Høgskulen på Vestlandet

Fakultet for ingeniør- og naturvitskap

Institutt for datateknologi, elektroteknologi og realfag

QR-koder — styling, generering og innsikt i hva som påvirker lesbarheten

Systemdokumentasjon

Versjon 4.0

Dokumentet er basert på Systemdokumentasjon utarbeidet ved NTNU. Revisjon og tilpasninger til bruk ved IDER, DATA-INF utført av Carsten Gunnar Helgesen, Svein-Ivar Lillehaug og Per Christian Engdal. Dokumentet finnes også i engelsk utgave.

REVISJONSHISTORIE

Dato	Versjon	Beskrivelse	Forfatter
22/03/22	1.0	Implementere innledning og arkitektur	Even Sleire
23/04/22	2.0	Sikkerhet og servertjeneste	Even Sleire
01/05/22	3.0	Dokumentasjon og testing	Even Sleire, Frede Berdal
21/05/22	4.0	Endelig revisjon	Even Sleire, Magnus Gjøsund, Frede Berdal

INNHOLDSFORTEGNELSE

1 INNLEDNING

Dette dokumentet har som hensikt å dokumentere og gi en oversikt over hvordan produktet er bygget opp. Dokumentet vil beskrive arkitekturen for systemet, samt hva som kreves for å kunne starte opp eller integrere mot systemet.

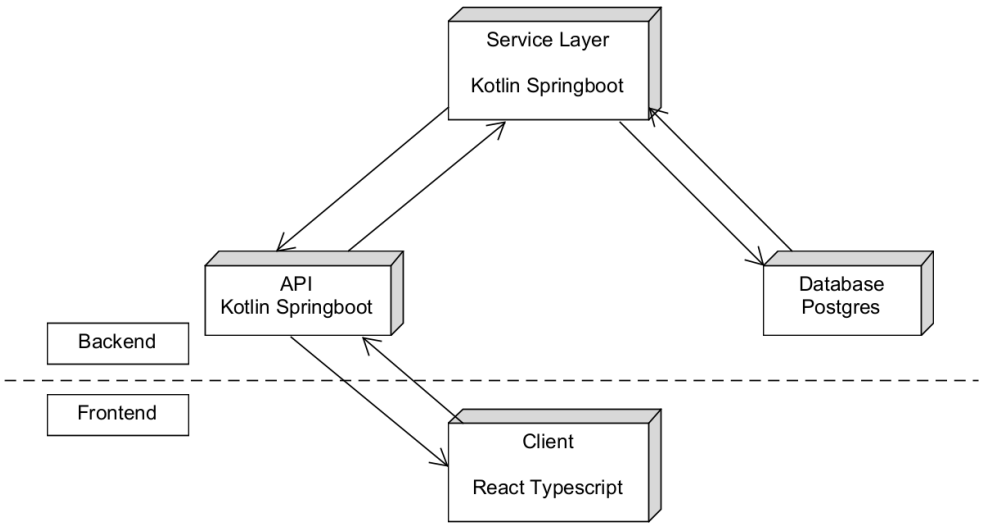
2 ARKITEKTUR

Dette kapittelet tar for seg arkitekturen til systemet. Det skal ta for seg overordnet arkitektur og lagdelt arkitektur.

2.1 Overordnet arkitektonisk skisse av systemet

Systemet vil fungere som en webapplikasjon. Serversiden benytter seg av teknologien Kotlin sammen med rammeverket Spring Boot. Den tar også i bruk en PostgreSQL database for lagring av data. Klientsiden er bygget opp av React Typescript.

