Bacheloroppgave

Per Christian Bach Kvalvik
Espen Kvernstad
Jon Sondre Moen

Valg av open source rammeverk for tilstandsløst API

Mai 2020

NTNU
Norges teknisk-naturvitenskapelige universitet.
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for IKT og realfag

NTNU
Kunnskap for en bedre verden

Bacheloroppgave

2020

NTNU

Kunnskap for en bedre verden

Per Christian Bach Kvalvik
Espen Kvernstad
Jon Sondre Moen

Valg av open source rammeverk for tilstandsløst API

Bacheloroppgave
Mai 2020

NTNU
Norges teknisk-naturvitenskapelige universitet.
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for IKT og realfag

NTNU
Kunnskap for en bedre verden

Obligatorisk egenerklæring/gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

Du/dere fyller ut erklæringen ved å klikke i ruten til høyre for den enkelte del 1-6:		
1.	**Jeg/vi erklærer herved at min/vår besvarelse er mitt/vårt eget arbeid, og at jeg/vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.**	☒
2.	**Jeg/vi erklærer videre at denne besvarelsen:** • ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands. • ikke refererer til andres arbeid uten at det er oppgitt. • ikke refererer til eget tidligere arbeid uten at det er oppgitt. • har alle referansene oppgitt i litteraturlisten. • ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse.	☒
3.	**Jeg/vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7 og 4-8 og Forskrift om eksamen §§14 og 15.**	☒
4.	**Jeg/vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert i Ephorus, se Retningslinjer for elektronisk innlevering og publisering av studiepoenggivende studentoppgaver**	☒
5.	**Jeg/vi er kjent med at høgskolen vil behandle alle saker hvor det forligger mistanke om fusk etter høgskolens studieforskrift §31**	☒
6.	**Jeg/vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider**	☒

Forord

Bacheloroppgaven og denne rapporten er skrevet av tre studenter og er den siste delen i vår utdanning som dataingeniører ved Norges teknisk-naturvitenskapelige universitet i Ålesund.

Gjennom denne oppgaven har vi utforsket flere spennende teknologier og rammeverk. Selv har vi stor interesse for temaet i oppgaven og det har vært en motiverende faktor gjennom hele løpet. Vi har fått mye erfaring og kunnskap som kan være til stor nytte i arbeidslivet framover.

Vi vil takke Mikael Tollefsen for gode innspill og støtte som hovedveileder gjennom hele prosessen.

Sammendrag

Det kommer stadig nye teknologier som gjør det enklere å bygge nettsider med separat logikk og brukergrensesnitt. Det kan være vanskelig å velge hvilken teknologi som er best tilpasset forskjellige bruksområder til personlig eller kommersielt bruk. I dette dokumentet bruker vi diverse teknologier for å utforske dette emnet. Problemstillingen i denne oppgaven er derfor valg av open source rammeverk for tilstandsløs API for utviklere og bedrifter.

For å løse problemstillingen har vi modellert et API for et nettsamfunn, og dette har vi implementert i tre forskjellige rammeverk: FastAPI, Spring Boot og ASP.NET Core Web APIs. For at vi skal kunne demonstrere at API-et er egnet til en reell nettside har vi også laget en front-end løsning med Vue.js. Denne er kompatibel med API-ene våre, og gir brukeren et valg av hvilken API som skal brukes.

For å sikre lik funksjonalitet blant API-ene har vi laget et testverktøy som tester selve API-et over HTTP. Ytelsesmålinger av API-ene er gjort med open source verktøyet JMeter.

Vi har sammenlignet brukeropplevelsen av rammeverkene fra en utviklers perspektiv, og vi har sammenlignet resultatene av ytelsesmålinger. I slutten av rapporten har vi drøftet rundt rammeverkene og hvilke vi vil anbefale andre å bruke.

Innhold

Terminologi

Begreper

API	Application Programming Interface: grensesnitt laget for å kommunisere med og operere ulike deler av et program.
Assignee	Oppdragstakeren. Den som står som ansvarlig for en issue på GitHub.
Back-end	Logiske prosesser som skjer i bakgrunnen i en applikasjon.
Backlog	Etterslep. En liste med oppgaver, gjerne alle oppgaver som skal løses i et prosjekt.
Base64	Algoritme som koder tekst til binært format, men representert av tekst som bruker 64 unike tegn.
Branch	Gren. En repository kan ha flere uavhengige grener der det jobbes med forskjellige problemer.
Bug	Feil i programmet som fører til ukorrekte eller uventede resultater.
Build tool	Byggeverktøy for å automatisere prosessen med å gjøre om kildekode til kjørbare filer.
Commit	Oppdatering av kodebasen i git: å lagre kodens nåværende tilstand slik at den kan gjenopprettes om det skal være nødvendig.
Debugging	Prosess som hjelper med å fjerne bugs i koden.
Flerfaktor-autentisering	Autentisering som krever mer enn én identitetssjekk.
Front-end	Presentasjonslaget i en applikasjon: delen som brukerne direkte ser og bruker.
GitHub Repository	En GitHub repository er en katalog der alt tilhørende prosjektet ligger. Filer, issues, wiki og mer.
Header	Ekstra data i en forespørsel, sendt som en overskrift i stedet for i datafeltet.
HTTP Basic Auth	Brukernavn og passord metode for autorisering med HTTP som sendes som en HTTP header.
IDE	Integrert utviklingsmiljø: En komplett programvarepakke for utviklingsmiljø egnet til programmering.
Interpreted	Programmeringsspråk hvor instruksene i koden blir lest og kjørt hver gang programmet kjøres.
Interpreter	Kjørbart program til et programmeringsspråk hvor man kan skrive og kjøre instrukser uten å sette opp et miljø først.
Issue	Problem. I GitHub er en issue noe som må gjøres.
JSON serialisering	Konvertere objekter i kode til JSON eller omvendt.
Keyword argument	Parameter i en funksjon satt med et nøkkelord i Python, f.eks *port=8000*.

Konstruktør	Metode som bygger instansen/objektet i et objektorientert programmeringsspråk.
Kontroller	En komponent i koden til et API som holder styr på alle endepunktene for API-et.
Label	Etikett. I GitHub bruker vi en label for å kategorisere issues.
Lazy loading	Å kjøre forespørsler om relasjoner i databasen automatisk når ORM feltene brukes i koden.
Merge	Å slå sammen kode (i kontekst fra en pull request i GitHub) med hovedbransjen i en git repository.
Multi-page application	Nettside hvor kun den relevante siden lastes inn. Navigering vil laste inn delte komponenter på nytt.
Open source	Kildekode som er åpen for offentligheten, hvor enhver person kan endre koden og distribuere den videre.
Path variable	Stivariabel. En variabel som skrives inn i stien til forespørselen.
Post	Et innlegg med tittel og innhold som lages i en resub.
Pull request	En funksjon i GitHub for å foreslå endringer i kildekoden.
Python dataclass	En måte i Python å lage oversiktlige datamodeller.
Rammeverk	Bibliotek innen programvareutvikling som implementerer en 'base' av et program, som kan utvides til å bygge en applikasjon med ønskede krav.
Refaktorering	Endring av navn og omskriving av kode.
Request body	Datafeltet i en forespørsel.
Resolver	Funksjon som henter en ressurs basert på parameter.
Resub	Et samfunn for en samling av poster i API-et vårt.
REST	En type arkitektur som brukes til å lage tilstandsløse nettjenester. Disse opererer altså uten å lagre klienters tilstand på serveren.
Ruter	Komponent i et API som definerer stien til endepunktene, ofte brukt i en kontroller.
Secret	Secret er en unik variabel, gjerne for konfigurasjon, som skal være hemmelig for andre.
Single-page application	Nettside hvor hele siden lastes inn på én gang som en applikasjon. Navigering laster kun ressursene som skal vises.
Spring Initializr	Et verktøy for å sette opp Spring Boot prosjekter med valgte innstillinger.
Spring Repository	En repository i Spring er koblingen til databasen.
Sti	Representasjon av plasseringen til en ressurs.
Stress-test	Bestemme PC-en sin evne til å vedlikeholde en effektiv fart under krevende omstendigheter.
Syntax highlighting	Med syntax highlighting menes det at tekst som inneholder kodeord for programmeringsspråk eller lignende vil bli farget eller på annen måte dekorert slik at de enkelt blir leselig av utvikleren.

Token	Tilgangstoken.
Uvicorn worker	En Uvicorn prosess som ofte settes på en egen tråd i prosessoren.
Webhook	Grensesnitt for å lytte til oppdateringer av en ressurs.

Forkortelser

API	Application Programming Interface
APT	Advanced Package Tool
ASGI	Asynchronous Server Gateway Interface
BSD	Berkeley Software Distribution
CORS	Cross-origin Resource Sharing
CRUD	Create, read, update and delete
CSS	Cascading Style Sheets
HTTP	Hypertext Transfer Protocol
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
IEFT	Internet Engineering Task Force
IP	Internet Protocol
JSON	JavaScript Object Notation
JWT	JSON Web Tokens
MPA	Multi Page Application
MVC	Model-View-Controller
OOP	Object-Oriented Programming
ORM	Object-Relational Mapping
OSI	Open Source Initiative
PSFL	Python Software Foundation License
REST	REpresentational State Transfer
SPA	Single Page Application
URL	Uniform Resource Locator
WSGI	Web Server Gateway Interface

1 Innledning

1.1 Bakgrunn

Vi er tre studenter som studerer dataingeniør ved NTNU i Ålesund. Alle tre har stor interesse for back-end, separert API og front-end, og det er dette vi ønsker å jobbe med i framtiden. I denne oppgaven har vi valgt å jobbe med open source rammeverk og open source verktøy.

Rammeverkene vi skal bruke til dette prosjektet er FastAPI skrevet i Python, Spring Boot skrevet i Java og ASP.NET Core Web APIs skrevet i C#.

1.2 Problemstilling

Problemstillingen er å velge det rammeverket som er best egnet til hvert enkelt prosjekt. Som programvareutvikler i dag er det veldig mange ulike rammeverk og programmeringsspråk å forholde seg til, og det kan medføre at utviklere og bedrifter ikke velger det mest optimale rammeverket for å lage løsningen til kunden. Vi vil ha spesielt fokus på open source rammeverk for å bygge et tilstandsløst API. De ulike rammeverkene er laget i forskjellige språk, med egen dokumentasjon og andre hjelpemidler for utviklere.

Det er mange faktorer å tenke på når man skal velge et rammeverk. Noen går på selve utviklingen slik som god dokumentasjon og veiledning til hvordan rammeverket skal brukes. Disse faktorene går ut på utviklerens “brukeropplevelse” med rammeverket. Andre faktorer er viktige for det ferdige produktet. Hastighet målt i forespørsler til API-et per sekund er et viktig mål, men også responstid under tung bruk.

1.3 Hensikt og målsetting

Målet med denne oppgaven er å skrive en rapport der vi sammenligner tre rammeverk for å lage et tilstandsløst API. Denne rapporten skal inneholde en beskrivelse av hvordan hvert enkelt rammeverk var å jobbe med. Vi vil se på hvor god introduksjon vi får av hvert rammeverk og hvordan det er å starte utviklingen. Hvor god er dokumentasjonen for de forskjellige delene av rammeverket, og er det enkelt å finne fram til de delene man er interessert i? Vi vil også måle ytelse som hastighet og responstid.

For å kunne svare grundig på disse spørsmålene vil vi anvende tre forskjellige rammeverk for å lage tre implementasjoner av samme API spesifikasjon. Produktet vi vil lage er et forum lignende nettsted der brukere kan lage sine egne underforum, poster og kommentarer med inspirasjon fra det massivt populære nettsamfunnet Reddit (*Homepage - Reddit*, no date). Vi vil også bruke et front-end rammeverk for å lage et grensesnitt som skal kunne kobles opp mot alle tre API-ene uten forskjeller i funksjonalitet.

1.4 Kravspesifikasjon

Repost skal være et forum lignende nettsted der brukerne skal lage sine egne underforum, og alle brukere skal kunne skrive poster og kommentarer i underforumene. Innholdet skal dermed være generert av brukerne. Brukere av Repost skal kunne lese informasjon uavhengig av om de er logget inn eller ikke. Om en bruker vil delta ved å stemme, poste, redigere eller slette innhold i repost må de lage en bruker og logge inn slik at de kan autentiseres med en autentiserings token.

En ikke-autentisert bruker skal kunne

- Lage en bruker
- Logge inn for å få en autentiserings token
- Få informasjon om en bruker
- Få informasjon om en resub
- Få informasjon om en post
- Få informasjon om en kommentar
- Få en liste over resuber eid av en bruker
- Få en liste over poster skrevet av en bruker
- Få en liste over kommentarer skrevet av en bruker
- Få en liste over resuber
- Få en liste over poster i en resub
- Få en liste over kommentarer i en post

En autentisert bruker skal kunne

- Lage en resub
- Redigere sin egen resub
    - Overføre eierskap av resuben til en annen bruker
- Slette sin egen resub
- Lage en post i en resub
- Redigere sin egen post
- Slette sin egen post
- Slette en post i sin egen resub
- Gi positiv eller negativ stemme til en post
- Fjern stemme fra en post
- Lage en kommentar i en post
- Redigere sin egen kommentar
- Slette sin egen kommentar
- Slette en kommentar i sin egen resub
- Gi positiv eller negativ stemme til en kommentar
- Fjerne stemme fra en kommentar

En autorisert bruker som eier en resub kan altså slette poster eller kommentarer laget av andre brukere i deres resub. De vil ikke kunne slette innhold i andre resuber og de kan ikke redigere noen andre sitt innhold.

En autentisert bruker kan også stemme opp eller stemme ned poster og kommentarer. Vi vil sortere poster og eventuelt kommentarer basert på en ratio mellom antall positive stemmer og tiden fra posten ble laget slik at de mest populære postene havner øverst i sorteringen.

1.5 Kriterier for god brukeropplevelse

Vi har satt opp kriterier som rammeverkene burde oppfylle for å gi en god opplevelse under utvikling. Brukeropplevelsen er viktig ettersom det kan lede til mindre arbeid av utvikleren, som vil spare tid og kostnader hos selskap. Kriteriene vi har laget baseres på vår tidligere erfaring med utvikling av API-er, og vi har satt opp noen mer spesifikke krav basert på hva vi selv mener er viktigst.

- OpenAPI dokumentasjon med støtte for Swagger UI
- OAuth2 autentisering med password flow
- ORM for å lage tabeller og relasjoner fra klasser
- ORM for å lage spørringer til databasen
- God og veiledende dokumentasjon for bruk av rammeverket
- Legger opp for lite gjenbruk av kode
- Raskt og enkelt å teste nye endringer i koden
- Enkelt å modellere endepunkter og parametere i endepunktene
- Enkelt å spesifisere formatering av navn i API skjema uavhengig av programmeringsspråket
- Enkelt å bygge og kjøre applikasjonen i et testmiljø
- God støtte for ekstra funksjonalitet gjennom tredjepartsbiblioteker
- Støtte for alle populære operativsystem slik at utvikleren kan jobbe i sitt foretrukne miljø

I tillegg til disse kriteriene vil vi se på antall linjer koder for hver implementasjon. Dette er ikke et godt mål for brukervennlighet, men det er greit å få et tall på dette. Det kan være interessant å se hvilke filer som har mest linjer kode og om det er en lignende trend blant alle rammeverkene.

2 Teoretisk grunnlag

2.1 Open Source

Open source i programvare sammenheng er kode som er fritt tilgjengelig for å brukes, endres og delt videre av alle. For at kode skal falle under definisjonen av open source er det ti kriterier i Open Source Definition, tilgjengelig på Open Source Initiative sin hjemmeside, som må være oppfylt.

Dette er hovedtrekkene i definisjonen. Lisensen skal ikke diskriminere mot personer eller grupper, og den skal ikke sette restriksjoner for hvilke fagfelt eller organisasjoner som kan bruke programvaren. Kildekoden skal være fritt tilgjengelig og det skal være tillatt å endre koden om det er ønskelig. Programvaren kan deles videre slik den er eller som en del av en annen programvare eller distribusjon. Programvaren kan også endres og bli utgitt som et nytt produkt under samme lisens (*The Open Source Definition* | *Open Source Initiative*, 2007).

I OSI sin FAQ går det igjen at det er lisensen som bestemmer hvilke regler som gjelder for hvert enkelt prosjekt. Det er veldig viktig at den originale lisensen beholdes om man skal bruke det videre i sitt eget produkt (*Frequently Answered Questions* | *Open Source Initiative*, no date).

Selv om open source sier at kildekoden skal være fritt tilgjengelig er det fortsatt mulig å selge tjenester basert på den åpne koden. Det er også lov til å selge programvare til andre, men da kan også de som kjøper programvaren selge den videre under samme lisens.

Det finnes flere forskjellige lisenser som er godkjente som Open Source License og følger den tidligere nevnte definisjonen på open source (*Licenses & Standards* | *Open Source Initiative*, no date). Følgende lisenser er brukt av de diverse rammeverkene og biblioteket i dette prosjektet, og de beskriver alle nevnte kriterier for open source.

The MIT License krever at selve lisensen må inkluderes videre der kode beskyttet av den er benyttet, endret eller solgt (*The MIT License* | *Open Source Initiative*, no date).

Apache License 2.0 har samme krav som MIT lisens, men krever også at alle notiser om opphavsrett er bevart. Notisen som er inkludert i lisensen må oppgis. Dersom man har gjort endringer av kode beskyttet av lisensen må dette tydelig oppgis. Lisensen beskriver også at man kan inkludere garantier i programvaren (*Apache License, Version 2.0* | *Open Source Initiative*, no date).

GNU GPL 2.0 har samme krav som Apache 2.0 lisens, men krever også at man ikke kan distribuere koden under en annen lisens. Altså kan man ikke inkludere en endret versjon av kode beskyttet med GNU GPL 2.0 i et prosjekt under f.eks MIT lisens. Man må også inkludere en kopi av den opprinnelige koden dersom noe er endret (*GNU General Public License version 2* | *Open Source Initiative*, no date).

The PostgreSQL License er en lisens laget av PostgreSQL og har de samme kravene som MIT lisens (*The PostgreSQL Licence (PostgreSQL)* | *Open Source Initiative*, no date).

The 3-Clause BSD Licence ligner på Apache 2.0 lisensen med unntak av at man ikke trenger å oppgi endringer av opprinnelig kode eller inkludere noe notis fra lisensen. Lisensen selv må likevel oppgis (*The 3-Clause BSD License* | *Open Source Initiative*, no date).

Python Software Foundation License (PSFL) er en lisens laget av Python Software Foundation. Den ligner på BSD lisensen men endringer av kildekoden må oppgis. Lisensen inkluderer heller ikke at man kan sette garantier på programvaren (*Python License (Python-2.0)* | *Open Source Initiative*, no date).

