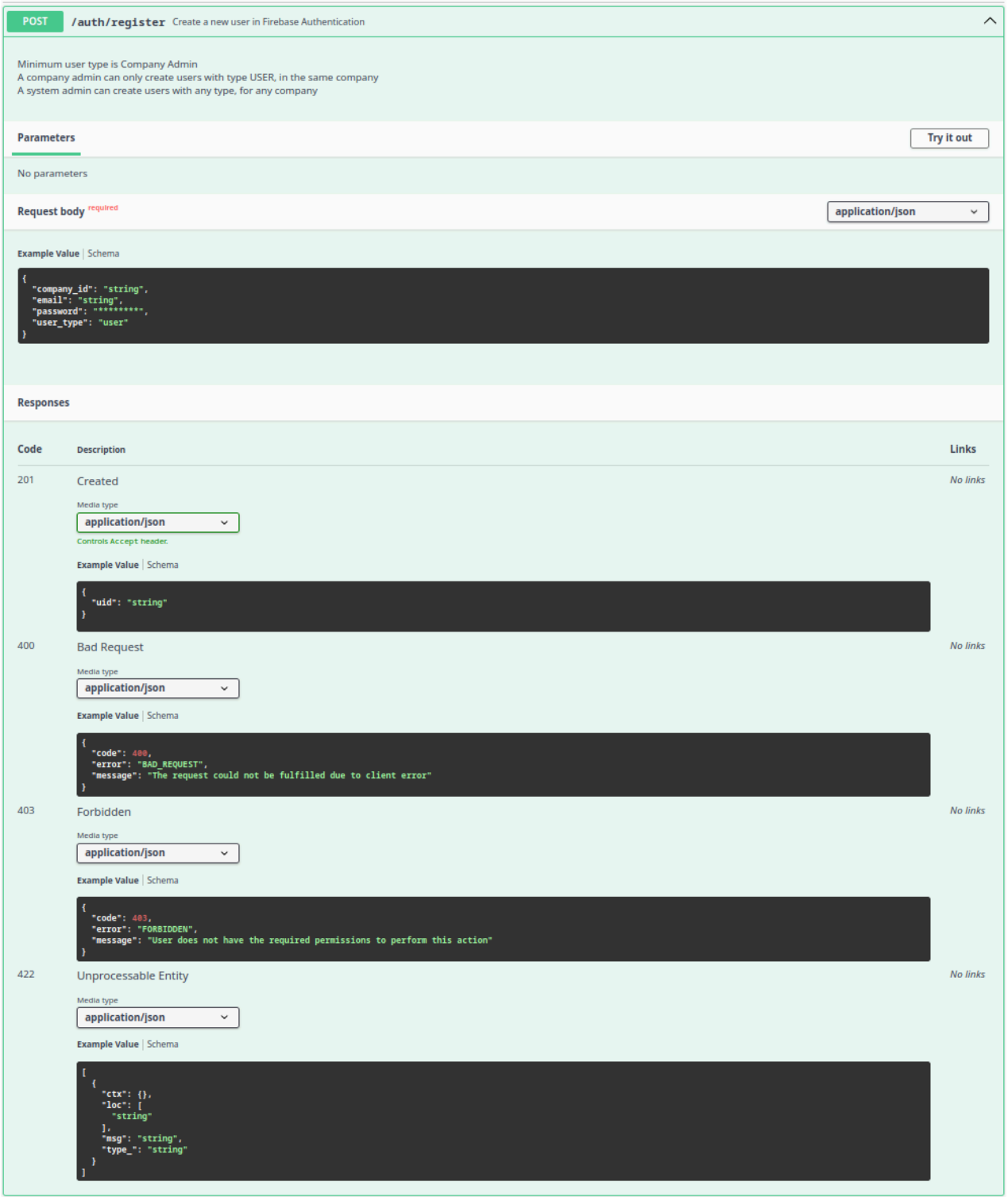
Figur 6-1 - OpenAPI Swagger dokumentasjon del 1