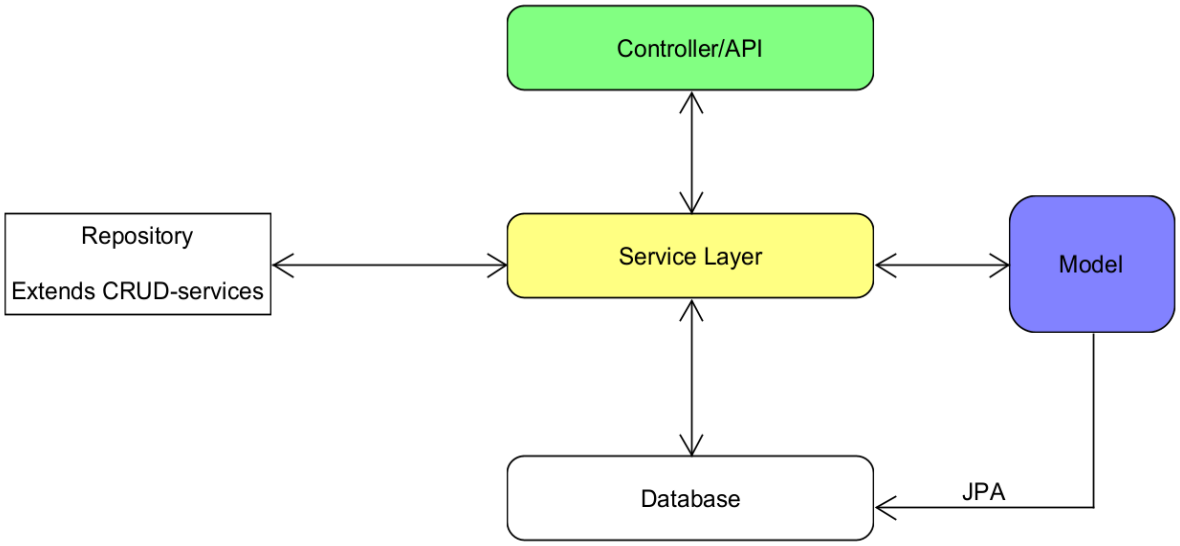
Figur 2.1 demonstrerer flyten i systemet. Klienten sender forespørsler til API-endepunkter og får en respons fra serveren. API-et vil lytte etter forespørsler fra klientsiden. Dersom et endepunkt treffes vil service-laget bli kalt på. Service-laget har ansvar for håndtering av forretningslogikken. Dersom det er behov for ressurser fra databasen vil repository-laget bli kalt på for uthenting og innsettelse av data. Service-laget vil så returnere nødvendig data til API-et som sender en respons til klienten.



Figur 2.1. Overordnet arkitektonisk skisse av systemet. Flyten starter og slutter i klienten.

2.2 Lagdelt arkitektur for serversiden

I dette kapittelet vil den lagdelte arkitekturen bli presentert (se Figur 2.2). Databasetabeller vil bli opprettet ved bruk av JPA (Java Persistence API). Service-laget vil benytte seg av en repository-klasse som utvider CRUD-grensesnittet (Create-Read-Update-Delete). Grensesnittet tillater å gjøre databasespørringer på et høyere abstraksjonsnivå. Det vil si at database-kall kan utføres uten å måtte skrive spørringene. Grensesnittet krever objektet og den riktig primitive datatypen som skal benyttes som ID (se Figur 2.3).



Figur 2.2. Lagdelt arkitektur for serversiden

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S entity);

    <S extends T> Iterable<S> saveAll(Iterable<S> entities);

    Optional<T> findById(ID id);

    boolean existsById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> ids);

    long count();

    void deleteById(ID id);

    void delete(T entity);

    void deleteAllById(Iterable<? extends ID> ids);

    void deleteAll(Iterable<? extends T> entities);