2.2 Tilgangskontroll

2.2.1 Autentisering

Autentisering innebærer å godkjenne identiteten til en bruker. For eksempel kan en bruker autentiseres ved bruk av påloggingspassord. Andre former for autentisering er fingeravtrykk, engangsnøkler eller tredjeparts tjenester som BankID på mobil hvor man er autentisert ved hjelp av personinfo, en godkjent enhet og en pin kode. BankID på mobil er også et eksempel på hvordan autentisering kan brukes som signering av dokumenter, pakker og mer (*Med din mobil - BankID*, no date). Autentisering med passord inkluderer også ofte flerfaktor-autentisering ved bruk av engangskoder (*Authentication vs. Authorization*, 2018).

2.2.2 Autorisering

Autorisering kan lett forvirres med autentisering. Forskjellen mellom disse begrepene er at autorisering ligger i å tillate en autentisert enhet å spørre om eller endre en ressurs. Dette steget skjer vanligvis etter autentisering og brukes til å sette opp et mer kontrollert system for tilgang til ressurser og metoder (*Authentication vs. Authorization*, 2018).

2.2.3 Tilgangstoken

Tilgangstoken er et konsept i autorisering hvor en autentisert bruker får tilsendt et kompakt dataobjekt som beskriver hvilke ressurser og metoder brukeren har tilgang til. Det er altså en måte for en klient å bevise at de har tilgang uten å måtte autentiseres på nytt (*Access Tokens - OAuth 2.0 Simplified*, no date).

2.2.4 OAuth2

OAuth2 er en standardisert autoriseringsprotokoll for HTTP styrt av IETF. Protokollens hensikt er å integrere pålogging via tredjepartsapplikasjoner i samme system som et tradisjonelt klient-server påloggingssystem (Hardt, 2012, p. 1).

Autoriseringsmetoder som ordinær servergodkjent passordinnlogging, hvor påloggingsøkten er autorisert direkte av serveren i minne, vil gjøre det vanskeligere å integrere rettigheter til tjenesten. Et typisk problem med denne metoden er at dersom tredjeparter skal kunne ha tilgang til tjenesten for en brukerkonto, vil de måtte lagre påloggingsinformasjonen til brukeren i klartekst for å kunne logge inn. Dette er et sikkerhetshull, ettersom brukeren selv er den eneste som burde vite sitt eget passord (2012, p. 4).

Et slikt system gir også andre utfordringer. En tredjepart som har tilgang til autorisering av en brukerkonto vil få de samme rettighetene som brukeren. Den får altså full tilgang, som ikke vil kunne begrenses. Samtidig vil alle tredjeparter knyttet til kontoen ligge under samme påloggingsinformasjon, slik at den eneste sikre løsningen for å fjerne tilknytningen er å endre brukerens passord. Dermed blir også alle andre koblinger fjernet.

OAuth2 løser disse problemene gjennom et nytt autoriseringslag. Tanken er at en ressurs (som regel eid av en server) kan gi forskjellige rettigheter til en tredjepart klient, som da vil få sin egen påloggingsinformasjon gjennom en slags nøkkel, i kilden referert til som en *access token*. Eieren av ressursen kan da lese av denne nøkkelen hvilke rettigheter som skal tilbys til klienten. Ved å bruke slike nøkler er da alt av brukeropplysninger frakoblet tredjepartsintegrasjon, og brukeren har kontroll over hvilke rettigheter som skal være tilgjengelig utenfra (2012, p. 5).

Bearer tokens brukes i OAuth2 for å autentisere brukere. En bearer token er en tilgangstoken som inneholder informasjon om hvilke ressurser en klient har tilgang til. Ved å bruke bearer tokens er det altså mulig å autoriseres gjennom OAuth2 uten å sende brukerens opplysninger. OAuth2 har ingen spesifikasjon på hvordan en slik tilgangstoken skal se ut, så det er opp til utvikleren hva som skal inkluderes i en bearer token (Hardt and Jones, 2012, p. 2).

2.2.5 JSON Web Tokens (JWT)

JSON Web Tokens (JWT) er en standard for å overføre data på en kompakt måte. Innholdet i en JWT er definert som JSON, og er kodet med Base64 formatet. JWT er nyttig for sikkerhet ettersom de kan signeres med en secret eller en asymmetrisk nøkkel, hvor signaturen er basert på innholdet i dataen. Slik kan dataen signeres fra serveren, og dersom en klient endrer innholdet vil signaturen ikke lenger være gyldig (auth0.com, no date).

En JWT er delt inn i tre deler separert med punktum. De tre delene er hode (header), data (payload) og signatur (signature). Hode beskriver algoritmen som brukes til å signere JWT. Data inneholder all informasjon som skal overføres. I en autoriseringsløsning er dette informasjon om tilgangsrettigheter eller gyldighet av tokenen. JWT er ikke nødvendigvis kryptert, og det er derfor viktig at dataen ikke inneholder sensitiv informasjon. Signaturen lages basert på innholdet i hode og data delene, og er hashet med en secret som kun serveren har tilgang til (auth0.com, no date). Strukturen på JWT med de tre delene i samme rekkefølge som definert over ser slik ut:

xxxx.yyyy.zzzz

2.3 Konsepter innenfor programmering

2.3.1 Objektorientert programmering

Objektorientert programmering (OOP) er bygget på ideen om å ha objekter med data og kode med metoder som utfører handlinger med disse dataene. De fleste populære OOP språk er klassebaserte. For å lage et objekt har man en klasse som beskriver hvilke felt og metoder objektet skal ha. Når den skal brukes lager man en instans av klassen og ender opp med et objekt (Wikipedia contributors, 2020d).

I de fleste OOP språk har man arv. Med arv kan man lage en ny klasse og spesifisere at den skal arve fra en eksisterende klasse. På den måten deles felles felt og metoder uten å måtte skrive det på nytt (Wikipedia contributors, 2020d).

2.3.2 REST API

REST er en forkortelse for REpresentational State Transfer. Det er flere veiledende prinsipper for hvordan en REST API skal være bygd eller hva den skal kunne støtte (*What is REST – Learn to create timeless REST APIs*, no date).

Klient og server skal være separert. På den måte spiller det ingen rolle for API-et hvordan klienten er implementert (*What is REST – Learn to create timeless REST APIs*, no date).

En REST API er tilstandsløs. At API-et er tilstandsløs betyr at alle forespørsler skjer i isolasjon. Alle forespørsler til en slik API må inneholde all informasjon nødvendig for å utføre forespørselen uten noen ekstra informasjon. Det innebærer som regel at man må ha med stien til endepunktet, alle parametre eller eventuelle data, og om nødvendig en autentiserings token eller lignende. Fordeler med dette er at det er enklere for serveren siden den kun trenger å svare på forespørselen og ikke noe annet (*REST – Statelessness – REST API Tutorial*, no date).

En REST API skal også være cacheable slik at samme svar kan gjenbrukes, skal ha et uniformt grensesnitt, og være lagvis slik at man bare ser det laget man samhandler med (*What is REST – Learn to create timeless REST APIs*, no date).

2.3.3 Hypertext Transfer Protocol (HTTP)

I følge Henrik Dvergsdal (Dvergsdal, no date) er Hypertext Transfer Protocol (HTTP) en kommunikasjonsprotokoll for å overføre HTML-dokumenter. Disse overføringene skjer imellom tjenere og klienter gjennom en transportprotokoll. Klienter får brukt denne protokollen ved å skrive inn HTTP i url-adressen i nettleseren de bruker.

2.3.4 OpenAPI spesifikasjon

OpenAPI Specification er et sett med regler som brukes for å lage verktøy for utvikling av et API. Tidligere var navnet på standarden Swagger, og mange verktøy bruker fortsatt Swagger navnet. OpenAPI spesifikasjonen ble donert til Linux Foundation i 2015, og som navnet tilsier er det en

åpen spesifikasjon som alle kan bruke. OpenAPI kan brukes i forskjellige stadier under utviklingen. Det er mulig å designe et API, dokumentere et API og generere kode og tester for dette API-et med verktøy som er bygd for denne standarden (*API Resources | Swagger*, no date).

2.4 Kodespråk

2.4.1 Python

Python er et open source, PSFL lisens, primært objektorientert programmeringsspråk laget av Guido van Rossum og nå eid av Python Software Foundation. På hjemmesiden til Python står det "*Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.*". Python er enkelt å lære siden syntax er relativt enkelt og det er stort fokus på at koden som blir skrevet skal være lett leselig. Siden Python er interpreted vil koden leses og kjøres av Python sin interpreter i sanntid, og det kreves ingen kompilering av koden. På grunn av dette er det veldig raskt å teste endringer i koden som kan gjøre utviklingen enda raskere enn med andre språk (*What is Python? Executive Summary*, no date).

Python har veldig mange tredjeparts biblioteker som tilbyr rammeverk og funksjonalitet for diverse bruksområder. Disse pakkene er tilgjengelige gjennom Python Package Index, eller PyPI. Det er også mye funksjonalitet inkludert i standard biblioteket, og dette er godt dokumentert i den offisielle dokumentasjonen. I tillegg til god dokumentasjon finnes det veldig mange veiledninger og eksempler på bruk av diverse moduler eller konsepter (*Welcome to Python.org*, no date).

2.4.2 Java

Java er et objekt- og klasse-orientert programmeringsspråk laget av Sun Microsystems og nå eid av Oracle. Det er et veldig populært språk, særlig i profesjonell sammenheng eller i integrerte systemer. Java kompileres ned til Java Bytecode i stedet for vanlig maskinkode. Koden må derfor bli kjørt av Java Virtual Machine (JVM) som gjør det mulig å kjøre den samme kompilerte koden på alle enheter som støtter Java uavhengig av hvilken arkitektur enheten bruker. Det gjør at Java kan brukes på veldig mange enheter, i alt fra servere, datamaskiner, mobiltelefoner til brødristere.

Sammenlignet med lavnivåspråk har Java fått kritikk for å dårligere ytelse og for å ha manglende tilgang til lavnivå styring. I gjengjeld er det innebygget en del funksjonalitet som skal gjøre utviklingen enklere, som garbage collector som frigjør utvikleren fra å måtte allokere og frigjøre minne manuelt (Contributors to Wikimedia projects, 2020).

Java har et stort standardbibliotek med mange nyttige pakker, men det er også veldig mange tredjepart biblioteker og pakker. For Java brukes ofte Maven eller Gradle som build tool for å holde orden på avhengigheter.

2.4.3 C#

C# er et programmeringsspråk laget av Microsoft. C# er objektorientert og er ment å være et enkelt, moderne programmeringsspråk som kan brukes til det meste. Språket benytter seg av klasser, og metoder er en del av en klasse og kan brukes som en funksjon. C# bruker properties for å få tilgang til verdier i et objekt uten å måtte lage egne get og set metoder. For å holde

orden på et prosjekt brukes et *namespace* som kan importeres i andre deler av koden med kodeordet *using* (Wikipedia contributors, 2020a).

Syntax er lignende andre C-stil språk, som gjør at det også ligner på Java. Selv om Java og C# har lignende syntax har de tatt forskjellige veier videre, og den første nye iterasjonen av C# begynte å gå en annen retning (Wikipedia contributors, 2020a).

2.4.4 JavaScript

I javascript.info står det at JavaScript er kode som kjøres automatisk når en nettside er lastet inn i en nettleser. Skriptene i dette språket er interpreted, og dermed tildelt som enkel tekst og trengs ikke å bli kompilert for å kjøres. JavaScript i nettleseren kan gjøre alt som er relevant for nettleser-manipulering. Det er en rekke med oppgaver JavaScript kan gjøre:

- Legge til ny informasjon til HTML Dokumentet, det vil si endre på eksisterende innhold eller endre stil.
- Reagere til brukerens handlinger. Kjøre funksjoner ved museklikk, musebevegelse eller tastetrykk.
- Be om informasjon fra diverse servere, eller laste opp og ned filer.
- Hente og sende cookies.
- Lagre data på klienten sin side (i nettleseren).

(*An Introduction to JavaScript*, 2020)

2.4.5 Hypertext Markup Language (HTML)

Computerhope.com forklarer at Hypertext Markup Language (HTML) er brukt til å lage elektroniske dokumenter som nettlesere kan lese og forstå. Dette var utviklet av Tim Berners-Lee i 1990. HTML legger til format til informasjon som gjør at tekst, bilder og videoer vises riktig i nettlesere. HTML bruker tagger <> til å formatere eller legge til funksjonalitet (Computer Hope, no date).

HTML5 er en oppdatering til HTML fra HTML4. HTML5 legger til nye tagger og egenskaper til disse taggene, og bruker fortsatt det samme oppsettet som HTML4. Disse nye taggene tillater bedre semantikk og dynamiske elementer som er aktivert av JavaScript (Computer Hope, no date).

2.4.6 Cascading Style Sheets (CSS)

Cascading Style Sheets (CSS) er beskrevet av tutorialspoint.com som et enkelt design språk tilægnet for å forenkle prosessen av å lage stil til nettsider. CSS behandler hvordan nettsiden ser ut og kan for eksempel skifte på farge og størrelse på tekst, oppsett, kanter, bildestørrelse osv. CSS har stor kontroll over presentasjonen av et HTML dokument (*What is CSS? - Tutorialspoint*, no date).

2.5 Relasjonsdatabase

En relasjonsdatabase er et system bestående av flere tabeller som er linket sammen med relasjoner. I tabellene lagrer man data av forskjellige typer, og for å lage en relasjon har man et felt med felles data. Ved å kjøre spørringer til en relasjonsdatabase vil man kunne hente ut informasjon om en rad som er lagret i tabellen, og man kan hente ut informasjon som er relatert til denne raden (IBM Cloud Education, 2019).

2.5.1 Relasjoner

Det er tre typer relasjoner som er vanlige i en relasjonsdatabase. Én-til-én, én-til-mange (også mange-til-én) og mange-til-mange. Én-til-én kobler en ressurs mot kun én annen ressurs. Den er ikke brukt ofte da man kan oppnå det samme ved å lagre denne dataen i én og samme tabell. Den mest brukte relasjonen er én-til-mange (eller mange-til-én). Med denne relasjonen kan man for eksempel gjøre en bruker til eier av flere objekter eller lignende. Mange-til-mange relasjoner er ofte en tabell som knytter sammen to andre tabeller slik at begge tabellene kan ha flere relasjoner begge veier (Ian, 2016).

2.5.2 Structured Query Language (SQL)

Structured Query Language (SQL) er et standardisert språk for å kjøre spørringer mot relasjonsdatabaser (IBM Cloud Education, 2019).

2.5.3 Object Relational Mapping (ORM)

For å kunne bruke data fra en database i et programmeringsspråk der man behandler objekter må man bruke teknikken Object Relational Mapping (ORM) for å konvertere mellom de to (Wikipedia contributors, 2020e). Med å bruke et ORM verktøy kan man lage en klasse i koden som tilsvarer en tabell i en database. Vi kan da lage relasjoner i koden og disse blir implementert i databasen. For å hente, slette og legge inn data bruker vi metoder i koden som blir overført til databasen (*SQL (Relational) Databases - FastAPI*, no date).

2.6 Utviklingsprosesser

2.6.1 Smidig prosess

Programvareprosjekter bruker ofte en smidig utviklingsprosess. Smidig prosess har ingen faste regler eller implementasjoner, men det finnes metoder som bruker konseptene og bygger videre på de. En smidig utviklingsprosess innebærer god kommunikasjon mellom utviklingsteam og kunde. Produktet får små men hyppige oppdateringer, slik at kunden kan se fremgangen og eventuelt be om endringer. En slik prosess gir dermed mulighet for å identifisere problemer tidlig og rette de opp før alt for mye arbeid er lagt ned. Det skal være kort vei fra forslag til implementasjon og kommunikasjon er satt i fokus. Med en slik prosess blir det mer åpenhet slik at for eksempel problemer ikke holdes skjult for kunden (Atlassian, no date).

Manifestet for smidig programvareutvikling (*Manifestet for smidig programvareutvikling*, no date) sier:

- *Personer og samspill* fremfor prosesser og verktøy
- *Programvare som virker* fremfor omfattende dokumentasjon
- *Samarbeid med kunden* fremfor kontraktsforhandlinger
- *Å reagere på endringer* fremfor å følge en plan

De mener at selv om alle disse punktene er viktige er det det som står i kursiv som er det viktigste. Dette er utgangspunktet for hvordan smidig utvikling blir brukt.

2.6.2 Scrum

Scrum er et sett med retningslinjer for hvordan man skal jobbe sammen for å effektivisere utvikling og levering av et produkt til kunden. Scrum har noen faste normer og regler, men mange velger å tilpasse prosessen slik det passer best for deres behov. Kilden for dette avsnittet og resten av dette underkapittelet bruker samme side fra Atlassian som kilde (Atlassian, 2018).

Scrum beskriver tre nødvendige roller: produkteier, scrum master og utviklingsteamet. En **produkteier** er en enkelt person som er knytningen mellom kunden og utviklingsteamet. Først og fremst må produkteier ha god kommunikasjon med og forståelse for hva kunden ønsker. I tillegg trengs domenekunnskap og kjennskap til markedets forventninger, standarder og krav. Å sette opp og organisere produkt backlog, informere teamet hva som må prioriteres og bestemme om produktet kan leveres er også produkteiers oppgaver.

Scrum master er ekspert på scrum prosessen og hjelper produkteier, utviklingsteamet og kunden med å forstå prosessen. En scrum master skal ha en helhetlig forståelse for arbeidet som utføres for å kunne gi de beste rådene og identifisere punkter som kan forbedres i prosessen.

Utviklingsteamet er de som utfører arbeidet som blir lagt i backlogen, og deres fokus er å fullføre oppgavene som er satt for sprinten. Utviklingsteamet består av personer med forskjellige

ferdigheter og spisskompetanse slik at de komplimenterer hverandre. Et effektivt team består av få personer. Det er ingen fast regel på hvor mange man kan være, men en tommelfingerregel er fem til syv medlemmer.

Produkt backlog er listen av alt som må gjøres i prosjektet. Her kan det legges til oppgaver eller omorganiseres underveis. Produkteier passer på at de riktige oppgavene blir prioritert i produkt backlogen.

Ideene fra smidig prosess der man har små iterasjoner av produktet der kunden får se nye endringer siden sist kalles springs i Scrum. En sprint er ofte to eller fire uker lang. En sprint skal ha klart definert hvilke oppgaver som skal gjøres og når en sprint er ferdig skal disse oppgavene være ferdige. Om ikke alle oppgavene ble ferdig må grunnen til dette identifiseres slik at neste sprint går som planlagt.