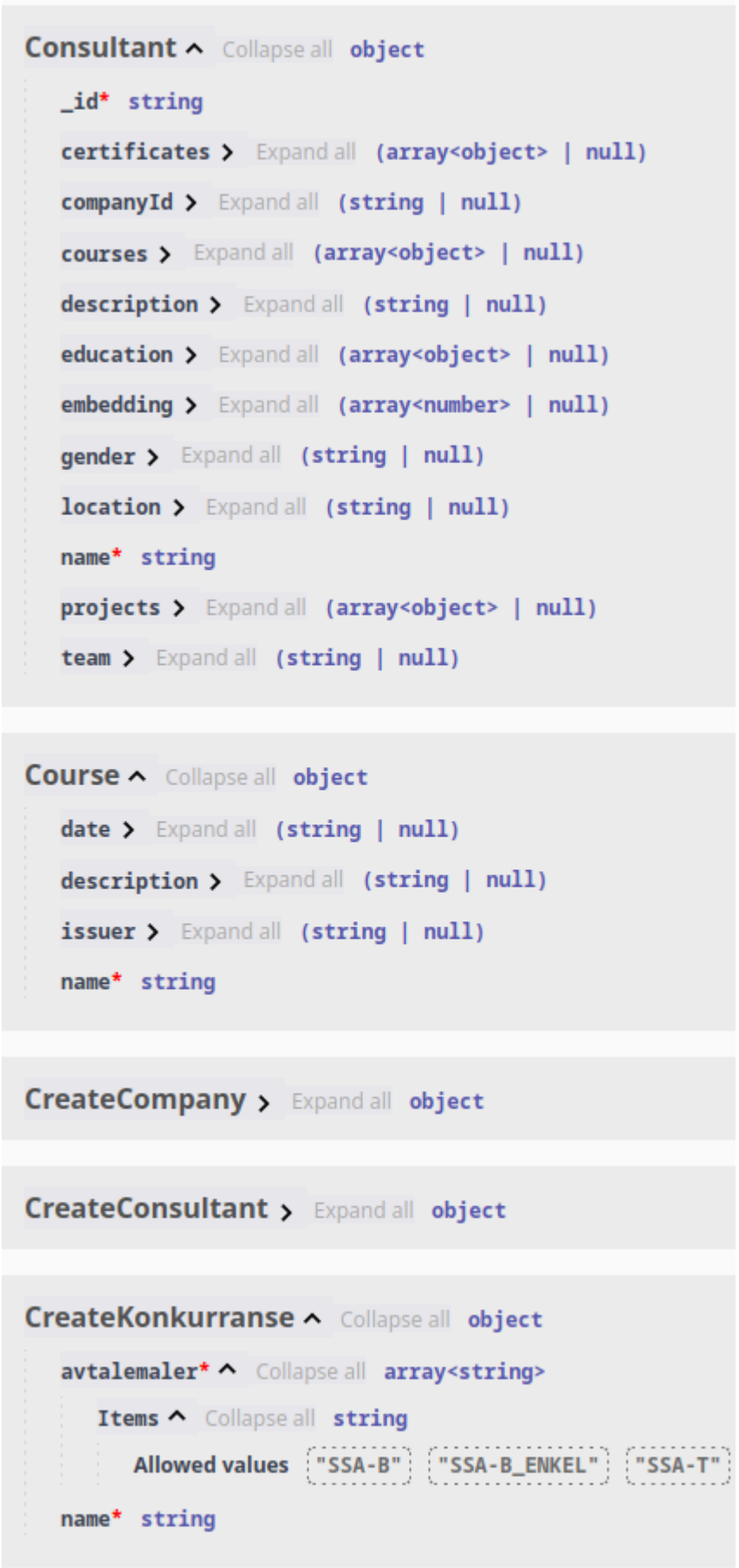
Figur 6-2 - OpenAPI Swagger dokumentasjon del 2