    void deleteAll();
}
```

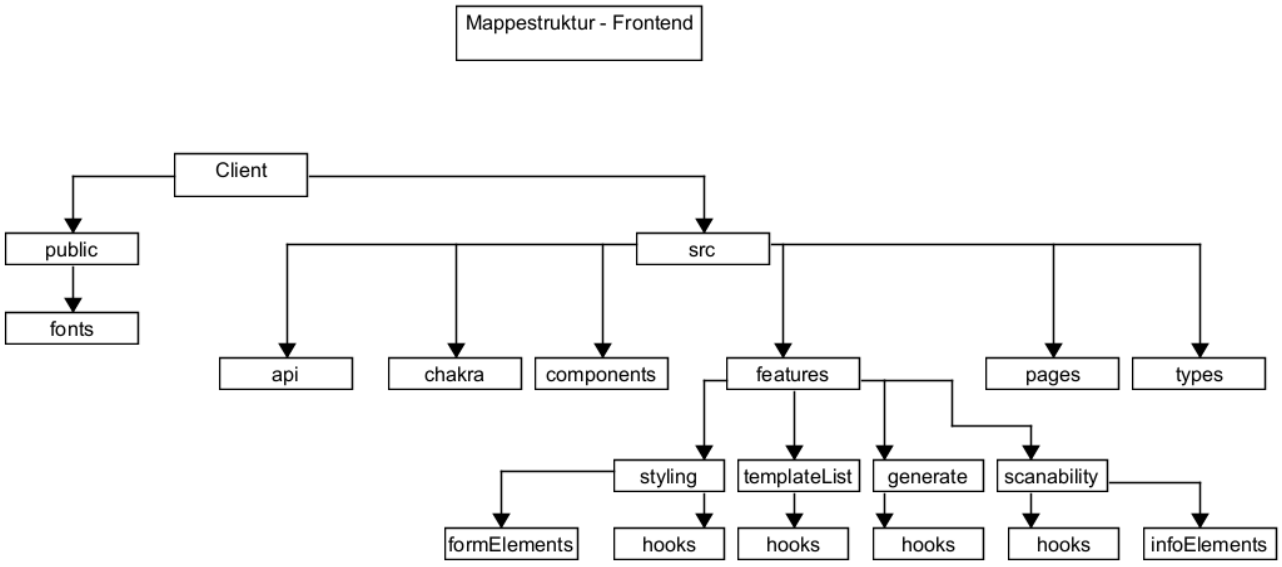
Figur 2.3. CrudRepository interface

3 PROSJEKTSTRUKTUR

Dette kapittelet tar for seg mappestrukturen i prosjektet. Prosjektet er i delt inn i to uavhengige deler bestående av en frontend og en backend.

3.1 Frontend

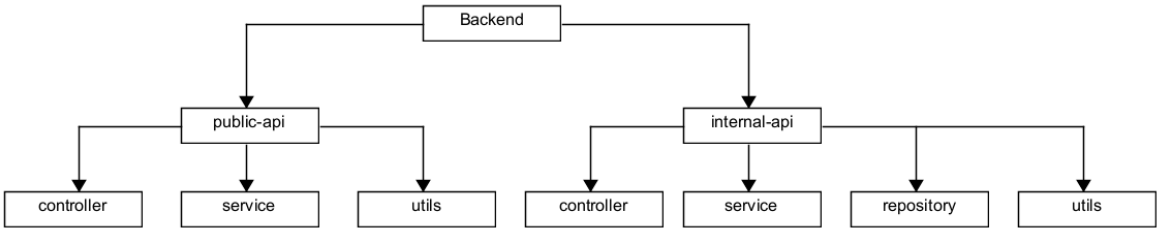
Prosjektstrukturen til frontend baserer seg på en "feature-based" inndeling (se Figur 3.1). Denne inndelingen sentrerer seg rundt hovedfunksjonalitetene til applikasjonen. Disse vil ligge i mappen *features*. I *components* mappen vil alle globale komponenter ligge. *Features* inneholder mapper for komponenter og funksjonalitet som er isolert. *Pages* mappen vil inneholde alle sidene i brukergrensesnittet. Disse sidene vil bestå av globale komponenter som ligger i *components* og komponenter fra *features* mappen. Funksjonalitet for kommunikasjon med API vil ligge i mappen *api*.



Figur 3.1. Mappestrukturen til frontend

3.2 Backend

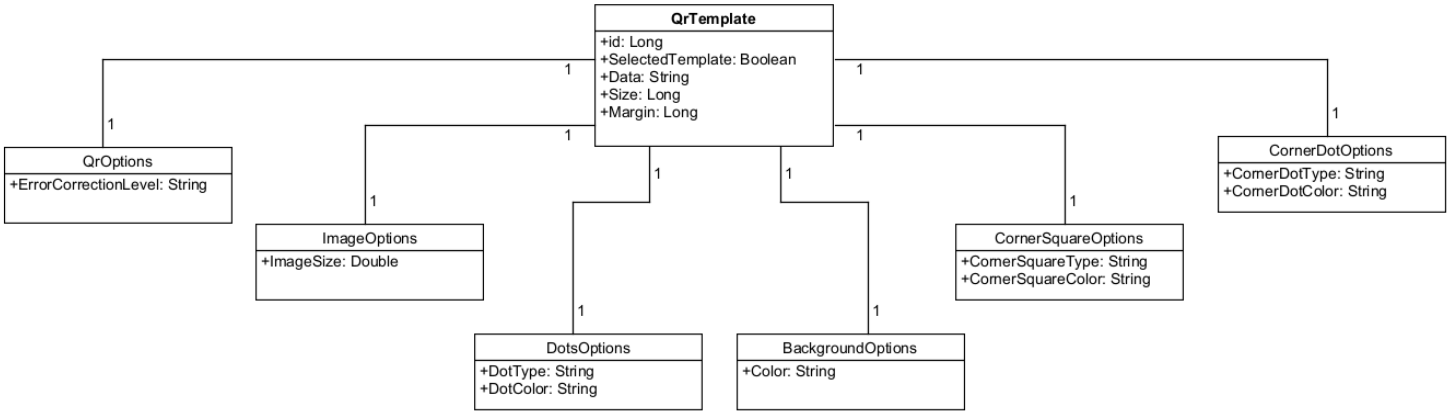
Prosjektstrukturen for backend vil være delt opp i to hoveddeler (se Figur 3.2). En del for public-API og en del for internal-API. Hensikten med dette er å lage et API som kun gir tilgang til relevante ressurser for eksterne klienter, og et API som tilhører klienten til systemet. Dette gir mer fleksibilitet ved innføring av ny funksjonalitet uten å påvirke API-et til eksterne klienter.



Figur 3.2. Mappestrukturen til backend

4 KLASSEDIAGRAM

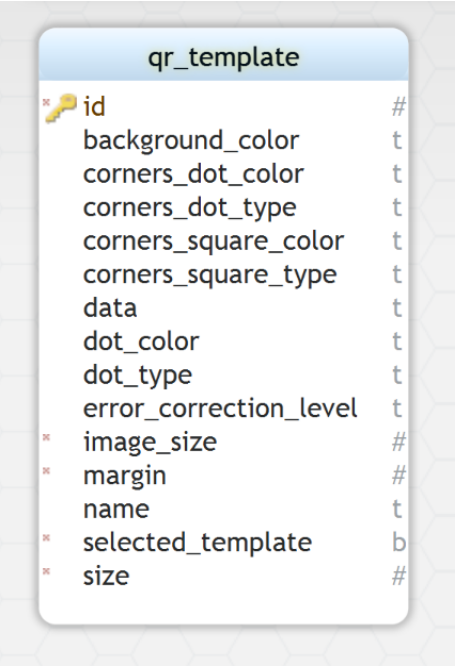
Systemets klassediagram (se Figur 4.1). En stylingmal består av flere én-til-én relasjoner. Stylingmalen må ha ett tilfelle av objektene QrOptions, DotsOptions, CornerDotOptions, CornerSquareOptions og ImageOptions. I tillegg må alle disse være del av én stylingmal.



Figur 4.1. Klassediagram

5 DATABASEMODELL

Prototypen vil kun ha behov for å lagre egenskapene til stylingmalene. Objektene blir flatet ut før de lagres i databasen (se Figur 5.1). Databasen vil bestå av én tabell. Primary key for hver entitet vil bli autogenerert ved innsettelse.



Figur 5.1. Databasemodell

6 SERVER-TJENESTER

Serversiden er delt opp i to ulike bruksområder, public-API og internal-API. Begge API-ene vil bli implementert som REST API. Dette er API som må følge visse krav til utforming (IBM, 2021). Kravene er at klient og server skal være uavhengig av hverandre, hvert endepunkt skal ha sin egen URI (Uniform Resource Identifier) og det skal være en lagdelt arkitektur. API-ene skal i tillegg være tilstandsløse, og ressursene skal kunne lagres i hurtigminnet (cacheable).

6.1 Internal-API

TemplateController