Når en sprint er ferdig vurderer teamet om produktet er klart for levering. Om det blir gitt klarsignal kan den nye versjonen rulles ut og tas i bruk. Ellers må det tas en ny vurdering etter neste sprint.

Sprint backlog er alle oppgavene som skal gjøres i en sprint. Valg av oppgaver diskuteres i sprint planleggingsmøtet før hver sprint. Oppgavene hentes fra produkt backlogen. Om det er andre oppgaver eller problemer som oppstår i sprinten skal disse legges ved i neste sprint backlog. Scrum vil nesten uten unntak ikke legge til eller fjerne arbeid fra en aktiv sprint backlog.

Sprint mål er den helhetlige forbedringen og produktet som skal være ferdig når sprinten er ferdig. For å nå sprint målet må sprint backlogen bli tømt.

2.6.3 Kanban

Kanban er en annen metode for smidig prosess. Det har mindre faste rutiner og lar teamet selv bestemme hva som passer best. Kanban er basert på kontinuerlig flyt der kommunikasjon skjer hele tiden og backlogen kan forandre seg underveis. Et kanban board er en tavle, gjerne digital, der alle oppgaver skal settes opp. Det enkleste oppsettet har kolonnene backlog, underveis og ferdig. Flere kolonner legges til etter behov. Det er vanlig å ha med en kolonne for godkjennelse slik at en annen på teamet må se over arbeidet før det blir lagt inn (Atlassian, 2019).

Prioritering kan bestemmes i samarbeid med kunden, men medlemmene i utviklingsteamet kan bestemme selv hva de ønsker å jobbe med. Kanban er veldig åpen på hvem som gjør hva og hvor langt i prosessen man er kommet. Kanban har ikke sprints på samme måte som Scrum, men det er viktig å ha dialog med kunden slik at de blir oppdatert på fremgang og kan komme med ønsker. En viktig forskjell fra Scrum er at om det oppstår et problem eller kunder kommer med et ønske som haster, vil det kunne bli jobbet med umiddelbart i stedet for å måtte vente til neste sprint. Produktet kan publiseres så ofte som teamet føler det er nødvendig. Noen velger å publisere ved store forandringer, mens andre gjør det så ofte som flere ganger daglig (Atlassian, 2019).

2.6.4 Git

Git er et open source, GNU GPL 2 lisens, versjonsstyringsprogram. Det er ment for å håndtere små og store prosjekter, men det er ikke begrenset til bare kildekode. Det er fullt mulig å bruke git for å holde kontroll på for eksempel tekst til en bok (*About - Git*, no date).

Raskt og effektivt var et av hovedmålene for prosjektet. Siden git utfører nesten alle operasjoner lokalt er det veldig raskt sammenlignet med systemer som er sentralt styrt. I tillegg til at det er veldig rask har det også integritet. Alle filer blir og commits får en checksum som brukes for å bekrefte at ingenting er endret siden sist (*About - Git*, no date).

Det som skiller git fra andre alternativer er branching og merging. Muligheten for å ha flere brancher slik at flere kan jobbe på forskjellige deler av koden uavhengige av hverandre er veldig viktig for effektiv utvikling der man unngår konflikter (*About - Git*, no date).

Det er også mulig å velge bare noen endringer som skal bli med i en commit. Du kan dermed jobbe med én funksjonalitet og om du ser at ikke alt du har gjort passer i en commit er det ingen problem å dele det opp (*About - Git*, no date).

Siden git er et distribuert system har alle en kopi av hele prosjektet. Når man kloner et prosjekt får man ikke bare nåværende versjon, man får hele historien til prosjektet. På den måten er det ingen enkeltpunkt for feiling så lenge flere enn én kopi av prosjektet finnes (*About - Git*, no date).

2.6.5 GitHub

GitHub Inc er et selskap eid av Microsoft og leverer tjenesten GitHub som er en nettside for deling av git prosjekter, kalt repositories. Andre tjenester som GitHub tilbyr er Pages og Gist. GitHub Pages kan tjene statiske nettsider, som for eksempel en blogg. For store prosjekter kan man ha dokumentasjon i en egen branch og tjene den som en GitHub page. Gist er en tjeneste for å dele små skript eller kodesnutter, men det har mange av de samme git funksjonene som et vanlig repository (Wikipedia contributors, 2020b).

GitHub lar brukere lage egne repositories. Det finnes to typer GitHub repository: privat og offentlig. Disse deler samme funksjonalitet med noen små unntak. Hovedunntaket er at i et privat repository blir gjerne samarbeidspartnere direkte invitert slik at de kan få skrivetilgang. I et offentlig repository er det vanlig at alle endringer blir lagt inn som pull request og eieren eller noen andre med rettigheter kan godkjenne og merge endringene inn i prosjektet. Pull request brukes også i private repository, men det er ikke strengt nødvendig som i offentlige (Wikipedia contributors, 2020b).

I tillegg til å være et delingssted for prosjekter har GitHub også mange verktøy for prosjektstyring og lignende. Noen av verktøyene er issues, pull request, milestone og projects (Wikipedia contributors, 2020b).

En issue er noe som skal gjøres, om det er et problem eller et forslag spiller ingen rolle. En issue har en beskrivende tittle, detaljer i beskrivelse og label for kategorisering (Wikipedia contributors, 2020b).

En milestone, eller milepæl, er et mål som skal nås før en bestemt dato. Her kan vi ha en tittel og beskrivelse. En issue kan knyttes opp mot en milestone og du kan da holde oversikt over progresjon (Wikipedia contributors, 2020b).

Projects er en Kanban tavle der du legger issues i relevante kolonner ettersom hva statusen er (Wikipedia contributors, 2020b).

En pull request er foreslåtte endringer, gjerne som en egen branch, der tittlelen sier hva den løser og beskrivelsen gir mer detaljer og linker til en issue som nå blir løst. Om en pull request blir godkjent av reviewer så kan den merges inn i master branchen. Da blir den automatisk lukket og issues som er referert blir lukket og flyttet til ferdig i Kanban tavlen (Wikipedia contributors, 2020b).

I tillegg til disse verktøyene finnes det utallige ekstra funksjonalitet som kan legges til gjennom markedsplassen (Wikipedia contributors, 2020b).

2.7 Programvaretesting

Testing av programvare er hovedsakelig utført for å teste kvaliteten av et program. Det kan innebære å sammenlikne ferdig implementasjon med kravspesifikasjon, testing av funksjonalitet og ytelsestester. Testene burde være implementert for å teste om endring av kode har uønskede resultat. Samtidig kan testene sørge for at nye versjoner av rammeverk eller programvarebibliotek fungerer som de skal. En av de største fordelene med programvaretesting er oppdagelse av bugs som kan oppstå når andre bugs blir fikset (Wikipedia contributors, 2020f).

2.7.1 Black-box testing

Black-box testing (også kalt spesifikasjonstesting) er del av “box framgangen” i testing av verktøy. Hensikten med denne framgangsmåten er å fokusere på funksjonalitet av et abstrahert lag av programmet. Black-box testing fokuserer på det et ytre lag, og tester altså ikke intern funksjonalitet av koden i dette laget. Testene er altså ikke begrenset av implementasjonsdetaljer som programmeringsspråk. Disse testene vil være nyttige for å forsikre om at deler av programmet fungerer som forventet i sin helhet, uten å teste intern struktur, kompleksitet eller ytelse (Wikipedia contributors, 2020f).

2.7.2 White-box testing

Det motsatte av black-box testing er white-box testing. I white-box testing blir det testet at de interne komponentene i et lag gjør som forventet, uten å teste ytre funksjonalitet. Disse testene krever altså kunnskap om den interne koden i laget. Slik kunnskap vil gjøre det enklere å designe et mer komplett testmiljø, siden man vil kjenne til alle de interne komponentene som må testes (Wikipedia contributors, 2020f).

3 Materialer og metode

3.1 Utviklingsprosess

Selv om vi har møte med veileder annenhver uke valgte vi å bruke kanban som utviklingsprosess. Vi har brukt samme prosess i tidligere prosjekter med gode resultater og føler at det vil fungere bra siden vi har flere utviklingsprosjekter som skal gjøres parallelt. Med kanban kan vi velge de oppgavene vi mener er viktige å gjøre uten at det skal være forhåndsbestemt. Dette er spesielt viktig om vi finner ut at det trengs å gjøres endringer for at en ny funksjonalitet skal fungere. Da kan vi prioritere å gjøre den endringen med én gang i stedet for å vente til neste sprint.

Vår utviklingsprosess er tett knyttet med verktøyet GitHub Projects (3.1.2) der vi har kanban tavlen. Projects legger opp til å bruke kanban, noe som er positivt for vår del.

3.1.1 Planlegging og møter

Siden vi har en iterativ utviklingsprosess er ikke alt planlagt i detalj før vi starter å arbeide. Siden vi skal ha flere implementasjoner av samme API spesifikasjon som demonstrasjonsprosjekt valgte vi å fokusere på spesifikasjonen først. Deretter laget vi en plan spesifikt for hver implementasjon i ettertid. Om det er noen endringer i planene eller noe som dukker opp underveis kan vi legge inn dette i backlogen når vi blir oppmerksomme på problemet.

3.1.1.1 Møte med veileder

Vi har møte med veileder annenhver uke. Før møtet sender vi en mail med kort oppsummering av fremgang siden sist møte og noen tanker rundt hva som skal gjøres de neste to ukene (Vedlegg 7). På møtet går vi gjennom det som har blitt gjort og eventuelle utfordringer. Etterpå diskuterer vi litt rundt hva som står på planen og hva som er viktig å fokusere på videre. Her noterer vi i grove trekk det som skal prioriteres, og eventuelt legger inn nye issues til backlogen om nødvendig.

3.1.1.2 Møte med gruppen

I sammenheng med møte med veileder har vi også møte internt i gruppen der vi diskuterer mer detaljert hva vi jobber med. For å planlegge oppgaver som må inn i backlogen på de forskjellige prosjektene møtes gruppen på skolen for å sette det opp sammen, men etter campus ble stengt på grunn av coronavirus måtte vi ta videre planlegging via Discord. Vi skriver til hverandre på Discord og har muntlige møter på samme plattform når det er nødvendig. Vi har også en kanal som automatisk poster aktivitet fra GitHub slik at alle er oppdatert på det som skjer.

3.1.2 Planleggingsverktøy

Vi har brukt git og GitHub i mange prosjekter tidligere og siden vi er vant med det ønsker vi å bruke det videre. Vi bruker den integrerte versjonsstyringsklienten i JetBrains IDE i stedet for å bruke kommandolinje eller et eksternt program. Den integrerte klienten er tett knyttet til andre funksjoner i IDEen og det er dermed veldig enkel og effektiv å bruke.

Her bruker vi GitHub Projects med Automation som verktøy for våre kanban tavler. Malene vi bruker er *Automated kanban* og *Automated kanban with reviews*. Vi har flere tavler siden vi skal implementere back-end flere ganger i tillegg til front-end. Vi har en tavle for hvert utviklingsprosjekt, altså hver back-end og front-end implementasjon, og vi har en tavle for helheten av prosjektet og en for skriving av rapporten.

3.1.2.1 Bachelorprosjekt

Den første kanban tavlen vi satt opp er bachelorprosjekt. Her skal vi ha overordnede oppgaver for prosjektet. Dermed blir disse relativt store oppgaver uten mye detaljer. Oppgavene kan deles opp videre i andre tavler der det er naturlig med mer detaljer.

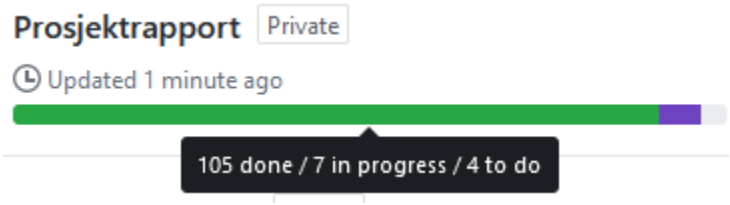
I denne tavlen bruker vi tre kolonner. *To do* er kolonnen for backlog der alle issues blir lagt. Når en issue blir jobbet med legges den i *In progress*, og når den er ferdig blir den lagt rett i *Done* når vi lukker issuen. Vi har altså ingen review i denne tavlen.

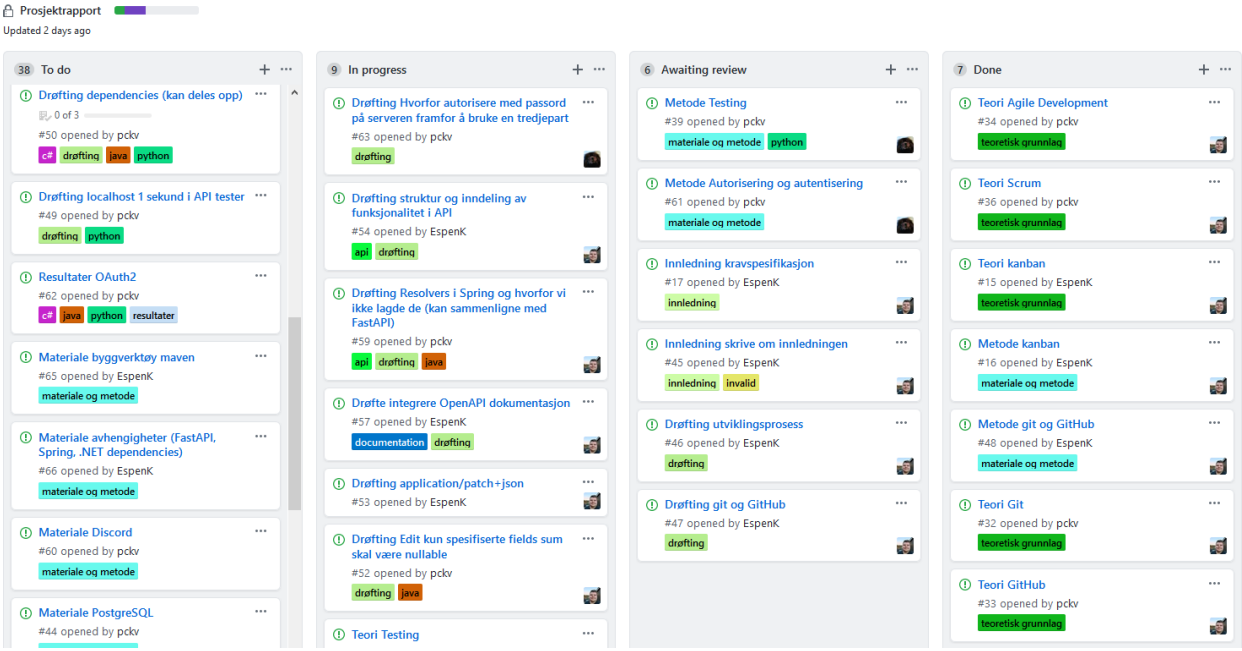
- Bachelorprosjekt
    - To do
    - In progress
    - Done

3.1.2.2 Prosjektrapport

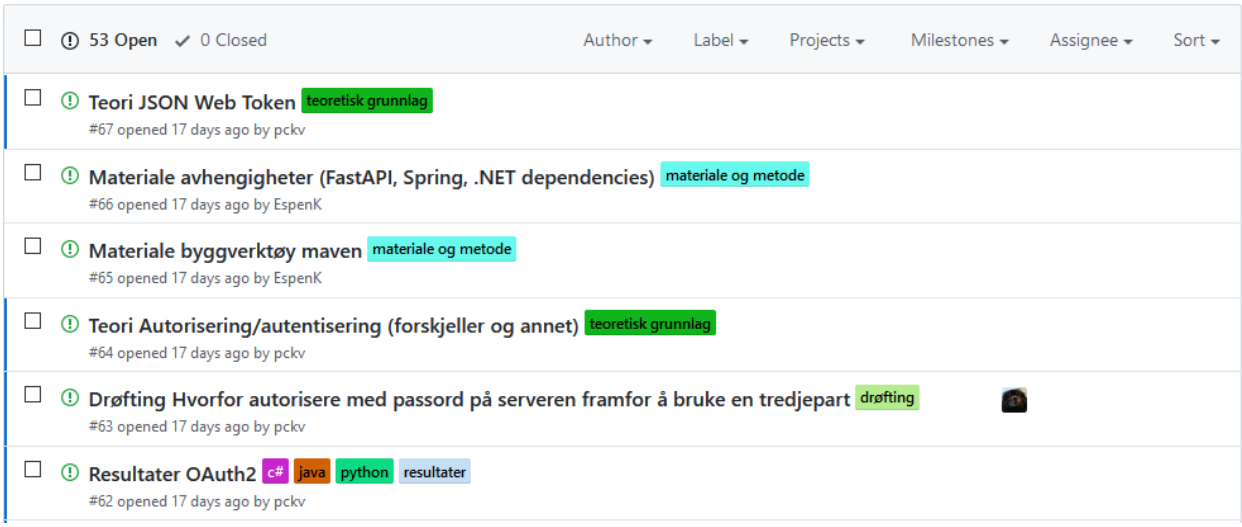
Prosjektrapport er en av de største tavlene vi har og omfatter alt som må skrives om i rapporten. Vi lager issue i *To do* for hver del som må skrives og legger til en label for kapittelet det tilhører. Her har vi lagt til en ekstra kolonne, *Awaiting review*, slik at noen må lese over det som blir skrevet før issuen blir lukket og lagt i *Done*. Kontroll av reviews blir håndtert med Google Docs (3.2.2).