Figur 6-3 - OpenAPI Swagger dokumentasjon del 3



Figur 6-4 - OpenAPI Swagger detaljert visning for et endepunkt



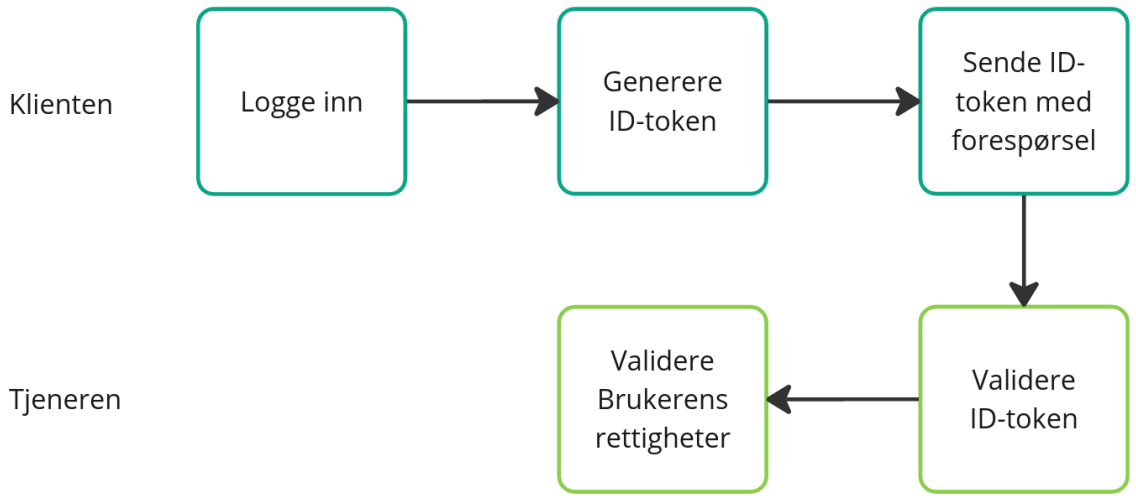
Figur 6-5 - Eksempel på datatype oversikt

7 SIKKERHET

7.1 Brukertilgang

Løsningen benytter Firebase Authentication for brukertilgang, hvor man må være innlogget for å kunne nå alle sidene med unntak av forsiden. Endepunktene er også sikret, slik at man enten må være innlogget fra klienten eller generere en ID-token for å bruke dem. Firebase Authentication håndterer alt av brukertilganger for oss, slik at man ikke trenger å tenke på hashing eller salting av passord, man behøver bare å opprette en bruker med ren tekst. Når brukeren forsøker å logge inn, vil det kalles en metode som validerer om e-post og passord er riktig, dersom det er det så vil brukeren bli innlogget. Brukerdata blir lagret i en egen database til Firebase Authentication.

Validering av bruker fungerer ved at man logger inn med sin egen bruker på klienten. Fra denne brukeren blir det generert en unik ID-token, som har en levetid på en time. Denne ID-token blir sendt kryptert til tjeneren over HTTPS og deretter validert. Dersom den er gyldig vil man kontrollere om brukeren har tilgang til den ressursen som blir etterspurt, dette blir gjort med *custom claims*, som er en måte å legge til ekstra data til en bruker, uten å ta i bruk en ekstern database. Brukerne har tre *custom claims*, brukerrolle, bedrift-id og en krypteringsnøkkel for bedriftshemmeligheter. Denne prosessen er vist i figur 7-1. For å unngå at brukere prøver å nå ressurser de ikke har tilgang til, så blir disse skjult. Slik at kun de som har tilgang til en side, ser lenken som går til den siden. Dersom en bruker som ikke har tilgang manuelt skriver inn denne lenken i URL-en, vil brukeren blir videresendt til landingssiden med en feilmelding.



Figur 7-1 - Brukerhåndtering

7.2 Cross-site Scripting (XSS)

Cross-site Scripting (XSS) er når en aktør infiserer en applikasjon med sin egen kode. Vue.js har innebygd beskyttelse mot dette i de mest vanlige tilfellene, hvor man setter inn variabler i koden (Vue, 2024).

For eksempel dersom man prøver å sette inn kode i *userProvidedString*:

```
`<h1>{{ userProvidedString }}</h1>`
```

Så får man:

```
&lt;script&gt;alert(&quot;hi&quot;)&lt;/script&gt;
```

Det Vue ikke sikrer, er tilfeller hvor man setter inn HTML direkte i koden. Det har vært diskutert mulighet for å generere Markdown fra GPT-modellene for å sette dette inn i koden, dette kan være et problem siden Markdown må konverteres til HTML, som kan inneholde en script-tag som kan kjøre kode. Dette er ikke aktuelt foreløpig. Dersom det blir det, må dette sikres ved å rense Markdown før innsetting.