```
@RestController
@RequestMapping(SAVED_TEMPLATES)
@CrossOrigin(origins = ["http://localhost:3000"], maxAge=3600, allowCredentials = "true")
class TemplateController {

    @Autowired
    lateinit var templateService: TemplateService

    @GetMapping(value = ["templates"])
    fun getTemplates(): Iterable<QrTemplateDto> {
        return templateService.getQrTemplates()
    }

    @GetMapping(value = ["/selected-template"])
    fun getSelectedTemplate() : QrTemplateDto {
        return templateService.getSelectedTemplate()
    }

    @DeleteMapping(value = ["/delete-template/{id}"])
    fun deleteTemplate(@PathVariable("id") id: Long) : Unit {
        return templateService.deleteQrTemplate(id)

    }

    @PutMapping(value = ["/update-selected"])
    fun updateSelectedTemplate(@RequestBody templates : List<QrTemplateDto>) : List<QrTemplateDto> {

        return templateService.updateSelectedTemplate(templates)
    }


}
```

Figur 6.1. API-endepunkter for TemplateController

/templates/v1/templates [GET]

Dette endepunktet henter alle stylingmaler som er lagret i databasen. Endepunktet vil returnere en liste med disse malene.

/templates/v1/selected-template [GET]

Endepunkt som returnerer stylingen til malen som er valgt for generering. Metoden vil returnere et objekt som inneholder alle egenskapene. I service-laget er denne ressursen deklarert som *cacheable*. Som vil si at objektet kan lagres i hurtigminnet.

/templates/v1/delete-template/{id} [DELETE]

Endepunkt for å slette en lagret mal. Klienten vil sende en forespørsel med en gitt ID. Denne ID-en brukes videre for å finne og slette objektet i databasen.

/templates/v1/update-selected [PUT]

Endepunkt for å oppdatere hvilken mal som er valgt av for generering. Dette endepunktet vil ta imot en oppdatert liste av alle malene dersom det har blitt utført endringer.

StylingController