- Prosjektrapport
    - To do
    - In progress
    - Awaiting review
    - Done





Figur 1. Kanban tavle til prosjektrapporten



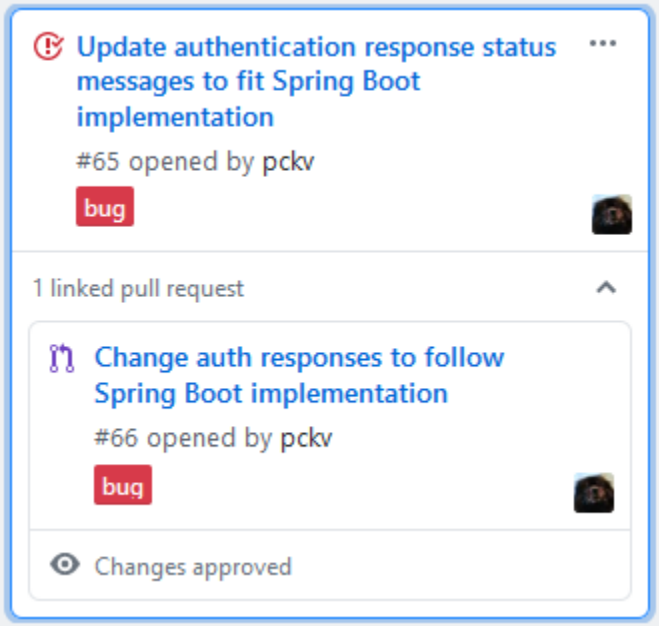
Figur 2. Liste over issues i prosjektrapporten sortert etter dato lagt til

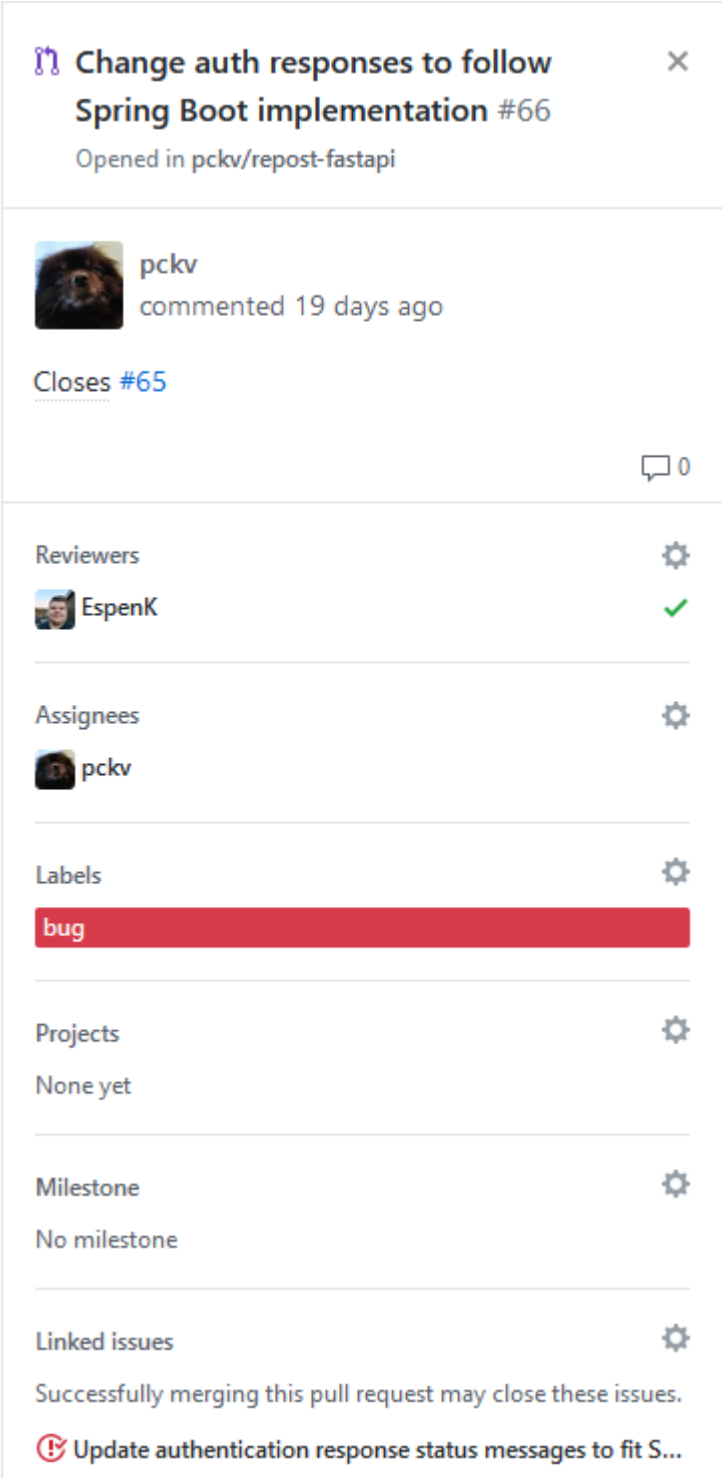
3.1.2.3 Back-end Python - Repost FastAPI

Issues for alle deler av koden som må implementeres i prosjektet for FastAPI. Hver issue har én eller flere labels som gjør det lettere å kategorisere og sortere. Vi bruker pull request for å lukke issues og styre reviews. Når en pull request får en godkjent review og blir merget, blir issuen lukket og flyttet til *Done* automatisk.

Nye pull requests som er koblet opp mot en issue havner sammen med issuen i kanban tavlen som en "linked pull request". I issuen får man da en visning av pull requesten som implemnterer det som er beskrevet. Slik havner relatert informasjon i samme issue, og vi får tettere navigering og kobling av issues.

- Back-end Python
  - To do
  - In progress
  - Awaiting review
  - Done





Figur 3. Detaljert visning av en automatisk lukket issue i kanban tavlen

3.1.2.4 Back-end Java - Repost Spring

Kanban tavlen for Spring implementasjonen er veldig lik den for FastAPI. Her har vi issues for alle delene av implementasjonen med label for enkel sortering.

- Back-end Java
    - To do
    - In progress
    - Awaiting review
    - Done

3.1.2.5 Back-end C# - Repost ASP.NET

ASP.NET tavlen er også lik de to andre back-end tavlene. Her bruker vi også label på issues for kategorisering, og vi bruker pull request for review og lukking av issues.

- Back-end ASP.NET
    - To do
    - In progress
    - Awaiting review
    - Done

3.1.2.6 Front-end - Vue

Vue tavlen er satt opp på samme måte som for tavlene til back-end prosjektene.

- Front-end Vue
    - To do
    - In progress
    - Awaiting review
    - Done

Vi bruker disse tavlene ved å lage issues i GitHub repositoryen og legge de inn i tavlen. Automation hjelper til med å sette issues på riktig plass. Nye issues blir lagt i *To do*. En issue blir manuelt flyttet til *In progress* når noen tar i oppgave å løse den. Når løsningen er klar blir issuen manuelt flyttet til *Awaiting review*, og en relatert pull request blir koblet opp mot issuen. Når løsningen er godkjent kan issuen lukkes og den blir automatisk flyttet til Done. For en relatert pull request blir issuen referert der og lukkes automasisk når pull requesten blir merget.

3.2 Utviklerverktøy

I dette kapittelet legger vi opp for diverse verktøy som er brukt under utvikling av dette prosjektet.

3.2.1 Git og GitHub

Hvordan vi bruker git og GitHub er relativt likt for ny funksjonalitet, retting av feil og rydding av kode. Her bruker vi ny funksjonalitet som eksempel, men det gjelder for alle issues.

Vi har en issue som beskriver ny funksjonalitet som skal legges til. Den som skal jobbe med dette problemet må sette seg selv som assignee og flytte issuen til *In progress* i kanban tavlen. Deretter må det lages en ny branch, gjerne lokalt først, der den nye funksjonaliteten blir laget. Når funksjonaliteten er lagt inn og branchen er pushet til GitHub skal det lages en pull request der issuen skal refereres og en annen på gruppen skal settes som reviewer, og issuen settes i *Awaiting review*. Reviewer skal laste ned den nye branchen, teste den nye funksjonaliteten og se over koden. Om koden blir godkjent er pull requesten klar til å merges. Når pull requesten blir merget med master blir issuen som er referert lukket sammen med pull requesten. Branchen kan slettes siden alle endringene ligger nå i master.

Om det er noe feil i pull requesten kan det legges til en kommentar eller så kan det bli lagt inn konkrete forslag til endringer som kan legges inn med et enkelt klikk av den som laget pull requesten.

Alle repositoriene der vi skriver kode er åpne for alle å se. Der vi planlegger skrivingen av rapporten er repository privat.

3.2.2 Google Docs og Google Drive

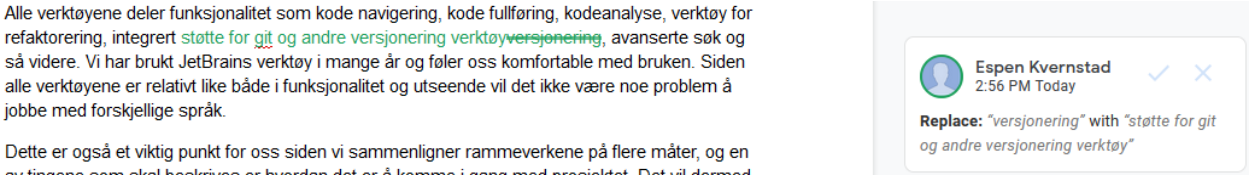
Rapporter, maler og noen bilder lagrer vi i en delt mappe på Google Drive. For å redigere dokumenter bruker vi Google Docs, som er et tekstredigeringsprogram tilgjengelig som webapplikasjon. Docs har mange av funksjonene som er forventet i et slikt program, og det er mulighet for å legge til ekstra verktøy som add-ons om det er behov for det. Siden det er en webapplikasjon fungerer det fint for alle operativsystemer der man kan bruke en moderne nettleser. Det er også app til mobil (Wikipedia contributors, 2020c).

Vi har tatt utgangspunkt i rapportmalen da vi opprettet dokumentet for prosjektrapporten. Vi valgte å opprette en ny fil og bruke Docs sine standardfonter og -overskrifter i stedet for å importere malen. Dette er for å unngå forskjeller i stil og skriftstørrelse.

Siden vi bruker GitHub Projects for å holde orden på skrivingen vet vi hva som jobbes med og hva som skal leses over av andre. Når vi har skrevet ferdig en del av rapporten blir den issuen lagt i *Awaiting review* og en annen på gruppen leser over det som er skrevet i rapporten. Om den som skal godkjenne kapittelet vil endre på noe kan man legge inn en kommentar eller endringsforslag. Det er mest effektivt å bruke endringsforslag for å fylle inn informasjon eller endre på en setning. Kommentarer er gjerne for større endringer. Den som skrev kapittelet kan

se over eventuelle kommentarer for forslag og velge å legge inn eller ignorere forslagene. Man kan også bruke kommentarer og forslag til å diskutere alternativer om det er nødvendig. Når alt er løst så kan issuen lukkes i GitHub Projects og den delen av rapporten er ansett som ferdig.

Figur 4. Kommentar i Google Docs



Figur 5. Forslag til endring i Google Docs

3.2.2.1 Paperpile

Vi har brukt Paperpile (*Citations and bibliographies for Google Docs - Paperpile*, no date) som en gratis Google Docs add-on for å håndtere referanser og referanseliste i rapporten.

3.2.3 Microsoft Word

Microsoft Word (*Microsoft Word - Word Processing Software | Office*, no date) er et tekstredigeringsprogram som del av Microsoft Office 365. NTNU har lisenser på Office 365 for studenter og dermed har vi tilgang på dette programmet. Vi skal bruke Word for å generere PDF dokument fra *.docx* eller *.odt* filer av rapporten som er skrevet i Google Docs.

3.2.4 Discord

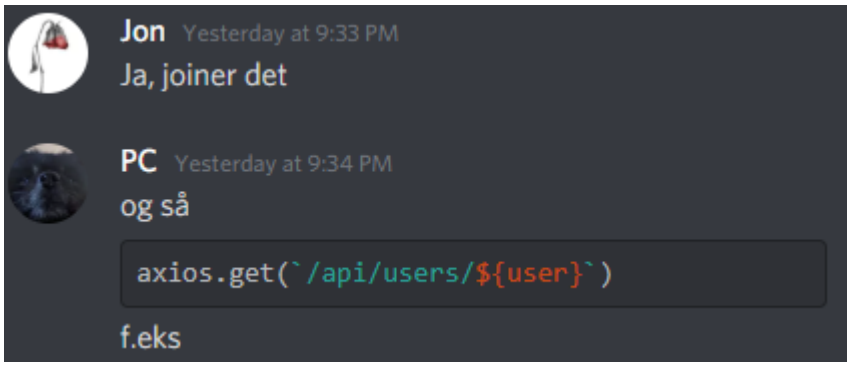
Discords primære målgruppe er blant folk med interesse for dataspill, men det er også relevant for andre målgrupper. Dette viser de til på sin hjemmeside der de skriver "*We created Discord to bring people together around games. We're humbled every day to learn how you use it for more.*" (*Discord — How People are Using Discord to Keep in Touch*, no date).

Vi bruker Discord som kommunikasjonsplattform internt i gruppa. Vi har brukt Discord i andre prosjekter tidligere og mener at det har alle de funksjonene vi trenger. Selv om Discord er markedsført spesielt mot dataspill samfunn er funksjonaliteten veldig bra for andre bruksområder også.

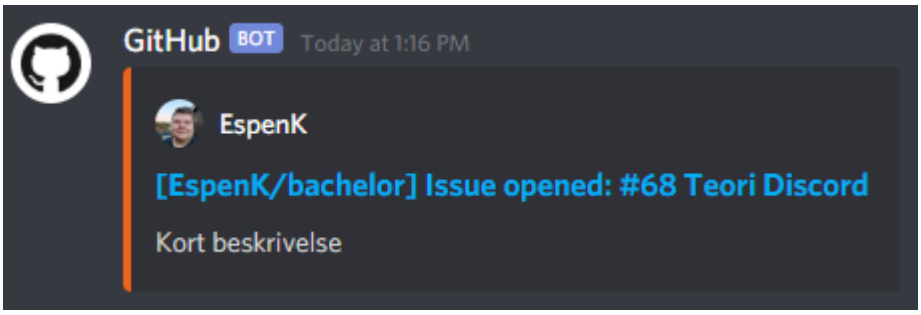
Måten Discord er bygd opp er at alle som skal bruke tjenesten må lage en brukerkonto. En bruker kan legge til andre brukere som venner og kontakte dem direkte. En av de viktigste

funksjonene til Discord er å la brukerne lage sine egne servere, der de selv setter opp kanaler, roller og regler. Her kan man ha avanserte roller som bestemmer hvilke kanaler man kan lese, skrive og koble seg til. Dette gjør at grupper kan sette opp sine egne servere på en slik måte det passer best for dem. Her kan man skrive i tekst i felles kanaler og man kan snakke i voice kanaler. Når man er i en voice kanal kan man dele skjerm med de som er i kanalen, men for møter er det mest praktisk med oppringing av en gruppe slik at man også kan ha videosamtale (Delfino, 2020).

Vi har satt opp en kanal for snakking, og to kanaler for skriving. Kanalene heter *general* og *git-spam*. *General* er der vi skriver med hverandre, og *git-spam* bruker webhook til GitHub slik at vi får oppdateringer av alt som skjer i ethvert GitHub repository som tilhører prosjektet. I tillegg til disse faste kanalene har vi også laget noen kanaler der vi skriver for eksempel spørsmål vi skal ha klare til møte med veileder, eller endringer som vi må få med når vi skriver mail til veileder.



Figur 6. Meldinger sendt med Discord i general kanalen



Figur 7. Melding automatisk sendt av GitHub til git-spam kanalen

Discord lar oss kommunisere på web, app og mobil. For å være sikre på at noen får med seg det som skrives kan man nevne en person eller gruppe med en alfakrøll før navnet. De får da et varsel internt i appen eller på mobil. Det er mulighet for å formatere tekst slik at vi kan sende kodesnutter med formatering og syntax highlighting som vist i Figur 6. I tillegg til tekst er det også mulig med snakke kanaler, men vi benytter oppringing til en gruppe i stedet. På den måten kan vi også ringe veileder. Når vi har møte over Discord kan vi bruke mikrofon, kamera og dele skjerm. Vi kan også skrive og sende små filer om det er nødvendig.

3.2.5 JetBrains IDE

Som utviklingsverktøy bruker vi IDE-er fra JetBrains (*JetBrains: Developer Tools for Professionals and Teams*, no date). Siden vi har studentlisens hos JetBrains får vi tilgang til alle deres verktøy gratis. Vi bruker derfor deres *Ultimate* eller *Professional* versjoner. Siden vi skal lage forskjellige prosjekter i forskjellige språk og rammeverk bruker vi flere ulike IDE-er. På den måten har vi det verktøyet som passer best til jobben i stedet for å prøve å gjøre alt i ett verktøy.

Alle verktøyene deler funksjonalitet som kode navigering, kode fullføring, kodeanalyse, verktøy for refaktorering, integrert støtte for git og andre versjonering verktøy, avanserte søk og så videre (*Features - IntelliJ IDEA*, no date). Vi har brukt JetBrains verktøy i mange år og føler oss komfortable med bruken. Siden alle verktøyene er relativt like både i funksjonalitet og utseende vil det ikke være noe problem å jobbe med forskjellige språk.

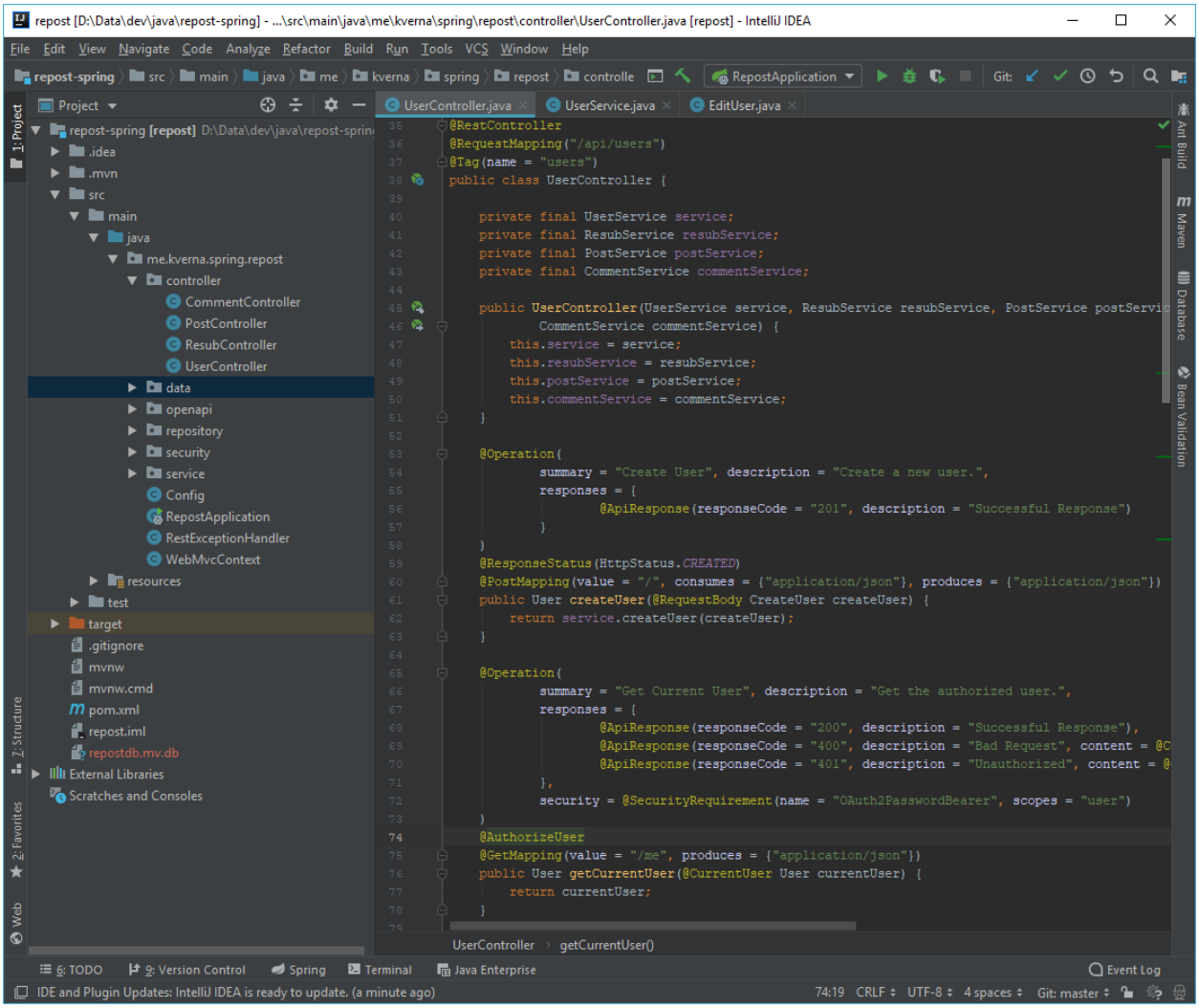
Dette er også et viktig punkt for oss siden vi sammenligner rammeverkene på flere måter, og én av tingene som skal beskrives er hvordan det er å komme i gang med prosjektet. Det vil dermed være dumt å lære et nytt verktøy, ettersom vi skal sammenligne arbeidsflyten med rammeverkene og ikke selve verktøyet som brukes for å jobbe med de. Å lære et nytt rammeverk vil skape forsinkelser i arbeidsflyten, som kan gi uklarhet på hvorvidt dette skyldes av rammeverket eller selve verktøyet vi har valgt å bruke. Dette vil ikke være et problem med JetBrains IDE-er, siden de ulike verktøyene gir samme brukeropplevelse.