7.3 Bruk av hemmeligheter

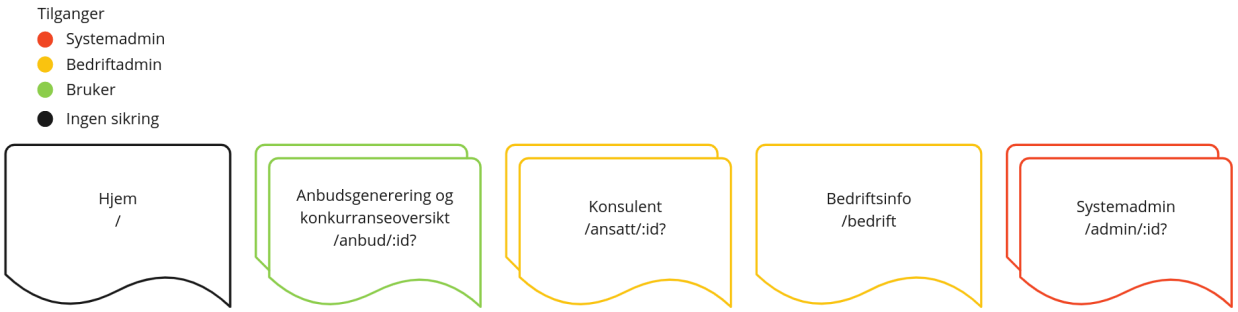
Siden prosjektet bruker API nøkler som skal holdes hemmelige, så må disse ikke lagres i versjonskontroll. Det er gjort ved å ignorere disse filene fra git, slik at de kun ligger lagret på brukerens maskin. Dette krever litt ekstra arbeid når man setter opp prosjektet, da man må skaffe disse og legge dem inn selv, men det er en sikrere løsning.

7.4 Lagring av API-nøkler i database

Hver bedrift skal ha sin egen API-nøkkel for OpenAI som blir laget og lagt inn av systemadmin. Hver bedrift skal bruke sin egen API-nøkkel når tjenersiden gjør API-kall til OpenAI. For å gjøre dette må API-nøkkel lagres i en database, slik at det er mulig å hente det ved behov for bruk på kun tjenersiden. Denne nøkkelen må dermed krypteres slik at de som har tilgang til database ikke har tilgang til disse nøklene, og det skal være mulig for alle ansatte i en bedrift å dekryptere denne ved forespørsler som krever API-kall. Dette er gjort ved å kryptere hver nøkkel som blir lastet opp, med en tilfeldig fernet krypteringsnøkkel. Denne nøkkelen blir deretter lagret sammen med hver bruker i bedriften, slik at alle brukerne har tilgang til å dekryptere API-nøkkelen.

8 RUTEKART

Figur 8-1 viser en enkel oversikt over de ulike rutene som er tilgjengelig i nettportalen. Rutene har ulike tilganger, slik at man må minst være en bruker av samme type eller høyere for å få tilgang. Den eneste ruten som ikke er sikret er landingssiden, hvor innlogging skjer. Det er også dit brukeren blir sendt dersom den prøver å nå en side man ikke har tilgang til. Siste linje definerer ruten til ressursen, hvor :id? definerer en valgfri variabel som kan settes inn. Dette gjør sidene dynamisk, da man kan vise innhold basert på hva som er fylt inn i URL-en.



Figur 8-1 - Rutekart for klienten

9 INSTALLASJON OG KJØRING

9.1 Tjener

9.1.1 Forutsetninger

- Python 3.11.6
- Pip
- Tesseract (med norsk språkpakke)
- En MongoDB Atlas cluster eller dedikert MongoDB instans.
- Firebase-prosjekt med Firebase Authentication og Cloud Storage.
- OpenAI API key.

9.1.2 Miljøvariabler og vektor søk indeks

OpenAI og MongoDB

Opprett en fil kalt .env i rotmappen til prosjektet og legg til følgende linjer i .env:

```
OPENAI_API_KEY='<din OpenAI API key>'
MONGODB_URL='<din MongoDB URL>'
```

Se filen .env.example for et eksempel.

Firebase

For å få kontakt med Firebase kreves en service account key, som er en JSON fil med autentiseringsinformasjon. Denne filen må legges i mappen src/main/resources til prosjektet med navnet service_account_key.json. Denne kan genereres i Firebase Console eller hentes fra en annen utvikler.

MongoDB Atlas

Siden applikasjonen bruker MongoDB Atlas med vektorsøk, må enkelte indekser opprettes. Dette må gjøres manuelt i Web-UI til MongoDB. Før en indeks kan defineres, må samlingen eksistere.