```
@RestController
@RequestMapping(STYLING)
@CrossOrigin(origins = ["http://localhost:3000"], maxAge=3600, allowCredentials = "true")
class StylingController {

    @Autowired
    lateinit var stylingService: TemplateService


    @PostMapping(value = ["/save-template"])
    fun saveTemplate(@RequestBody qrTemplateDto: QrTemplateDto): QrTemplateDto {

        stylingService.addQrTemplate(qrTemplateDto)
        return qrTemplateDto

    }


}
```

Figur 6.2. API-endepunkter for StylingController

/styling/v1/save-template [POST]

Endepunkt for å lagre en stylet mal. Klienten sender en forespørsel til endepunktet med et objekt som inneholder alle egenskapene til malen. Objektet konverteres til ønsket struktur i service-laget før det lagres i databasen.

6.2 Public-API

API-et inneholder endepunkt for eksterne klienter som ønsker å benytte seg av funksjonaliteten til systemet.

```
@RestController
@RequestMapping(GENERATE)
@CrossOrigin(origins = ["http://localhost:3000"], maxAge=3600, allowCredentials = "true")
class QrCodeController {

    @Autowired
    lateinit var qrCodeService: QrCodeService

    @GetMapping(value = ["/selected-template"])
    fun getSelectedTemplate() : QrTemplateDto {
        return qrCodeService.getSelectedTemplate()
    }
}
```

Figur 6.3. API-endepunkter for internal-API

public/v1/selected-template [GET]

Endepunktet vil være det eneste endepunktet eksterne klienter trenger for å generere QR-koder. Endepunktet vil gjøre det samme som internal-API ved at den henter stylingen til den valgte malen.

7 SIKKERHET

Systemet vil fungere som et internt system hos oppdragsgiver. Dersom systemet blir tatt i bruk blir sikkerheten rundt innlogging håndtert gjennom oppdragsgiver sitt SSO (Single-Sign-On) system.

API

Når en kjører prototypen lokalt vil serveren og klienten kjøre på ulike servere. Ved å aktivere CORS (Cross-Origin Resource Sharing) vil det være mulig å spesifisere for API-et hvilke domener som skal få tilgang til endepunktene. Dette fører til bedre kontroll over hvem som har tilgang til ressursene på serveren.

8 INSTALLASJON OG BIBLIOTEK

Systemet benytter seg av flere bibliotek, som blir presentert i dette kapittelet. Alle bibliotekene i klienten vil bli lastet ned ved å kjøre kommandoen "*npm install"*. Dette forutsetter at en har lastet ned Node.js. Avhengigheter på serversiden er strukturert og håndtert av filen *pom.xml*. Disse avhengighetene blir lastet ned ved kjøring av programmet.