Vi bruker en plugin med navn Statistic (*Statistic - Plugins | JetBrains*, no date) for å se antall linjer kode i hver implementasjon. Denne plugin blir installert på alle tre IDEene.

3.2.5.1 Java - JetBrains IntelliJ IDEA Ultimate Edition

For Java utvikling med Spring Boot bruker vi IntelliJ IDEA Ultimate Edition. Denne IDEen har integrert støtte for Spring. Vi har også installert plugin for Lombok slik at IDEen forstår hvilke metoder som blir definert automatisk av Lombok.

Ultimate Edition er lisensiert med en kommersiell lisens for personlig bruk eller bedrifter. Det finnes også en IntelliJ IDEA Community Edition som er tilgjengelig som open source under Apache 2.0 lisensen. Community versjonen har mindre funksjonalitet, men hovedfunksjonaliteten er der. Det meste som ekskluderes er kompatibelt med andre verktøy og rammeverk (*Features - IntelliJ IDEA*, no date).

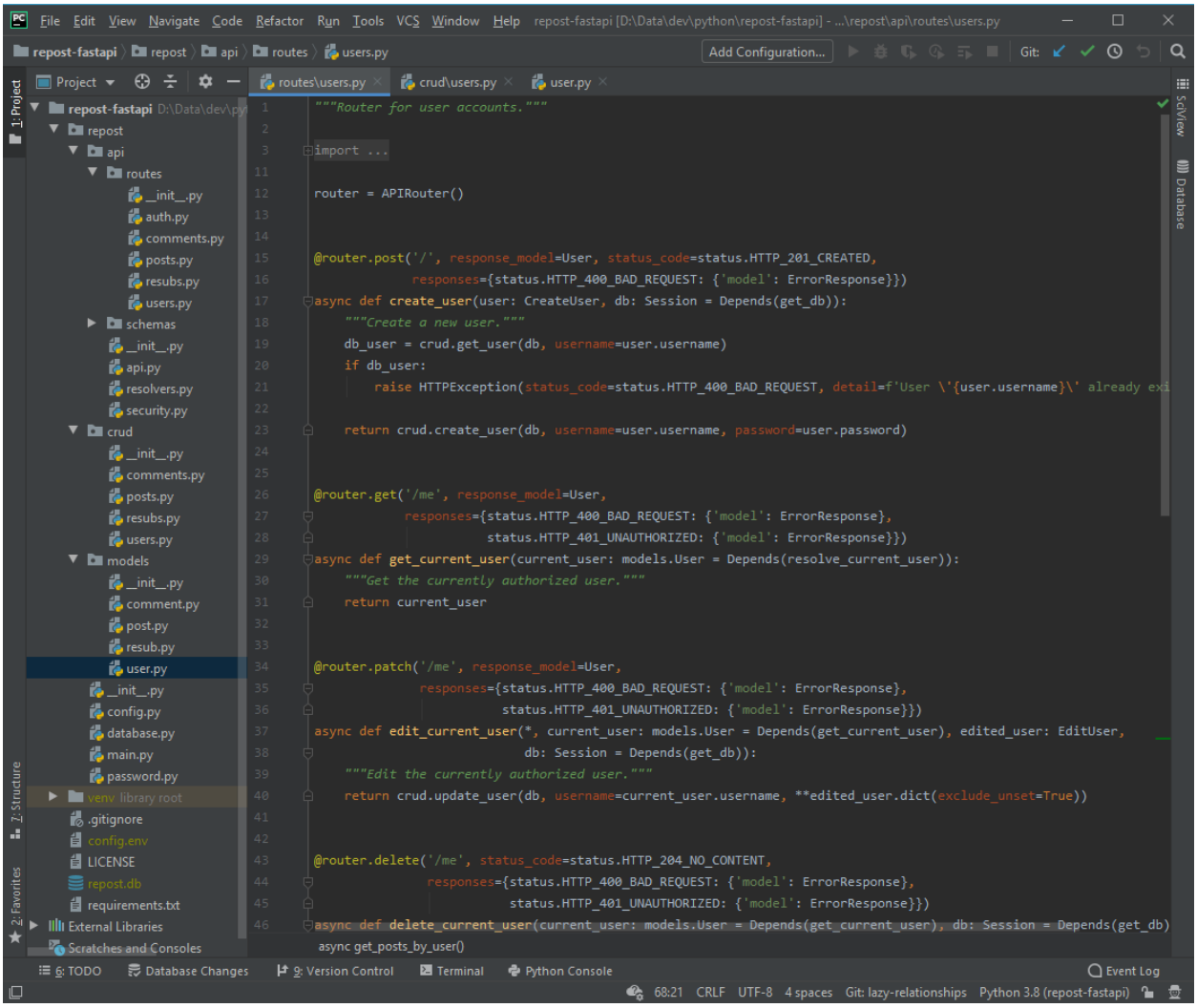


Figur 8. JetBrains IntelliJ IDEA Ultimate Edition

3.2.5.2 Python - JetBrains PyCharm Professional

For Python utvikling med FastAPI bruker vi PyCharm Professional. Det er ingen dedikert støtte for FastAPI, men FastAPI har støtte for at IDEen kan vise hvilken funksjonalitet som er tilgjengelig for strukturer i rammeverket. Dette gjelder også for SQLAlchemy.

PyCharm Professional er lisensiert med en kommersiell lisens for personlig bruk eller bedrifter. Det er også en PyCharm Community Edition tilgjengelig. Denne er open source. Community versjonen har ikke støtte for andre rammeverk og verktøy (*Features - PyCharm*, no date).

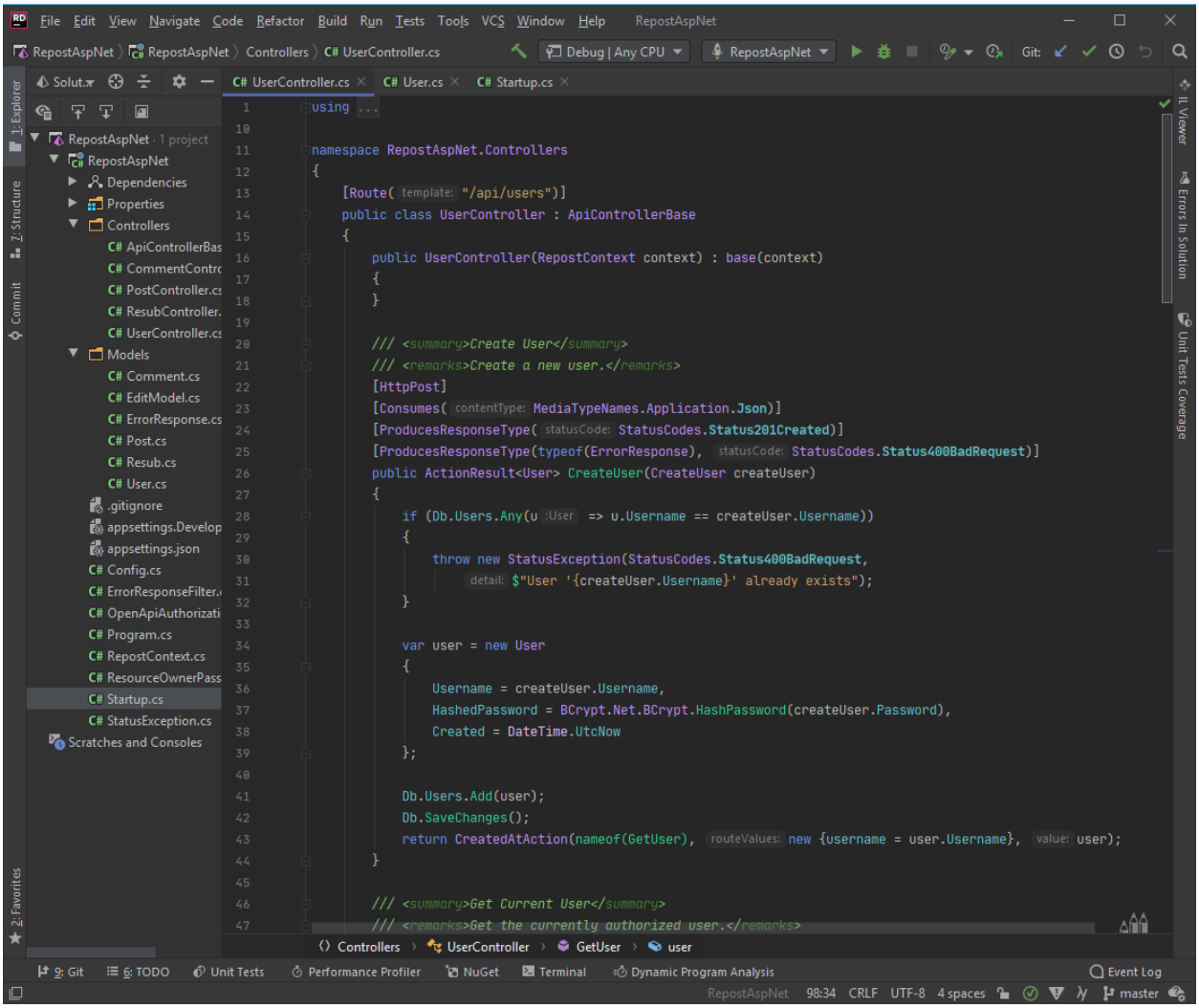


Figur 9. JetBrains PyCharm Professional

3.2.5.3 ASP.NET - JetBrains Rider

For C# utvikling med ASP.NET Core Web APIs bruker vi Rider. Denne IDEen har støtte for de verktøyene vi skal bruke.

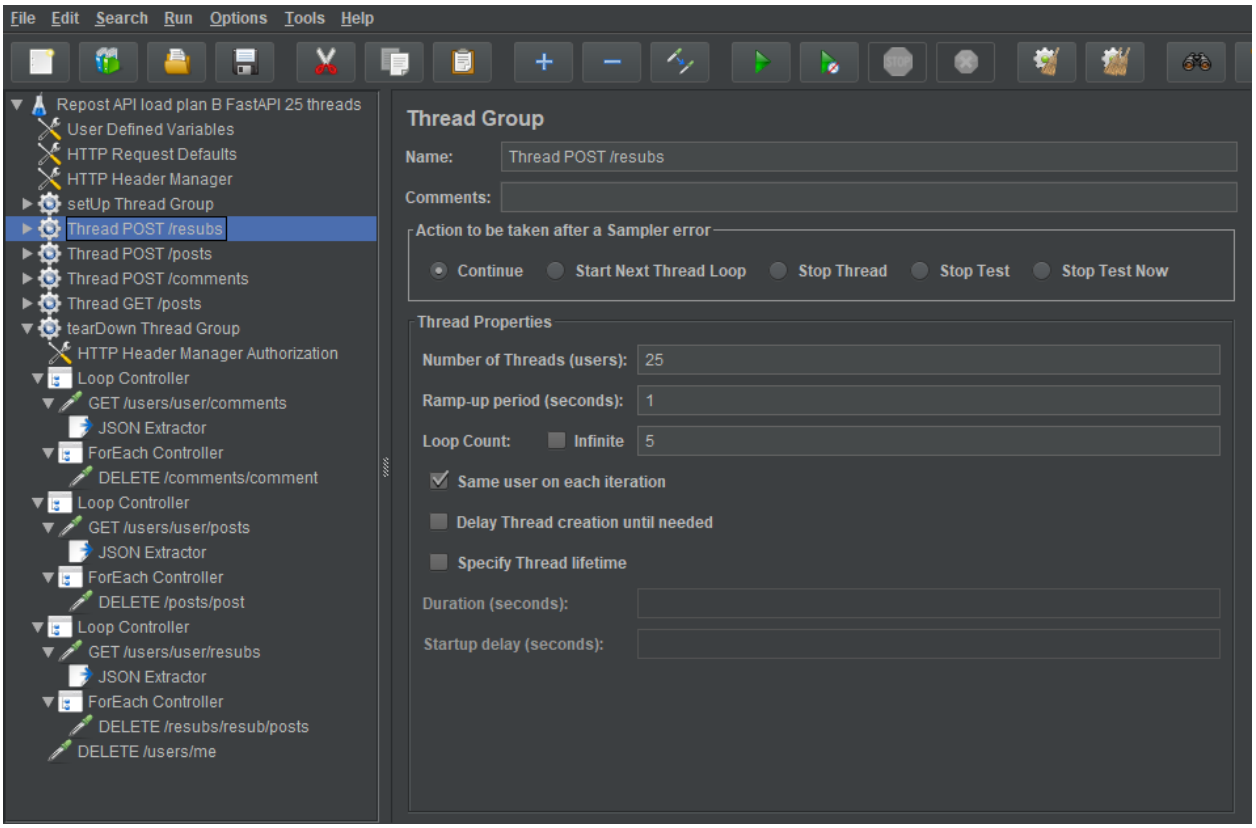
Rider er lisensiert med en kommersiell lisens for personlig bruk eller bedrifter. Det er ingen open source versjon tilgjengelig (*Features - Rider*, no date).



Figur 10. JetBrains Rider

3.2.6 JMeter testverktøy

JMeter er et open source, Apache 2.0 lisens, testverktøy laget i Java av Apache Software Foundation. Siden det er laget i Java kan det brukes i Linux, Windows og Mac OS. Verktøyet er spesielt ment for å simulere en tung last fra flere samtidige brukere og fungere med mange forskjellige protokoller. For å lage tester kan man bruke GUI og for å kjøre tester kan man kjøre det i kommandolinjen. For å kunne simulere flere samtidige brukere er JMeter et multitråd program. I tillegg til å simulere last vil også verktøyet samle inn data som kan brukes til å beregne ytelse (*Apache JMeter - Apache JMeter™*, no date).



Figur 11. Apache JMeter

For å lage testplaner som er tilnærmet reelt bruk har vi tatt utgangspunkt i Reddit sin 2019 statistikk rapport der de går gjennom diverse tall og høydepunkter. Her viser de til at de har 430 millioner brukere, 190 millioner poster, 1700 millioner kommentarer og 32 milliarder stemmer (Murphy, 2019).

Med utgangspunkt i denne dataen ser vi at ikke alle brukere lager poster. Det er relativt få som lager poster, flere som lager kommentarer og veldig mange som bruker stemmer på poster og kommentarer. Vi ønsker å lage noen test planer der vi bruker dette som basis og prøver å simulere et slikt bruksmønster.

For å kunne visualisere resultatene og få fram nøkkeltall bruker vi JMeter til å generere *Report Dashboard* (*Apache JMeter - User's Manual: Generating Dashboard Report*, no date). Vi bruker standardinnstillingene for generatoren, med unntak av én endring. Vi har valgt å bruke 1 sekund i stedet for 60 sekunder som standard intervall på testene. På den måten er det enklere å se trender i resultatene siden vi har mange flere datapunkter.

3.3 Konsepter i våre API-er

Dette kapittelet beskriver ulike konsepter og programvare som skal brukes i alle API-ene.

3.3.1 Påloggingsflyt

Vi har valgt å implementere autentisering i API-et fremfor å bruke en tredjepart til autentisering. For autentisering har vi behov for en måte å spørre om påloggingsinformasjon fra brukeren. Ved bruk av OAuth2 spesifikasjonen kan vi løse dette med en password flow. Autentisering kan da utføres gjennom brukernavn og passord sendt av brukeren til klienten. Lokalt kjører vi altså en OAuth2 autentiseringsserver, som i vår API vil være et eget endepunkt */auth/token*. Dette endepunktet svarer med en OAuth2 token, videre i denne rapporten kalt autentiserings token, som kan videre brukes til autorisering.

Token modellen vi bruker er JWT. De gir en kryptert “nøkkel” som inneholder all informasjonen vi trenger til et sikkert autoriseringssystem. Vi har en secret lagret på serveren, som må brukes til både kryptering og dekryptering av våre tokens. På denne måten er det kun autentiseringsserveren vår som kan levere tokens i API-et, siden andre parter ikke vil ha denne secreten.

Enhver token sin data består av brukernavnet til brukeren som token er utgitt til og utløpsdato. Disse er lagret i feltene “sub” og “exp” i datadelen av JWT. Samtidig er en kryptert versjon av signaturen lagret i token slik at server kan godkjenne token. En slik token gir da altså informasjon om hvilken bruker som skal utføre autorisering senere, og om brukerens token fortsatt er gyldig. Dermed får vi en enkel og sikker måte for autorisering etter brukeren er autentisert av serveren gjennom OAuth2 password flow.