For consultants samlingen:

```
{
  "fields": [
        {
              "type": "vector",
              "path": "consultant_index",
              "numDimensions": 1536,
```

```
            "similarity": "cosine"
        },
        {
            "type": "filter",
            "path": "companyId"
        }
 ]
}
```

For konkurranseChunks samlingen:

```
{
 "fields": [
        {
            "type": "vector",
            "path": "konkurranse_chunk_index",
            "numDimensions": 1536,
            "similarity": "cosine"
        },
        {
            "type": "filter",
            "path": "konkurranseId"
        },
        {
            “type”: “filter”,
            “path”: “filename”
        },
 ]
}
```

9.1.3 Installasjon og kjøring

Last ned alle avhengigheter

Første gang prosjektet kjøres må alle avhengigheter lastes ned. Dette kan enkelt gjøres ved å kjøre følgende kommando:

`pip install -r requirements.txt`

Se requirements.txt for en liste over alle avhengigheter.

Starte utviklingtjener

Linux/macOS/Windows:

`python3 main.py`

Utviklingtjeneren vil nå kjøre på http://localhost:8000.

Starte lokal produksjontjener

Linux/macOS:

`ENV=prod python3 main.py`

Windows:

`set ENV=prod & python3 main.py`

Produksjontjener vil nå kjøre på http://localhost:8000.

Starte WSGI server med Gunicorn

Linux/macOS:

```
ENV=prod gunicorn main:app
```

Windows:

```
set ENV=prod & gunicorn main:app
```

Testing

For å kjøre tester, kjør følgende script:

`sh test.sh`

For å kjøre med coverage, kjør scriptet med `--coverage` flagget:

`sh test.sh --coverage`

Dette krever at coverage er installert.

`python3 -m pip install coverage`

9.2 Klient

9.2.1 Forutsetninger

- Node.js

9.2.2 Installasjon

Last ned alle pakker med:

`npm install`

9.2.3 Oppsett av prettier

VS Code

Installer “Prettier extension” for VS Code. Sett opp Prettier i VS Code settings.json.

```
"editor.defaultFormatter": "esbenp.prettier-vscode",
"editor.formatOnSave": true
```

IntelliJ IDEA

1. Åpne setting -> Languages & Frameworks -> JavaScript -> Prettier
2. Velg "Automatic prettier configuration"
3. Kryss av for "format on save"

9.2.4 Starte utviklingtjener

`npm run dev`

9.2.5 Bygge for produksjon

`npm run build`

9.2.6 Kjøre tester

`npm run test:unit`

10 DOKUMENTASJON AV KILDEKODE

10.1 Tjener

Tjeneren bruker Python Docstring for å dokumentere klasser, metoder, funksjoner og lignende. Python Docstring gjør det lett å dokumentere på en organisert måte, hvor dokumentasjonen følger det som er dokumentert. Det vil si at dersom man bruker en IDE, så kan man slå opp dokumentasjonen på enhver funksjon og variabel man bruker ved å f.eks. holde musepekeren over. Figur 10-1 viser et eksempel på dokumentasjon for en metode hvor """ definerer starten og slutten av dokumentasjonen. Inne i dokumentasjonen kan man definere parametere og returverdier for funksjonen, som gjør det lettere å lese dokumentasjonen.

```python
def get(self, object_id: ObjectId, projection: dict[str, bool] = None) -> dict[str, any] | None:
    """
    Get a document from a collection by its ID.
    :param object_id: The ID of the document
    :param projection: Used to define which fields to include or exclude from the result. An empty projection will return all fields
    :return: The document if found, otherwise None. The ID of the document will be serialized to a string. Embeddings will be excluded.
    """
```

Figur 10-1 - Python Docstring for en metode

10.2 Klient

Klienten bruker TS Doc som minner veldig mye om Python Docstring, bare at det er for TypeScript. Det virker på samme måte, men syntaksen er litt annerledes. Figur 10-2 viser et eksempel på TS Doc for en funksjon, hvor /** definerer starten av dokumentasjonen og */ definerer slutten. TypeScript kan definere parametere og returverdier i dokumentasjonen på samme måte som Python.

```typescript
/**
 * Fetch a single konkurranse by its id.
 * @param konkurranseId The id of the konkurranse to fetch.
 * @returns The konkurranse if it exists, otherwise null.
 */
export async function getKonkurranse(konkurranseId: ObjectId): Promise<Konkurranse | null> {
  return await request( path: `/konkurranse/${konkurranseId}`)
}
```

Figur 10-2 - TS Doc for en funksjon

10.3 API-dokumentasjon

REST-API er dokumentert ved hjelp av OpenAPI dokumentasjon, vist i kapittel 6.