Qr-code-styling

Et Javascript bibliotek som tilbyr styling og generering av QR-koder på klientsiden.

Colorpicker

Et klientbasert bibliotek som tilbyr visuell representasjon for valg av farger. En kan redigere farger ved hjelp av RGB-komponenter eller med HSV.

React-Hook-Form

Et klientbasert bibliotek som forenkler håndtering av HTML-forms i React.

Chakra UI

CSS-bibliotek for å raskere sette opp komponenter og forenkle designprosessen.

Dokka

En plugin på serversiden som brukes for å generere dokumentasjon av koden.

Springframework

Tar for seg ulike avhengigheter som er vedlagt ved initialisering av et Spring Boot prosjekt.

8.1 Instruksjon for kjøring

Kjøring av frontend

For kjøring av frontend kreves node.js.

1. Klone prosjektet fra Github.
2. Naviger til mappen som heter *frontend.*
3. Kjør kommandoen "*npm install".*
4. Kjør kommandoen "*npm start*".
5. Klienten begynner og vil kjøre på localhost:3000.

Kjøring av backend

For kjøring av serversiden kreves bruk av en kjørende database.

1. Last ned postgreSQL.
2. Sett opp en tom lokal database.
3. Klone prosjektet fra Github.
4. Naviger til mappen som heter *backend.*
5. Naviger til filen application.properties.
6. Endre url, brukernavn og passord til din lokale database.
7. Kjør filen BackendApplication.

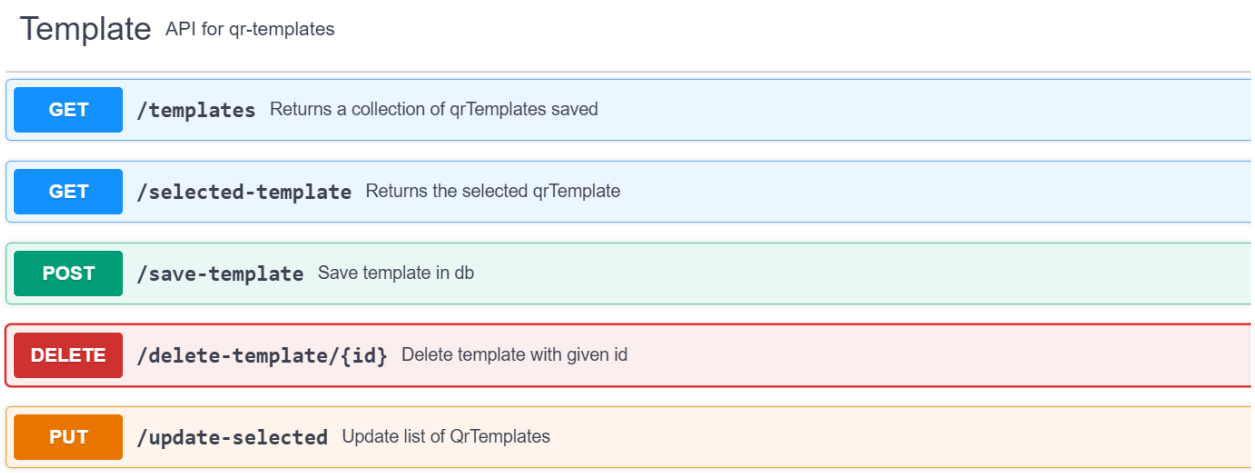
9 DOKUMENTASJON AV KILDEKODE

Dette kapittelet tar for seg dokumentasjon av API og kildekode.

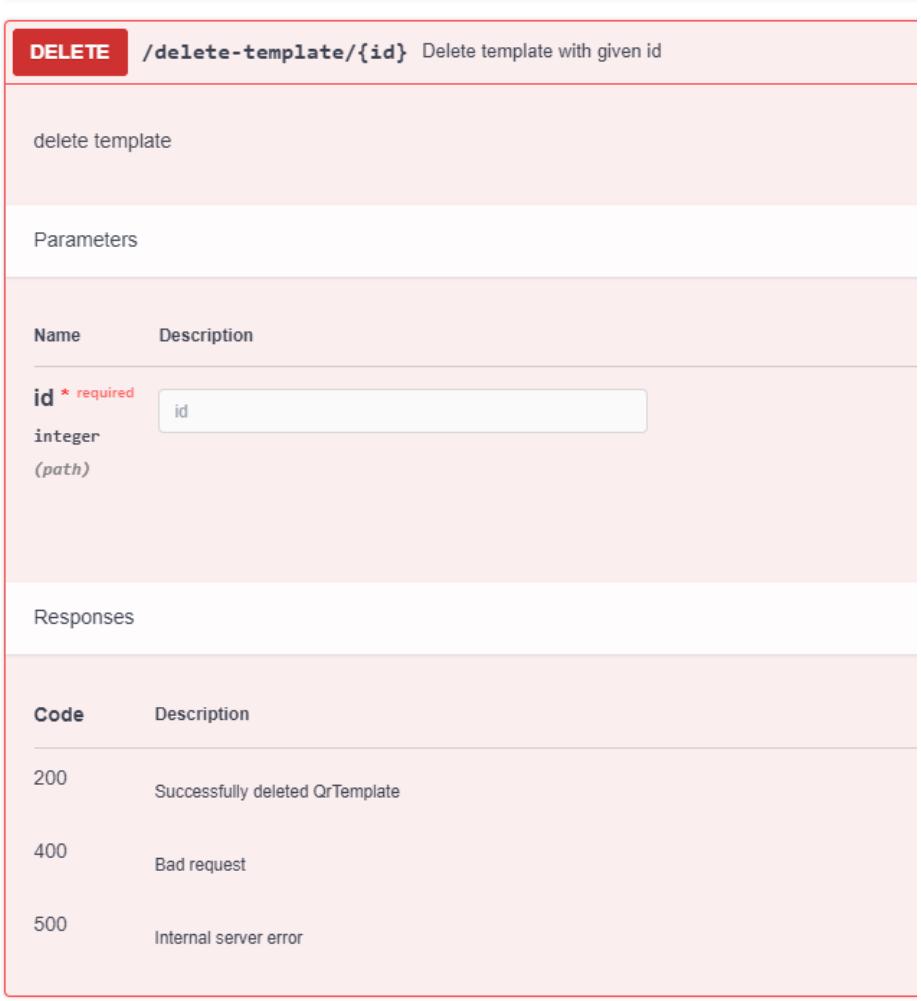
9.1 Dokumentasjon av API

API-et er dokumentert ved å bruke verktøyet Swagger. Dette er et verktøy som visualiserer dokumentasjonen på en oversiktlig måte slik at en unngår å måtte lete i kildekoden for å finne endepunktene av interesse.

Figur 9.1 viser en liste over alle endepunktene i internal-API. Dersom man trykker på en av endepunktene vil det vises en oversikt over parametere om det kreves og HTTP statuskoder og deres betydning (se Figur 9.2).



Figur 9.1. Swagger - Endepunkt



Figur 9.2. Eksempel på delete endepunkt

9.2 Dokumentasjon av kildekode

Dokumentasjon kan skrives på samme måte som Javadocs. Kotlin bruker en plugin kalt "Dokka" for generering av dokumentasjonen. Det genereres en HTML-fil med innholdet i target-mappen i prosjektet. Gruppen får ikke dette til å fungere på grunn av konfigurasjons-problemer. Figur 9.3 viser et eksempel på hvordan dokumentasjonen er skrevet.