For å autorisere brukeren bruker vi da også OAuth2 sin flyt av bearer tokens med JWT som blir gitt av autentiseringsserveren. Når en bruker skal autorisere seg må JWT sendes som en bearer i en HTTP header. Denne flyten passer godt til vår API siden den skal være tilstandsløs. Det er da ingen behov for noe sesjon på serveren og en tokens gyldighet er kun definert av secret lagret i server og utløpsdato. Til slutt blir det ingen data som må lagres eller holdes i minne for å opprettholde pålogging av brukeren.

I tillegg skal vi bruke en OAuth2 client og scopes for å teste kompatibilitet. Da skal det også være mulig å bruke client secret, slik at man får et ekstra lag med beskyttelse på endepunkt som krever autorisering. I vårt prosjekt vil ikke en secret gi mening til denne OAuth2 klienten, ettersom vi da måtte ha lagret den i nettsiden som er skrevet i JavaScript. Dermed hadde secret vært synlig for offentligheten via JavaScript koden på nettsiden. Samtidig skal en scope bli brukt til brukerrettigheter. Da blir det lagt opp til at andre scopes kan bli brukt senere om nødvendig.

3.3.2 PostgreSQL

PostgreSQL er en open source, PostgreSQL Licence lisens, relasjonsdatabase som bygger på SQL. PostgreSQL har alle funksjonene som man kan forvente av en relasjonsdatabase og er veldig populært for mange utviklere og organisasjoner (*PostgreSQL: About*, no date).

Vi valgte å bruke PostgreSQL på grunn av at det er open source, veldig enkelt å sette opp og det er god dokumentasjon. I tillegg til god dokumentasjon gjør en stor brukerbase det enkelt å finne løsninger på vanlige problemer.

3.3.3 OpenAPI

Siden OpenAPI er en veldig utbredt standard med mange verktøy som kan integreres med de rammeverkene vi skal bruke er det naturlig for oss å ta det i bruk.

OpenAPI har verktøy for flere steg i utviklingen, men vi har vi valgt å bare bruke dokumentasjonsverktøy. Vi har skrevet koden selv og generert dokumentasjonen ut i fra den funksjonaliteten vi har laget. Enten er denne funksjonaliteten bygget inn i rammeverket eller så bruker vi en avhengighet som gir oss denne muligheten. API spesifikasjonen som blir generert kan brukes videre av andre OpenAPI verktøy.

3.3.4 Swagger UI

For å kunne vise informasjonen som er i OpenAPI dokumentasjonen bruker vi Swagger UI. Dette er et open source, Apache 2.0 lisens, verktøy som er kompatibel med OpenAPI og som viser dokumentasjonen som en interaktiv nettside. Informasjonen blir presentert på en oversiktlig måte som gjør det enkelt for mennesker å få oversikt. I tillegg er det mulig å bruke denne nettsiden for å gjøre en forespørsel til APIen, også med autentisering, slik at man kan se om API-et fungerer som forventet (*REST API Documentation Tool | Swagger UI | Swagger*, no date).

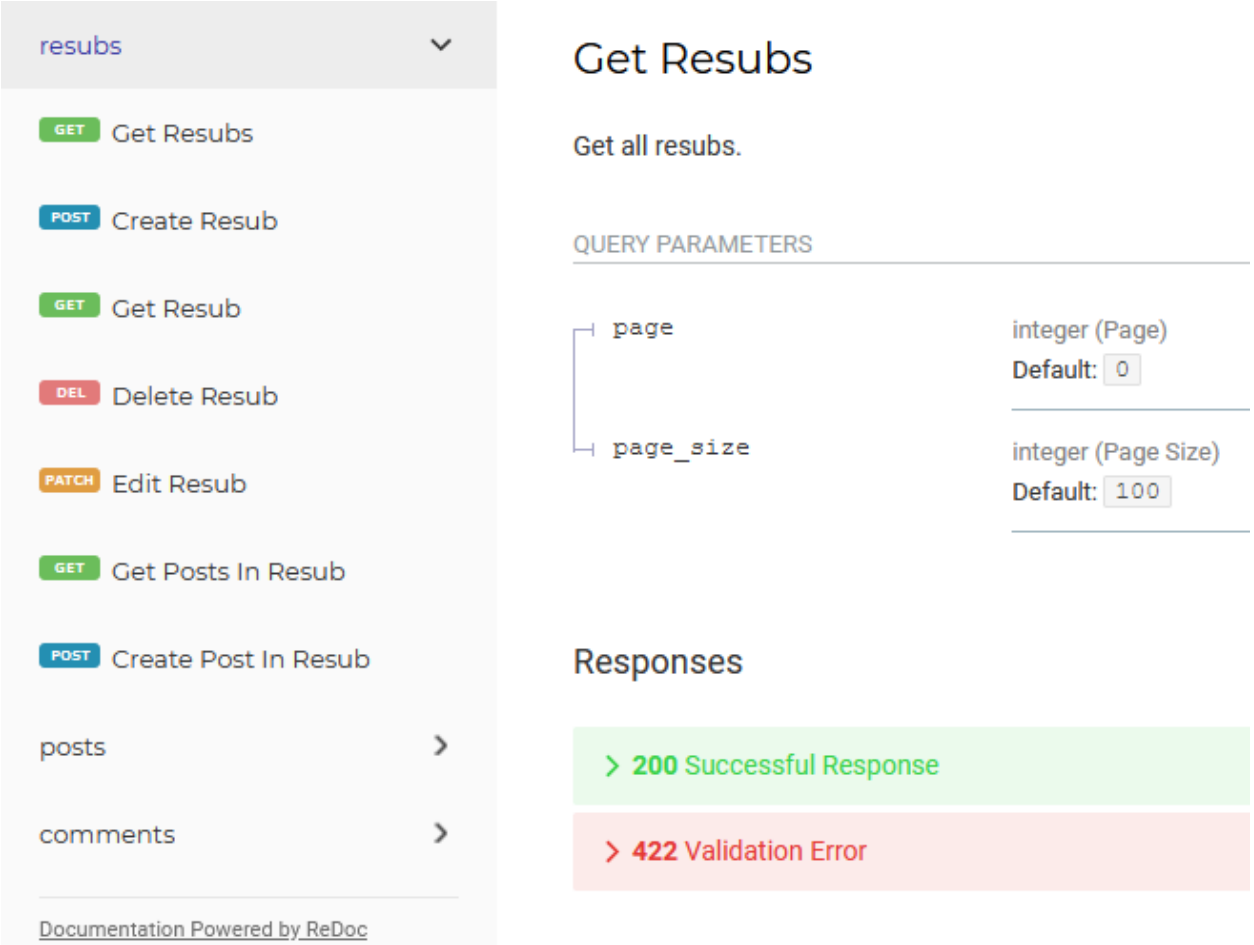
Denne nettsiden blir satt opp av de samme avhengighetene som generer dokumentasjonen.

Figur 12. Endepunkter til API-et i Swagger UI

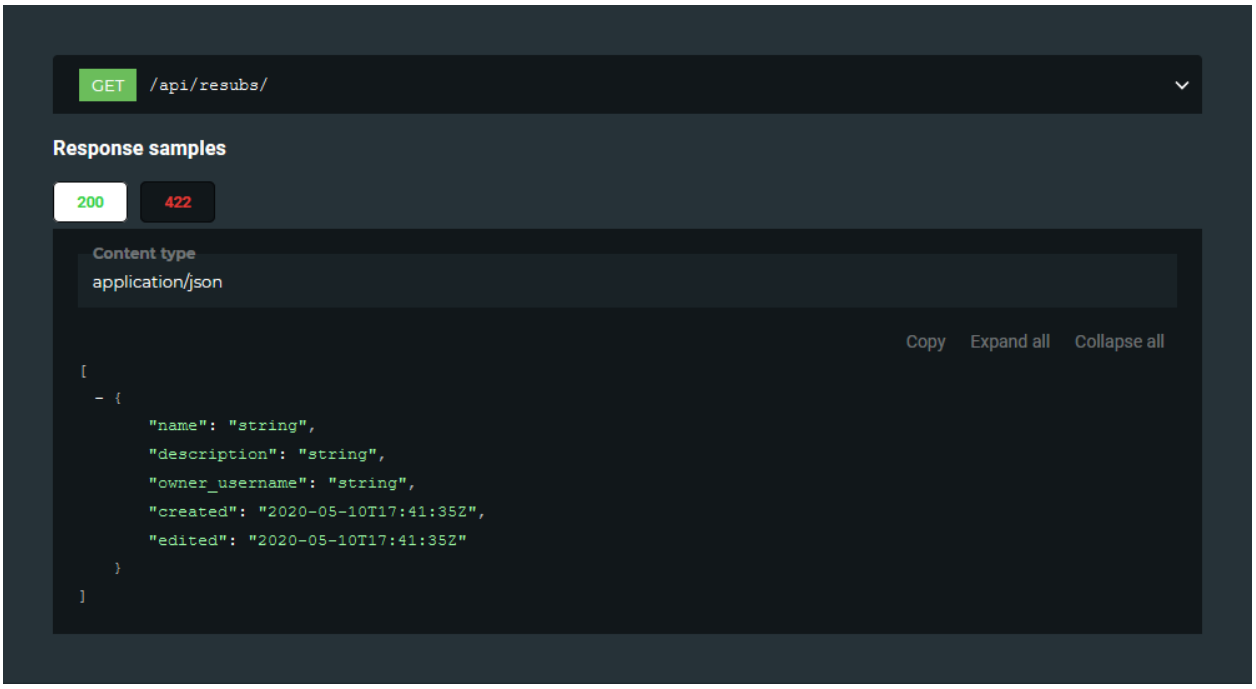
3.3.5 ReDoc

ReDoc er et annet open source, MIT lisens, verktøy som viser OpenAPI dokumentasjon som en interaktiv nettside. Dette verktøyet har mange av de samme funksjonene som Swagger UI, men mangler den interaktive delen og kan altså ikke gjøre en forespørsel til APIen. ReDoc viser den samme informasjonen, men det er lagt opp på en annen måte, og det skal være mobilvennlig visning (Redocly, no date).

Vi har ikke brukt ReDoc i særlig grad, men FastAPI setter opp både Swagger UI og ReDoc automatisk så vi vil inkludere det her.



Figur 13. Endepunkter til API-et i ReDoc



Figur 14. Visning av et spesifikt endepunkt i ReDoc

3.3.6 Testing av løsningene

Ettersom vi skal implementere samme API i tre forskjellige rammeverk vil det være nødvendig å kjøre tester for å forsikre at alle implementasjonene er like. Vi har valgt å kjøre testene på selve API-et, slik at de kan gjenbrukes i alle implementasjoner. Det blir altså en black-box test (2.7.1) hvor testesystemet er frakoblet implementasjonsdetaljene, og trenger ikke å vite noe om de ulike programmeringsspråkene eller rammeverkene som er brukt.

Hensikten med testene er å sjekke at de tre løsningene er like. Derfor er det ikke så viktig å kunne teste komponenter hver for seg, slik som i en white-box test (2.7.2). Dette gjør at vi kan velge en enkel testmetode som gjør forespørsler til API-et i kronologisk rekkefølge. Løpet av disse API kallene må bli lagt opp slik at alt som krever referanser blir laget og testet først, og helt til slutt kan alt slettes igjen. Testene må omfatte alle mulige inputs og outputs fra APIen. Det betyr at alle endepunkt som krever autorisering må testes med og uten gyldig autentiserings token, endepunkt som krever spesielle rettigheter må testes med forskjellige brukere, gyldige og ugyldige spørringer må prøves.

For testverktøy har vi valgt å lage et enkelt Python skript som bruker Requests biblioteket. Skriptet kjøres med en URL som parameter, og vil kjøre en full gjennomgang av API-et. Ved feil i et endepunkt skal skriptet stoppe og vise forskjell på forventet og mottatt verdi. Skriptet skal altså kun brukes mot slutten av implementasjon av et API, ettersom den vil stoppe når den møter en feil. Samtidig vil skriptet kunne brukes om vi må endre noe i API-et senere.

3.3.6.1 Requests

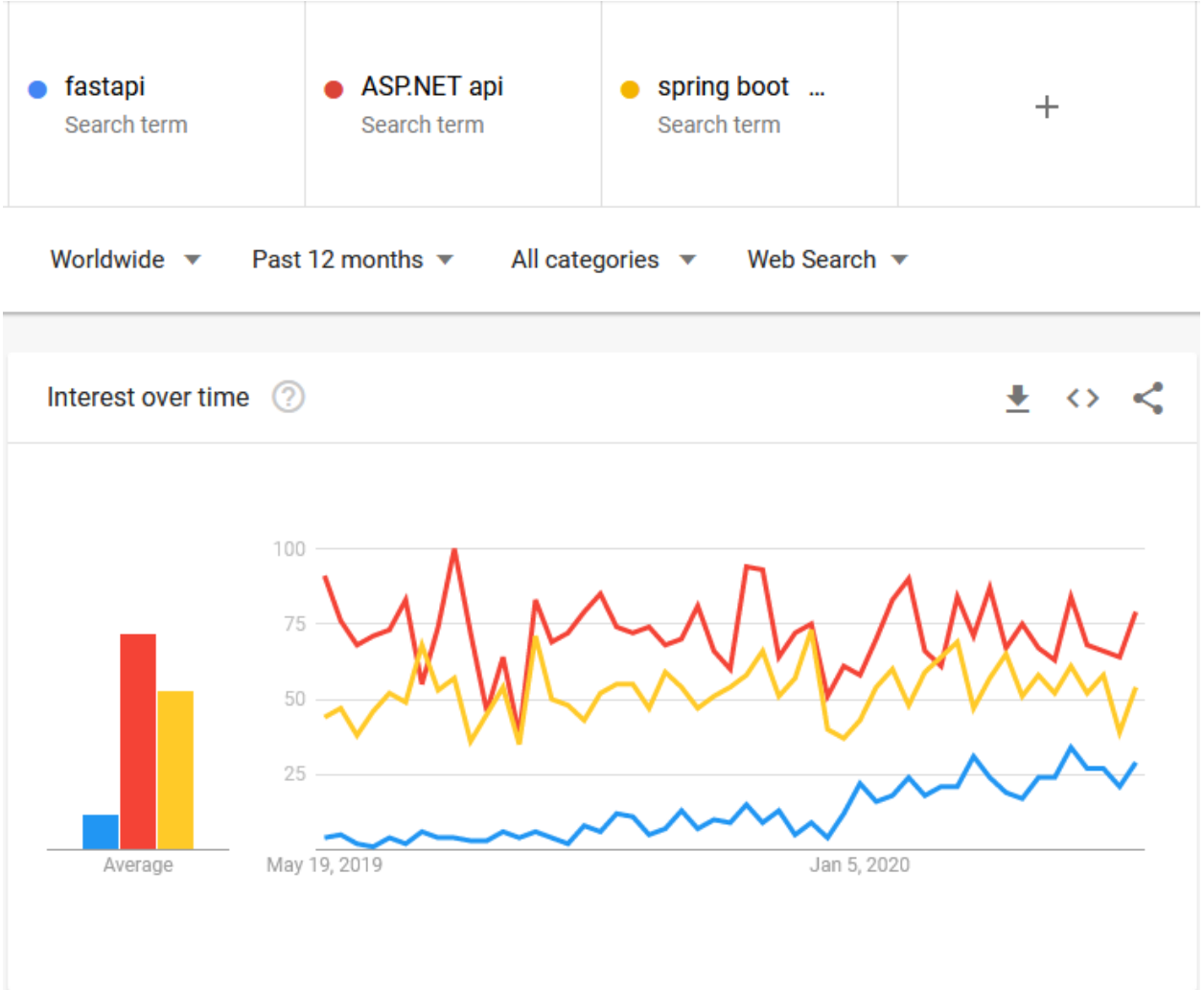
Requests er et open source, Apache 2.0 lisens, Python bibliotek for HTTP operasjoner. Requests er kompatibelt med Python 2 og Python 3. Biblioteket er laget for å gjøre HTTP forespørsler med enkle metoder. Det er også enkelt å hente ut resultatet. Alt fra statuskode til headere og data er tilgjengelig som en property eller gjennom en metode. På nettsiden til prosjektet er det veldig mange veiledninger, helt fra enkle forespørsler til mer avanserte bruksområder. Det er også linker til veiledninger for hvor man ellers kan finne informasjon og hjelp fra andre brukere (*Requests: HTTP for Humans™ — Requests 2.23.0 documentation*, no date).

3.4 Valg av rammeverk

I dette kapittelet beskriver vi de ulike rammeverkene vi har valgt og hvorfor vi valgte dem. Her gis også kort beskrivelse på viktige komponenter som er brukt i vår implementasjon av API-ene med disse rammeverkene.

For å kunne gjøre oss opp en mening om et rammeverk må vi ha noe å sammenligne med. Vi bestemte at om vi velger tre rammeverk vil det gi et godt grunnlag for å si noe om hvert av de, men det vil fortsatt være en overkommelig arbeidsmengde.

Grunnen til at vi valgte de rammeverkene vi gjorde er forskjellig for hver av de tre. Vi valgte Spring Boot på grunn av at vi har brukt det i tidligere oppgaver under utdanningen. Vi brukte det med MVC oppbygging i faget ID202712 i prosjektet AaIS Inventory for en skole og med API oppbygging i faget ID303911 i prosjektet kompis (pckv, no date). FastAPI leste vi om da det ble annonsert sommeren før vi skulle velge oppgave. Det virket som et veldig spennende prosjekt som vi gjerne ville prøve. Etter å ha vært i samtale med og jobbet litt for et utviklingsfirma virket det som at ASP.NET var populært i området der vi skal søke jobb. Firmaet hadde stor interesse for å gå over til ASP.NET Core ettersom det er en nyere og moderne versjon av hva de allerede hadde erfaring med. Etter vi hadde sett noen demoer var også vi interesserte i ASP.NET Core Web APIs, og vi ønsket å fordype oss mer rundt dette.



Figur 15. Sammenligning mellom rammeverkene i Google Trends (Google Trends, no date)

Søkeordene i Figur 15 er fastapi, ASP.NET api og spring boot api (*Google Trends*, no date). Dataene viser populariteten av søkeordene de siste 12 måneder i hele verden. Vi kan se at Spring Boot og ASP.NET er godt etablert mens FastAPI er relativt nytt men øker i popularitet.

3.4.1 FastAPI

FastAPI er et open source, MIT lisensiert, web API rammeverk laget av Sebastián Ramírez i Python. FastAPI er ment for å gjøre det raskt og enkelt å bygge web API-er som følger OpenAPI standarder. Det bygger på andre raske open source Python prosjekter, noe de mener gjør det til et av de raskeste Python rammeverkene, samtidig som det er et av de enkleste og raskeste Python rammeverkene å utvikle i (*FastAPI*, no date).