10.4 Readme

Klient og tjener har hver sin fil kalt README.md, som inneholder informasjon om applikasjonene. Det er blant annet informasjon om hvordan man setter opp applikasjonene, bygger og kjører dem. Disse filene skal være lett å forstå, slik at noen skal kunne følge innholdet i filen for å sette opp og kjøre applikasjonene. Disse kan også inneholde annen nyttig informasjon, som for eksempel mappestruktur.

11 KONTINUERLIG INTEGRASJON OG TESTING

11.1 CI-CD

Både klienten og tjeneren er satt opp med GitHub Actions skript, som kjører når man pusher kode til hovedgrenen eller lager en pull-forespørsel. Siden klienten og tjeneren er adskilt i to ulike oppbevaringssteder, så vil de kjøre uavhengig av hverandre. Formålet med disse er det samme, man ønsker å teste at det bygges, og hvis det bygges så kjører alle testene, dersom det er bygget og alle testene ble godkjent så blir skriptene godkjent. For å gjøre det enkelt så kjører disse skriptene på GitHub sine egne tjenere, dette krever ingen oppsett og har en generøs gratis plan for bruk.

Figur 11-1 viser skriptet for tjeneren, Figur 11-2 og 11-3 for klienten med bygg og deploy jobber. Skriptene for klient og tjener er satt opp på samme måte, de bare jobber mot ulike teknologier og klienten har en ekstra jobb. De er satt opp til å kjøre når man enten gjør en “git push” til hovedgrenen eller dersom man lager en “pull-request” mot hovedgrenen. Disse skriptene vil deretter gjøre:

1. Hente filene med “checkout”
2. Sette opp miljøet
3. Laste ned avhengigheter
4. Cache resultatet for å unngå å laste ned de samme hver gang
5. Kjøre tester

11.1.1 Distribuering

For å kunne lettere gjøre applikasjonen klar for testing, var det ønskelig å lansere tjenesten på et domene slik at testere slipper å sette opp løsningen på egen maskin eller låne. For å løse dette fikk gruppen tilgang til en DigitalOcean droplet av Link Utvikling, som applikasjonen kan hostes på. For å håndtere HTTP forespørsler er tjenesten Caddy brukt, hvor man kan definere hvilket innhold som vises basert på hvilken URL som mottas (Server, 2024). Man kan også definere reverse proxy for å sende brukeren til en annen tjeneste og automatisk oppgradere forespørsler til HTTPS med sertifikat fra Let’s Encrypt.

Klient

For å distribuere klienten er det lagt til en ekstra jobb i GitHub Actions skriptet, som kjører etter bygging. Denne jobben kjører alltid etter bygg-jobben og henter filene som ble generert etter bygging, og flytter disse til en egen branch. Det gjør at man har en egen branch som inneholder kun de ferdige byggfilene som skal sendes til klienten. Denne jobben blir kun kjørt dersom bygg-jobben blir fullført, på denne måten sikrer man at applikasjonen virker før man oppdaterer.

På DigitalOcean-tjeneren er git oppbevaringsstedet til klienten med byggfilene satt opp, slik at man kan bruke kommandoen `git pull` for å hente oppdateringer. Det er dermed satt opp en CronJob som kjører denne kommandoen hvert 5. minutt, slik at oppdateringer blir reflektert relativt kjapt (Kubernetes, 2024). Dette er en veldig enkel løsning, men fører til en del ekstra ressursbruk.

Tjener

For å distribuere tjeneren er det litt mer jobb, da applikasjonen skal kjøre lokalt på tjeneren og krever diverse pakker og skript for å virke som det skal. Først og fremst må det settes opp en WSGI tjener, som er et grensesnitt mot applikasjonen som er laget. Flask rammeverket inneholder en egen tjener, men denne er ment kun for utvikling og er ikke stabil nok for bruk i produksjon. For applikasjonen er det bruk Gunicorn, da den har god støtte og er lett å bruke. Tjeneren blir kjørt i sin egen Systemd tjeneste, slik at det er lett å håndtere oppstart og kjøring av den.