```
   /**
     * Method for mapping a qrTemplate to a QrTemplateDto
     * @param qrTemplate
     * @return QrTemplateDto
     */
    fun mapQrTemplateToDto(qrTemplate: QrTemplate) : QrTemplateDto {
        return QrTemplateDto(
            qrTemplate.id,
            qrTemplate.selectedTemplate,
            qrTemplate.data,
            qrTemplate.size,
            qrTemplate.margin,
            QrOptions(qrTemplate.errorCorrectionLevel),
            ImageOptions(qrTemplate.imageSize),
            DotsOptions(qrTemplate.dotType, qrTemplate.dotColor),
            CornersSquareOptions(qrTemplate.cornersSquareType, qrTemplate.cornersSquareColor),
            CornersDotOptions(qrTemplate.cornersDotType, qrTemplate.cornersDotColor)
        )
    }
}
```

Figur 9.3. Eksempel på dokumentasjon av kildekode

10 TESTING

Dette kapittelet tar for seg enhetstesting, instrukser for kjøring av testene, API-testing og manuell testing.

10.1 Enhetstesting

En enhetstest er en isolert enhet av koden der en sammenligner faktisk verdi opp mot forventet verdi (SmartBear, 2022). I Java er enheten ofte en klasse. Enhetstester består av en eller flere metoder fra en klasse for å generere synlige observasjoner som verifiseres automatisk.

Utils

Utils klassen har ansvar for å konvertere data som skal sendes og mottas fra klienten. Dette er nødvendig for å lagre data med ønsket struktur og dermed også for å sende data med ønsket struktur. Testen er bygget opp ved at to ulike objekter med samme data initialiseres. Deretter konverteres det ene objektet og sjekker at de blir like.

Controller - Service - Repository

Controller-laget, service-laget og repository-laget vil alle ha lignende tester. Dette grunnes at det er tilnærmet identisk data som sendes gjennom lagene. Hvert lag vil bli testet for å identifisere i hvilket lag testene eventuelt skulle feile.

```
@SpringBootTest
internal class TemplateControllerTest {

    @Autowired
    lateinit var controller: TemplateController

    @Test
    fun `should return the whole collection of templates`() {
        // when
        val templates = controller.getTemplates()
        // then
        Assertions.assertThat(templates.count()).isGreaterThan(1)
    }