For å komme i gang er det et avsnitt for installasjon (*Tutorial - User Guide - Intro - FastAPI*, no date), dermed er det et enkelt eksempel på hvordan man lager endepunkt og starter tjenesten. Siden Python ikke krever en main metode er det veldig få linjer kode som skal til før man har et

fungerende eksempel. Siden FastAPI er kompatibelt med OpenAPI har det også automatisk generasjon av OpenAPI spesifikasjoner, og det er satt opp Swagger UI og ReDoc for å visualisere dette.

3.4.1.1 SQLAlchemy

SQLAlchemy er et open source, MIT lisens, verktøy for database operasjoner og Object Relational Mapping (ORM) i Python. Verktøyet har veldig mange nyttige funksjoner og gjør det enkelt å gjøre database operasjoner uten å måtte skrive SQL spørringer manuelt. På grunn av ORM kan også tabeller og relationships bli håndtert av SQLAlchemy (*Features - SQLAlchemy*, no date).

I tillegg til at det er enkelt å skrive kode som gjør spørringene vi trenger kan det brukes med mange forskjellige databaser. Vi trenger ikke å skrive om koden om vi skal bruke en ny database. Det eneste som må forandres er database koblingen og eventuelle innstillinger for koblingen. Koden blir den samme. Dette gjør det veldig enkelt å komme i gang med prototyping siden man kan benytte seg av embedded databaser når man setter i gang. En ordentlig database kan settes opp senere i utviklingen.

3.4.1.2 Pydantic

Pydantic er et open source, MIT lisens, verktøy for datavalidering i Python. Det er raskt og ser ut som vanlig Python, dermed er det liten læringskurve for de som kjenne Python (*pydantic*, no date).

FastAPI bruker Pydantic modeller for API skjema. Disse modellene kan mappes til en SQLAlchemy modell. På den måten kan vi hente ut et objekt fra database som SQLAlchemy modell og returnere det til brukeren som en Pydantic modell (*SQL (Relational) Databases - FastAPI*, no date).

3.4.1.3 Uvicorn

I dette prosjektet bruker vi Uvicorn til å kjøre FastAPI serveren. Uvicorn er en open source, BSD lisens, ASGI server (*Uvicorn*, no date). ASGI fungerer som et standardisert grensesnitt som Python rammeverk kan bruke for å lage en asynkron web server (*ASGI Documentation — ASGI 2.0 documentation*, no date). Ettersom FastAPI bygger på Starlette, som er et ASGI rammeverk, kan vi velge selv hvilken ASGI server vi ønsker å bruke. Vi valgte Uvicorn fordi det har stor fokus på hastighet, som det oppnår ved å bruke uvloop som event loop. uvloop er kun støttet i *"*nix platformer"* (Selivanov, 2016). Altså får man best ytelse i Linux. Uvicorn støtter både HTTP/1.1 og WebSockets, men i vårt prosjekt bruker vi bare HTTP/1.1 (*Uvicorn*, no date).

Uvicorn fungerer som et verktøy i terminalen og kan importere og bruke *app* instansen vi lager i FastAPI prosjektet. Slik kan Uvicorn og andre ASGI servere kjøre applikasjoner som er laget med ASGI standarden.

```
repost-fastapi — Konsole
(venv) $ uvicorn repost:app --host 0.0.0.0
INFO:     Started server process [14833]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000
```

Figur 16. Kjøre FastAPI med Uvicorn

3.4.1.4 Gunicorn

Gunicorn er brukt for å administrere flere prosesser av Uvicorn slik at vi kan oppnå mer bruk av CPU og få bedre ytelse. Gunicorn er en open source, MIT lisens, WSGI server for UNIX plattformer (*Gunicorn - Python WSGI HTTP Server for UNIX*, no date).

```
repost-fastapi — Konsole
(venv) $ gunicorn repost:app -b 0.0.0.0:8000 -w 3 -k uvicorn.workers.UvicornWorker
[2020-05-10 20:26:20 +0200] [15320] [INFO] Starting gunicorn 20.0.4
[2020-05-10 20:26:20 +0200] [15320] [INFO] Listening at: http://0.0.0.0:8000 (15320)
[2020-05-10 20:26:20 +0200] [15320] [INFO] Using worker: uvicorn.workers.UvicornWorker
[2020-05-10 20:26:20 +0200] [15322] [INFO] Booting worker with pid: 15322
[2020-05-10 20:26:20 +0200] [15324] [INFO] Booting worker with pid: 15324
[2020-05-10 20:26:20 +0200] [15325] [INFO] Booting worker with pid: 15325
[2020-05-10 20:26:21 +0200] [15322] [INFO] Started server process [15322]
[2020-05-10 20:26:21 +0200] [15322] [INFO] Waiting for application startup.
[2020-05-10 20:26:21 +0200] [15322] [INFO] Application startup complete.
[2020-05-10 20:26:21 +0200] [15324] [INFO] Started server process [15324]
[2020-05-10 20:26:21 +0200] [15324] [INFO] Waiting for application startup.
[2020-05-10 20:26:21 +0200] [15324] [INFO] Application startup complete.
[2020-05-10 20:26:21 +0200] [15325] [INFO] Started server process [15325]
[2020-05-10 20:26:21 +0200] [15325] [INFO] Waiting for application startup.
[2020-05-10 20:26:21 +0200] [15325] [INFO] Application startup complete.
```

Figur 17. Kjøre FastAPI med Gunicorn og 3 Uvicorn workers

3.4.1.5 Avhengigheter

Avhengigheter fra tredjepartsbiblioteker blir styrt ved bruk av The Python Package Installer (pip) verktøyet. Dette verktøyet installerer samfunnsbaserte biblioteker fra Python Package Index (PyPI) (Python Packaging Authority, 2020). pip har støtte for å lese fra en fil, vanligvis ved navn *requirements.txt*, som beskriver alle bibliotekene krevd av prosjektet sammen med versjonene som kreves for bibliotekene. Nedenfor er en kort oppsummering av bibliotekene vi bruker i FastAPI prosjektet som ikke er beskrevet tidligere.

psycopg2 Adapter laget for å koble til PostgreSQL databaser (Di Gregorio and Varrazzo, no date). Denne brukes internt av SQLAlchemy.

ujson ujson er en raskere versjon av den innebygde JSON koderen i Python (*ujson*, no date). FastAPI anbefaler å bruke denne for bedre ytelse.

pyjwt Bibliotek for å opprette og verifisere JWT i Python (*Welcome to PyJWT — PyJWT 1.7.1 documentation*, no date).

python-dotenv Bibliotek som leser variabler fra en *.env* fil og legger dem til som miljøvariabler (Kumar, 2020).

python-multipart Bibliotek som kreves av FastAPI for å lese multipart form-data (*Python-Multipart — python-multipart 0.0.1 documentation*, no date). Internt brukes dette til lesing av OAuth2 skjema.

passlib Passord hashing bibliotek for Python med støtte for over 30 forskjellige algoritmer (*Passlib 1.7.2 documentation — Passlib v1.7.2 Documentation*, no date). Vi bruker spesifikt *passlib[bcrypt]* pakken for å installere kun BCrypt algoritmene.

3.4.2 Spring Boot

Spring Boot er et open source, Apache 2.0 lisens, rammeverk for å konfigurere og bruke andre Spring rammeverk i en egen applikasjon. Spring Boot gjør det enklere å komme i gang med utviklingen uten å sette seg inn i alle detaljene rundt Spring. Spring Boot pakker med seg mye funksjonalitet, men det er også mulig å legge til andre avhengigheter om man trenger andre verktøy. Vi skal bruke komponenter til å lage et tilstandsløst API (*Spring Boot*, no date).

For å komme i gang med et Spring Boot prosjekt bruker man gjerne Spring Initializr, et verktøy for å velge innstillinger og ekstra pakker for prosjektet (spring-io, 2020). Det finnes også et innebygget verktøy for dette i IntelliJ IDEA.

3.4.2.1 Apache Maven

Apache Maven er et open source, Apache 2.0 lisens, verktøy for å bygge og håndtere Java-baserte prosjekter. Målet med prosjektet er å gjøre det enklere for et utvikler å bygge sitt prosjekt og legge inn avhengigheter uten å ha full forståelse for de underliggende mekanismene. Siden Maven standardiserer prosessen vil alle Maven prosjekter bruke de samme kommandoene (*Maven – Introduction*, 2020).

3.4.2.2 Lombok

Lombok (*Project Lombok*, no date) er et open source, MIT lisens, Java verktøy for å unngå repeterende kode i Java prosjekter. Ved å bruke Lombok er det veldig enkelt å lage data klasser, dvs. klasser som kun benyttes til overføring og lagring av data. Lombok bruker annotasjoner for å sette egenskaper på en klasse, eller felt og metoder i klassen, og mye kode blir automatisk generert. Noe av det Lombok tar seg av er constructor, getter og setter, toString og equals. Dette gjør at koden er mer oversiktlig, ettersom standardfunksjonalitet blir generert i bakgrunnen. Lombok lar oss sette mer fokus på modellering av Data (baeldung, 2020a).

3.4.2.3 Avhengigheter

Vi bruker Apache Maven sitt innebygde system for å holde styr på eksterne biblioteker som prosjektet krever. Disse blir lastet ned og installert automatisk med Maven når prosjektet bygges. Nedenfor er en oversikt over pakkene som er beskrevet i *pom.xml* fila, med unntak av Spring pakker.

springdoc-openapi springdoc er et verktøy som automatisk genererer et OpenAPI skjema basert på annotasjoner i koden. Den har også innebygd server for Swagger UI (*springdoc-openapi | Library for OpenAPI 3 with spring-boot*, no date).

h2database H2 er en embedded database driver laget i Java (*H2 Database Engine*, no date). Embedded betyr her at den lagrer databasen lokalt til en fil. Vi bruker dette under utvikling slik at vi kan teste Spring Boot løsningen uten en ekstern database server.

postgresql postgresql pakken refererer til PostgreSQL JDBC, som er en PostgreSQL database driver for Java (*PostgreSQL JDBC About*, no date). Denne kreves for å kunne koble til og kommunisere med PostgreSQL databaser i Spring Boot.

3.4.3 ASP.NET Core Web APIs

ASP.NET Core Web APIs er et open source, Apache 2.0 lisens, rammeverk for web API-er laget av Microsoft i C# og .NET Core. Siden det er laget i .NET Core er det kompatibelt med alle moderne operativsystemer (*What is ASP.NET Core? A cross-platform web-development framework*, no date). .NET Core er del av den nyeste implementasjonen av Microsofts .NET, og er laget for å støtte flere plattformer enn den tidligere .NET Framework som kun støtter Windows maskiner. Microsoft anbefaler derfor å bruke .NET Core for nyere prosjekter hvor det er mulig (Microsoft, 2020a).

Rammeverket gjør det veldig enkelt å komme i gang med byggingen av API-et og har enkle instrukser for dette på sin hjemmeside. Her demonstreres det hvordan man enkelt kan lage en klasse og bruke denne modellen i APIen. Videre viser de til flere høydepunkter og linker til mer utdypende artikler (*ASP.NET Web APIs | Rest API's with .NET and C#*, no date).

Microsoft har eksempler på hva man kan bygge og veiledninger på hvordan man kommer i gang med et prosjekt. Her er det lagt opp til at man bruker Microsoft sine egne verktøy for å redigere kode, men selve innholdet er overførbart til de som bruker andre verktøy. Her viser de også hvordan man kan teste API-et med et eksternt verktøy (Anderson, Larkin and Wasson, 2020).

3.4.3.1 Avhengigheter

For å legge til avhengigheter i ASP.NET Core bruker vi NuGet. NuGet holder styr på alle samfunnsbaserte biblioteker som kreves til et prosjekt (*NuGet Gallery | Home*, no date). Bibliotekene er oppført i *RepostAspNet.csproj* filen, som også inkluderer versjoner av bibliotekene. Nedenfor er en liste over avhengighetene til prosjektet.

BCrypt.Net-Next	Bibliotek for hashing av passord med BCrypt algoritme (*BCrypt.Net-Next 4.0.0*, no date).
IdentityServer4	Bibliotek for autorisering og autentisering med OAuth2 (*Welcome to IdentityServer4 (latest) — IdentityServer4 1.0.0 documentation*, no date).
Npgsql	Database adapter for PostgreSQL (*Npgsql - .NET Access to PostgreSQL | Npgsql Documentation*, no date)
Swashbuckle	Bibliotek for å generere OpenAPI skjema basert på attributter og endepunktene i koden. Den inkluderer også en server for Swagger UI (*Swashbuckle.AspNetCore 5.4.1*, no date).

3.4.4 Vue.js

Vue.js er et open source, MIT lisens, rammeverk for å bygge brukergrensesnitt. Rammeverket er laget av Yuxi Evan You i JavaScript. Rammeverket er kompatibelt med alle nettlesere som har støtte for ES5. Vue.js har en hovedpakke som fokuserer på visningslaget, og et økosystem med en rekke støttende bibliotek som kan hjelpe til med å implementere store Single-page applications *(Introduction — Vue.js, no date)*.

Det er veldig enkelt å importere rammeverket til prosjektet. Vue CLI er et system som er ment for smidig Vue.js utvikling. Den hjelper til med å strukturere prosjektet og har fornuftige standard konfigurasjoner som lar utviklerne heller fokusere på å skrive applikasjonen enn å holde på med masse konfigurasjon. Den tilbyr også offisielle plugins som man kan utvide prosjektet med (*Introduction — Vue.js*, no date).

PyCharm kan integreres med Vue.js. Når man skal lage et prosjekt med vue.js, kreves det å ha node.js og vue.js plugin på forhånd. Når man da lager prosjektet, velges det vue.js som prosjekt og node.js som interpreter (*Vue.js - Help | PyCharm*, 2020).

3.4.4.1 Sammenligning mellom front-end rammeverk

I dette prosjektet ønsket vi å bruke et nytt og moderne JavaScript rammeverk. De tre mest populære vi kunne velge mellom var Vue.js, Angular og React (Goel, 2020).

De største forskjellene mellom Vue og **React** er optimalisering og hvordan front-end applikasjonen blir skrevet. React bygger på at alt skal være laget i og med hjelp av JavaScript. React bruker hva de selv kaller JSX, som er en utvidelse av JavaScript hvor HTML-lignende strukturer og CSS kan bli definert i selve koden. Dersom man ønsker å endre innholdet i siden må dette eksplisitt bli varslet om i koden, i forhold til Vue hvor det oppdateres automatisk (*Comparison with Other Frameworks — Vue.js*, no date).

Dette er første gang vi bruker et JavaScript rammeverk. Vi er derimot erfaren med HTML og CSS fra tidligere prosjekter, så det virket mer naturlig å velge Vue ovenfor React på grunn av det sannsynligvis gjør utviklingen lettere ettersom Vue bruker tradisjonelt vanlig HTML og CSS. I følge Vue selv gjør dette at man får større fokus på å bygge selve applikasjonen (*Comparison with Other Frameworks — Vue.js*, no date).

Angular er basert på TypeScript, men Vue har det valgfritt. Minste pakkestørrelse til Angular er ca 65 KB, i forhold til Vue som er nærmere 30 KB. Vue er mer fleksibel med koden sin, og tilbyr mer frihet til å hvordan applikasjonen blir kodet i forhold til Angular. I følge sammenligningen til Vue sin nettside er Angular også mye vanskeligere å lære siden den krever at utvikleren skulle kunne flere konsepter med Angular før de blir produktive (*Comparison with Other Frameworks — Vue.js*, no date).

Vi bestemte at å lære TypeScript er unødvendig for fokuset i prosjektet vårt, så vi forventet å spare tid ved å bruke Vue istedenfor. For oss var pakkestørrelse irrelevant fordi dette ikke har noen påvirkning på selve demoen eller testene. Siden Vue er mer fleksibel, kan det bli satt av mer tid til utviklingen av nye funksjoner istedenfor struktur. Det ble også lettere for oss med frihet til hvordan front-end applikasjonen ble kodet, istedenfor å følge en spesifikk metode.

3.4.4.2 Nginx

Nginx er en open source, BSD lisens, HTTP server som brukes til å tjene statiske nettsidefiler. Den kan også brukes som en omvendt proxy, som vil si at den kan sende applikasjoner på forskjellige porter ut gjennom én og samme port (*Welcome to NGINX Wiki!* | *NGINX*, no date). Nginx er brukt i testoppsettet for å tjene HTML, CSS og JavaScript filene som blir bygget med Vue.js.

3.4.4.3 Avhengigheter

For å håndtere eksterne biblioteker i Vue.js brukes Node package manager (npm) (*npm | build amazing things*, no date). Bibliotekene er definert i *packages.json* filen, hvor versjonene også er inkludert. Nedenfor er en liste over bibliotekene som brukes i Vue.js prosjektet.

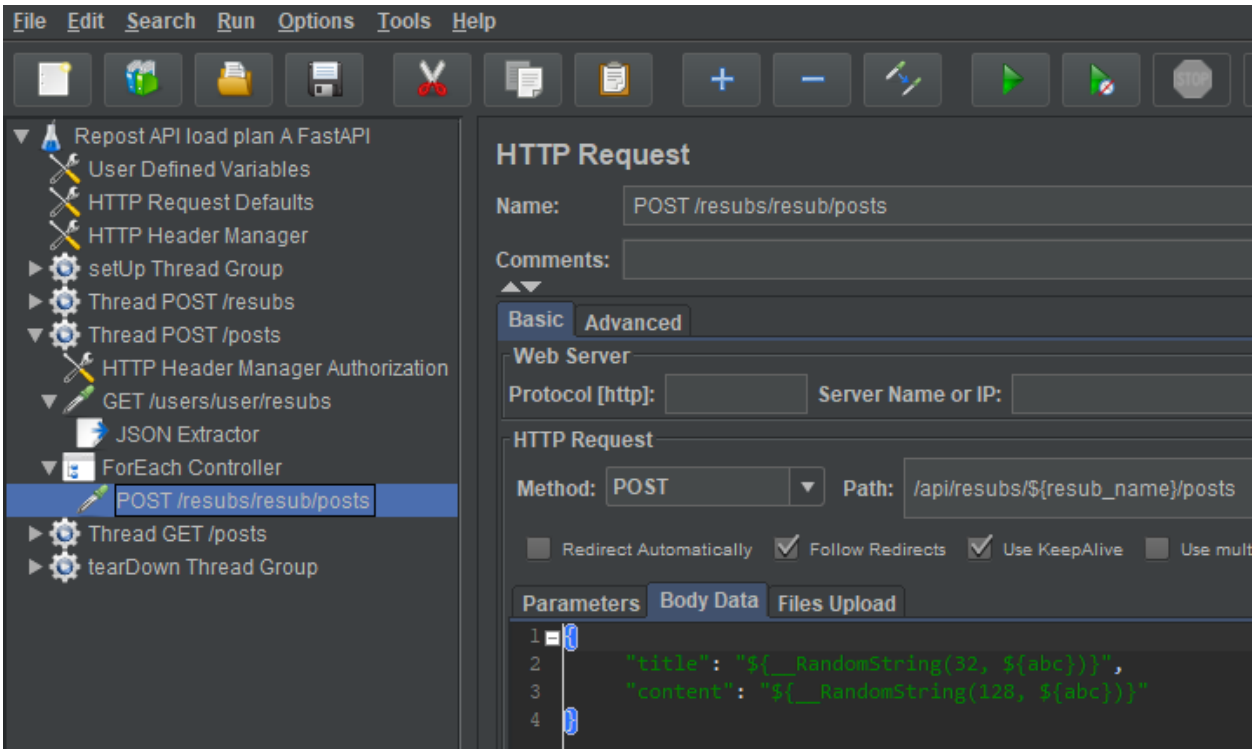
qs	Bibliotek for å gjøre om JavaScript objekter til query parameter og form data (*npm: qs*, no date). Det brukes til å enkelt sende OAuth2 skjema til API-et.
axios	Bibliotek for enklere bruk av HTTP metoder i JavaScript (*npm: axios*, no date). I dette prosjektet er det brukt til valg av API og automatisk tillegg av autentiserings token.
nprogress	Bibliotek som legger til en fremdriftsindikator på toppen av siden (*npm: qs*, no date, *npm: nprogress*, no date). Denne brukes til å vise nedlasting av elementer.