På lik måte som klienten er det satt opp en CronJob som kjører hvert 5. minutt for å sjekke mot endringer. Men i stedet for å bare hente oppdatert kode, så vil skriptet først sjekke om det er endringer, og kun om det er endringer vil koden hentes. Etter at koden er hentet må eventuelle nye pakker eller oppgraderinger lastes ned, før Gunicorn tjeneren startes på nytt, og nye endringer blir reflektert i produksjon. Det er kun endringer som ligger i hovedgrenen som vil hentes, det er dermed fornuftig at bygging og testing blir utført før man henter koden, gjerne ved å bruke pull-requester.

```yaml
name: Python Application

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

permissions:
  contents: read

jobs:
  test:

    runs-on: ubuntu-latest
    timeout-minutes: 30

    steps:
      - uses: actions/checkout@v4
      - name: Set up Python 3.11
        uses: actions/setup-python@v5
        with:
          python-version: "3.11"
          cache: "pip"
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
      - name: Test with unittest
        run: |
          sh test.sh
```

Figur 11-1 - GitHub Actions script for tjeneren

```yaml
name: Node.js CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4
      - name: Use Node.js LTS
        uses: actions/setup-node@v4
        with:
          node-version: 20.x
          cache: 'npm'
      - run: npm ci
      - run: npm run build --if-present
      - run: npm run test:unit

      - name: Upload production-ready build files
        uses: actions/upload-artifact@v4
        with:
          name: production-files
          path: ./dist
```

Figur 11-2 - GitHub Actions script for klienten (build)

```yaml
deploy:
  name: Deploy
  needs: build
  runs-on: ubuntu-latest
  if: github.ref == 'refs/heads/main'

  steps:
    - name: Download artifact
      uses: actions/download-artifact@v4
      with:
        name: production-files
        path: ./dist

    # Deploy to local repo
    - name: Deploy
      uses: s0/git-publish-subdir-action@develop
      env:
        REPO: self
        BRANCH: build
        FOLDER: dist
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

Figur 11-2 - GitHub Actions script for klienten (deploy)

11.2 Enhetstesting

Siden tjenersiden inneholder det meste av logikken til applikasjonen, så er det også der det har vært fokus på å skrive tester. Applikasjonen bruker enhetstester, men ingen integrasjonstesting. I tilfeller hvor det gjøres API-kall til OpenAI, blir data mocket med unittest mocking funksjonene (Unittest, 2024), slik at det er mulig å teste resten av logikken i metodene. Mens ved testing av MongoDB, så brukes mockingbibliotektet *Mongomock,* som gir oss en falsk MongoDB database som man kan teste mot, uten å definere returverdier eller andre ting. Testene skal teste alle mulige resultater av de ulike metodene, som flere mulige feiltilfeller og hvis alt er riktig brukt. Testene på tjenersiden kan kjøres ved å kjøre shell-scriptet "test.sh", som vil oppdage alle testene og kjøre dem.

De fleste klassene og funksjonene har tester, men de som mangler inneholder veldig mange API-kall, slik at det blir unødvendig å teste dem, da all data må mockes uansett. Dette gjelder da Chains klassene, som gjør API-kall til OpenAI API-et, ofte flere asynkront.

For å sikre at testene dekker mest mulig, ønsker man en testdekning på minst 80% i tjeneren. Det er ikke alt som kan testes direkte uten mocking, så det vil ikke være nødvendig å dekke alt. Da kan man i noen tilfeller ende opp med å teste mock resultatet i stedet for faktisk kode.

Ved innlevering av prosjektet hadde tjenersiden en testdekning på ~83%.

12 REFERANSER

Kubernetes (2024) *CronJob*. Tilgjengelig fra: https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/ (Hentet: 25 April 2024).

Server, C.W. (2024) *Caddy - The Ultimate Server with Automatic HTTPS*, *Caddy Web Server*. Tilgjengelig fra: https://caddyserver.com/ (Hentet: 25 April 2024).

The Linux Foundation (2024) *Home 2024*, *OpenAPI Initiative*. Tilgjengelig fra: https://www.openapis.org/ (Hentet: 11 April 2024).

Unittest (2024) *unittest.mock — mock object library*, *Python documentation*. Tilgjengelig fra: https://docs.python.org/3/library/unittest.mock.html (Hentet: 30 Mars 2024).

Vue (2024) *Vue.js | Security*. Tilgjengelig fra: https://vuejs.org/guide/best-practices/security (Hentet: 30 Mars 2024).