    @Test
    fun `should return the selected template`() {
        // when
        val template = controller.getSelectedTemplate()
        // then
        Assertions.assertThat(template.selectedTemplate).isEqualTo(true)
    }
}
```

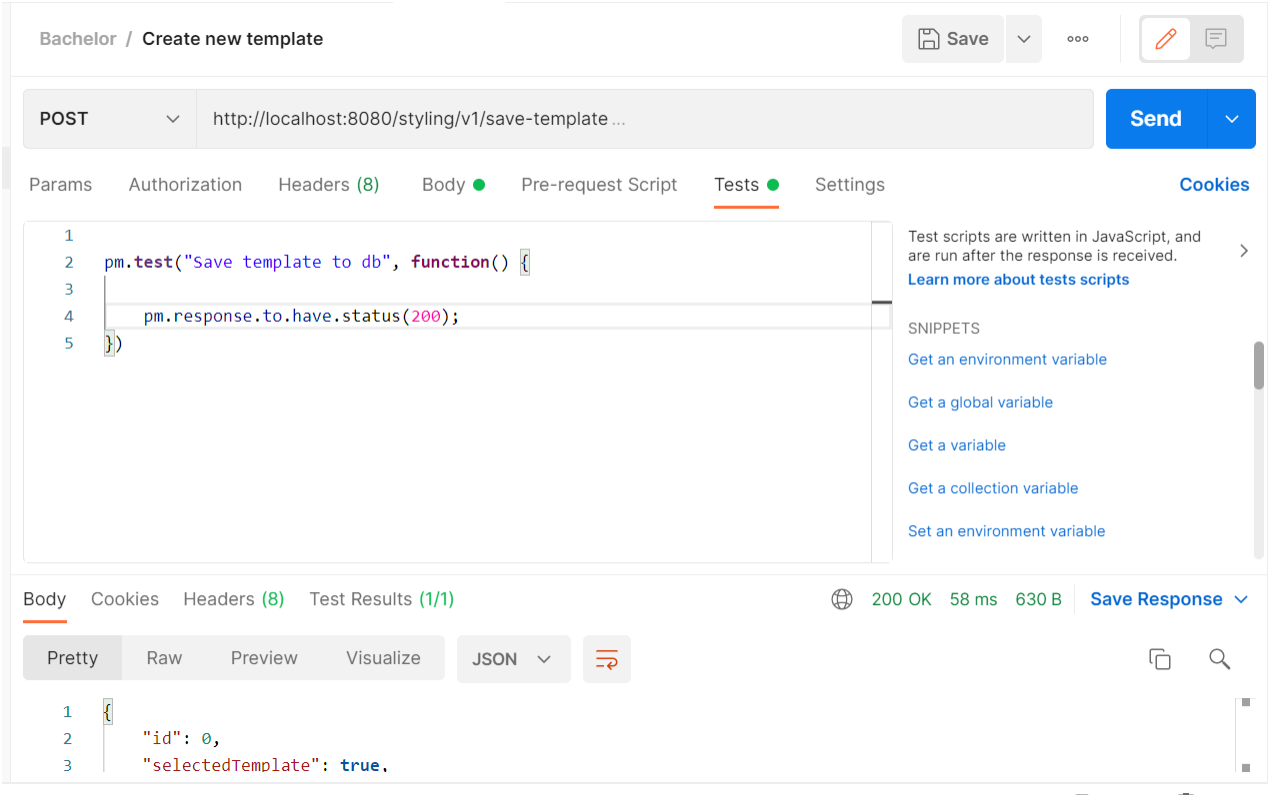
Figur 10.1. Testklasse for TemplateController

10.2 Instruks for kjøring

For å kjøre testene må en navigere til test-mappen i prosjektet. Ved å høyreklikke på kotlin-mappen kan man velge å kjøre “all tests”. Dette forutsetter at databasen inneholder data.

10.3 API - Postman

For testing og utvikling av API er verktøyet Postman blitt anvendt. Dette er et verktøy som kan sende forespørsler til API-endepunkter på en enkel måte uten å ha implementert en klient. I Postman kan man også definere tester (se Figur 10.2).



Figur 10.2. Postman test eksempel

10.4 Klient - Manuell testing

Under utvikling av klienten har gruppemedlemmene utført kontinuerlig manuell testing. Etter ferdigstilling av hver MVP er det blitt satt av tid til testing for å verifisere at klienten følger forventet flyt. Dersom det oppdages avvik har dette blitt utbedret. I tillegg har det blitt utført

testing av klienten av uavhengige personer som ikke har hatt en forventning til flyt. Tilbakemeldingene på funksjonalitet har vært verdifulle.

11 REFERANSER

SmartBear (2022) *What Is Unit Testing?*. Tilgjengelig fra:

https://smartbear.com/learn/automated-testing/what-is-unit-testing/ (Hentet 1. mai 2022).