3.5 Ytelsestest med JMeter

3.5.1 Lage en testplan

For å lage en test plan bruker vi JMeter med GUI. I planen kan man legge generell konfigurasjon som adresse og port til APIen. For at testen skal gjøre noe må vi ha en tråd-gruppe. En forespørsel er en oppgave som utføres, og den må ligge i en tråd-gruppe. Vi bruker kun HTTP forespørsler for å kommunisere med vår API. For å se resultater mens man lager testen kan det legges inn forskjellige visninger som sammendrag, tabeller og grafer. Disse visningene skal bare

brukes under utvikling og fjernes før testen skal kjøres på serveren. Resultatene fra serveren vil bli lagret og vist på en annen måte.



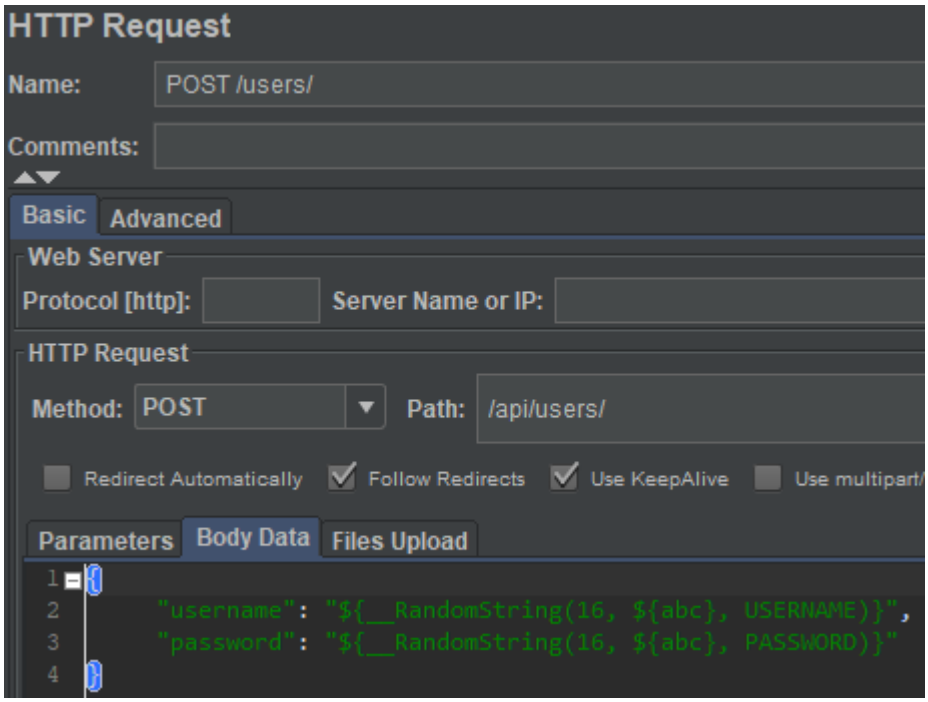
Figur 18. JMeter GUI som viser en POST forespørsel i tråd-gruppen Thread POST /posts

Som første steg i test planen lager vi konfigurasjoner som kan brukes videre i planen. *User Defined Variables* brukes for å lage variabler som kan brukes både i konfigurasjonen og videre i testen. Videre har vi en *HTTP Request Defaults* konfigurasjon der vi setter protokoll, adresse og port for APIen. I alle testene vi lager bruker vi addressen *127.0.0.1* siden vi kjører testene fra samme server som API-ene kjører på. I *HTTP Header Manager* setter vi media typen til *application/json*.

Når vi lager en tråd-gruppe har vi noen valg. Vi kan velge hvor mange tråder som skal kjøre, altså hvor mange brukere vi skal simulere. Det er også mulig å sette et tidsrom for når alle trådene skal kjøres i gang. Om man skal simulere 10 brukere og har en oppstartstid på 10 sekunder vil det bli laget én tråd per sekund. Med mindre annet er spesifisert vil alle våre tester bruke 1 sekund oppstartstid. Vi kan også velge hvor mange runder den tråd-gruppen skal kjøre. Antall runder bestemmes for hver enkelt test.

Alle testplanene har en *setUp* tråd-gruppe der vi lager en bruker med tilfeldig navn og passord. For å lage tilfeldige strenger bruker vi *RandomString (Apache JMeter - User's Manual: Functions and Variables, no date)*. Vi gjør en forespørsel for å lage en bruker med disse verdiene. Deretter bruker vi verdiene til å få en token som skal brukes videre i alle forespørsler i testen. Token henter vi ut fra responsen med en *JSON Extractor* (*Apache JMeter - User's Manual: Component Reference*, no date). Token og brukernavnet trengs videre i andre tråd-grupper. Vi bruker derfor

et BeanShell (beanshell, no date) skript for å lagre verdiene som en property slik at vi kan gjenbruke de senere. Vi setter properties med *${__setProperty(USERNAME, ${USERNAME})}* og kan bruke de senere i testen med *${__property(USERNAME)}* (Kashlach, 2012).



Figur 19. JMeter GUI som viser at brukernavn og passord til testing er tilfeldig generert tekst

I en tråd-gruppe må vi sette opp oppgaver som skal utføres. En tråd i en tråd-gruppe simulerer en bruker. Her er det HTTP spørringer som er det enkleste å sette opp, men vi kan også bruke andre funksjoner for å gjøre mer avanserte handlinger. Vi kan for eksempel gjøre en forespørsel for å få alle postene i en bestemt resub og bruke en *JSON Extractor* for å hente ut ID for alle postene. Videre kan vi bruke en *forEach* logisk kontroller som går gjennom alle post ID-ene og vi kan lage kommentarer til hver enkelt post. På den måten kan vi gjøre mer utfyllende tester.

Alle endepunkter som returnerer en liste med objekter har en begrensning på antallet de kan returnere. Standard begrensning er 100 objekter. Med mindre det er nevnt i en testplan eller en spesifikk forespørsel har vi ikke satt noen annen begrensning på dette.

Alle testplanene bruker også en *tearDown* tråd-gruppe der vi rydder opp alt sammen. Her bruker vi GET for å hente alle kommentarer, poster og resuber som vi har laget med test brukeren og sletter disse. Dette skjer med en tråd siden vi kun kan slette noe én gang og vi må gjøre det i riktig rekkefølge.

3.5.2 Oppsett av resultater

Alle testplaner vil beskrives nedenfor i dette kapittelet. Resultatene av å kjøre testplanen vil stå i resultatkapittelet. Noen av testene vil være nesten like, med endringer på noen få variabler for å se om det har noen effekt. Vil vil legge ved bilder av grafer og tabeller der det er naturlig. Om vi

endrer noen verdier i en test men ikke får store variasjoner kan vi velge å bare forklare endringen i stedet for å legge ved alle bilder på nytt. Alle rapportene blir lagt med i vedlegg 5.

3.5.2.1 Oversikt

Oversikten viser et sammendrag av alle forespørsler som er gjort i denne testplanen. For at det skal være leselige resultater har vi satt label på alle forespørslene som viser HTTP metoden som er brukt, og hvilket endepunkt den er rettet mot. Merk at selv om navnet er GET */users/user* så er den egentlige forespørselen for eksempel GET */api/user/di3nc8elgu4jd00r*.

Før vi ser på tallene for ytelse er det viktig at vi ser på antall samples, eller forespørsler, til hvert endepunkt. Alle tallene vises per endepunkt og totalt for alle forespørslene. Vi ser mest på tallene for de endepunktene vi er ute etter, men totalt gir også et greit estimat. Merk at estimat også teller med DELETE endepunktene som blir kjørt på en tråd av JMeter. Response Times viser forskjellige mål på responstiden i millisekunder, og throughput viser antall forespørsler per sekund. En lav responstid med høyt antall forespørsler per sekund er det som er ønskelig.

3.5.2.2 Forespørsler per sekund per responskode

En graf som viser forespørsler per sekund etter hvilken statuskode serveren svarer med. Siden vi har forskjellige koder for forskjellige metoder vil dette gi en pekepinn på hvilke operasjoner som er raskest og hvordan det utvikler seg over tid. Her ønsker vi selvfølgelig å se et høyt tall, men også at det er relativt stabilt.

Husk at en test har et bestemt antall forespørsler. Grafer kan se forskjellige ut på grunn av at tidsaksen tilpasser seg til den tiden det tar å gjennomføre testen på det bestemte rammeverket. Studer tidsaksen for å se etter merkeligheter.

Fargene på grafen er forskjellig for forskjellige statuskoder. Gul er for statuskode 200, eller en suksessfull GET. Lys blå er for statuskode 201, eller en suksessfull POST. Rød er statuskode 204, eller en suksessfull DELETE.

3.5.2.3 Distribusjon av responstid

Et stolpediagram som viser distribusjonen av responstid. Her er det best med minst mulig forespørsler som har lang responstid. Om alle responsene er gruppert til venstre i diagrammet har vi få forskjeller. Vi vil ha en kombinasjon av lav responstid og god gruppering.

Et viktig punkt med disse resultatene er at vi kjører testen på samme maskin som kjører API-ene. Det er derfor ikke noen ekstra tid brukt på ruting via Internett.

3.5.3 Repost API load plan A

Som nevnt har vi sett på tallene fra Reddit sin 2019 rapport (Murphy, 2019). Vi prøver å lage en test der vi har flere GET forespørsler enn vi har POST, siden flere er lesere enn aktive deltagere. Denne testplanen har relativt få forespørsler totalt, men det er for å sette en basislinje for hva vi kan forvente. Alle DELETE forespørsler er kjørt i tearDown tråd-gruppen, og denne har

kun en tråd. Det er viktig å merke seg før man sammenligner denne operasjonen med andre forespørsler.

De endepunktene som har veldig få forespørsler er ikke særlig viktige for statistikken siden det er for lite grunnlag til å si noe konkret om de. Grunnen til at de er med er for eksempel at vi trenger å gjøre en forespørsel for å finne alle postene som vi har laget med testbrukeren. Det kan vi ikke gjøre ellers siden id blir generert av serveren.

Alle tråd-gruppene utenom setUp og tearDown er satt til 25 tråder, altså 25 simulerte brukere. Antall runder per tråd-gruppe varierer fra 2 til 10 i denne testplanen. Det er 1 sekund oppstartstid for alle tråd-grupper. Tekstverdier vi lager i denne testen er laget med RandomString. Resub navn er 16 tegn, resub beskrivelse er 128 tegn, post tittel er 32 tegn, post innhold er 128 tegn.

Når vi gjør en forespørsel for å hente poster fra en resub har vi satt en begrensning på 10 i stedet for standard grensen på 100. Dette er for at det ikke skal være alt for mange GET forespørsler i forhold til POST, siden hver av de postene skal hentes inn med en egen forespørsel.

Totalt antall forespørsler i denne testen er 150482. 6302 POST, 6301 DELETE og 137880 GET.

3.5.4 Repost API load plan B

Denne testplanen er basert på plan A, men den har mange flere POST og en del mer GET. Vi vil se om hastigheten holder seg jevnt over en lengre periode. Som i plan A så er alle DELETE forespørsler kjørt i tearDown tråd-gruppen som bare har en tråd.

For å lage flere POST forespørsler har vi en egen tråd-gruppe som lager kommentarer på poster. Vi henter også postene flere ganger i en egen tråd-gruppe senere i testen.

I denne testplanen har vi kjørt samme plan men med forskjellig antall simulerte brukere. Her har vi kjørt med 10 tråder, 25 tråder og 50 tråder på testen. Dette gjør vi for å se om det har noen effekt på resultatene. Når vi endrer antall tråder endrer vi også antall runder tråd-gruppen skal kjøre slik at det totalt er like mange forespørsler for alle testene.

Denne testen har en del flere forespørsler, særlig POST og DELETE. Totalt har vi 613505 forespørsler hvorav 137552 er POST, 137551 er DELETE og resten er GET.

3.6 Materiale og utstyr

I dette kapittelet lister vi alt utstyr som er brukt under utvikling av dette prosjektet.

3.6.1 HP Envy 13-ad101

Maskin brukt til utvikling og rapportskriving.

Spesifikasjoner:

Operativsystem	Linux Ubuntu 18.04 LTS
Prosessor	Intel Core i5-8250 @ 1.60 GHz
Minne	8GB 1866 MHz LPDDR3 SDRAM
Grafikkort	NVIDIA GeForce MX150 / Intel UHD Graphics 620
Skjerm	13,3” 1920 x 1080
Lagring	256 GB M.2 SSD

Figur 20. HP Envy 13-ad101 (foto: komplett.no)

3.6.2 Acer Predator G3620

Maskin brukt til utvikling og rapportskriving.

Spesifikasjoner:

Operativsystem	Windows 10 Home
Prosessor	Intel Core i7-3770 @ 3.4 GHz
Minne	8GB 1866 MHz
Grafikkort	NVIDIA GeForce GTX 680
Skjerm	27” 2560 x 1440 + 24” 1920 x 1080
Lagring	128 GB SSD + 2 x 1 TB HDD

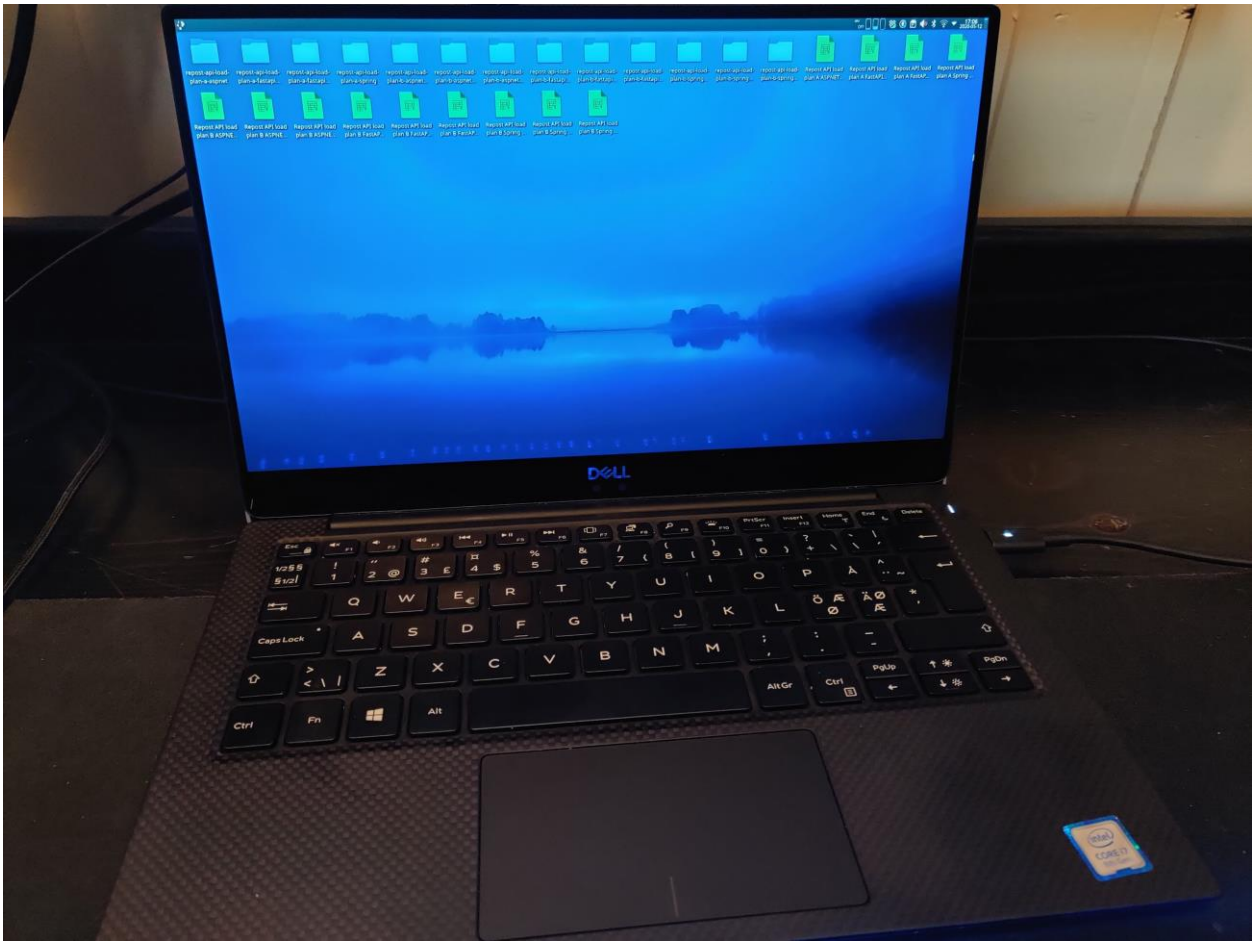
Figur 21. Acer Predator G3620 (foto: komplett.no)

3.6.3 Dell XPS 13 9370

Maskin brukt til utvikling og rapportskriving.

Spesifikasjoner:

Operativsystem	Linux Ubuntu 18.04 LTS
Prosessor	Intel Core i7-8550 @ 1.80 GHz
Minne	8GB 1866 MHz LPDDR3 SDRAM
Grafikkort	Intel UHD Graphics 620
Skjerm	13,3” 1920 x 1080
Lagring	256 GB M.2 SSD



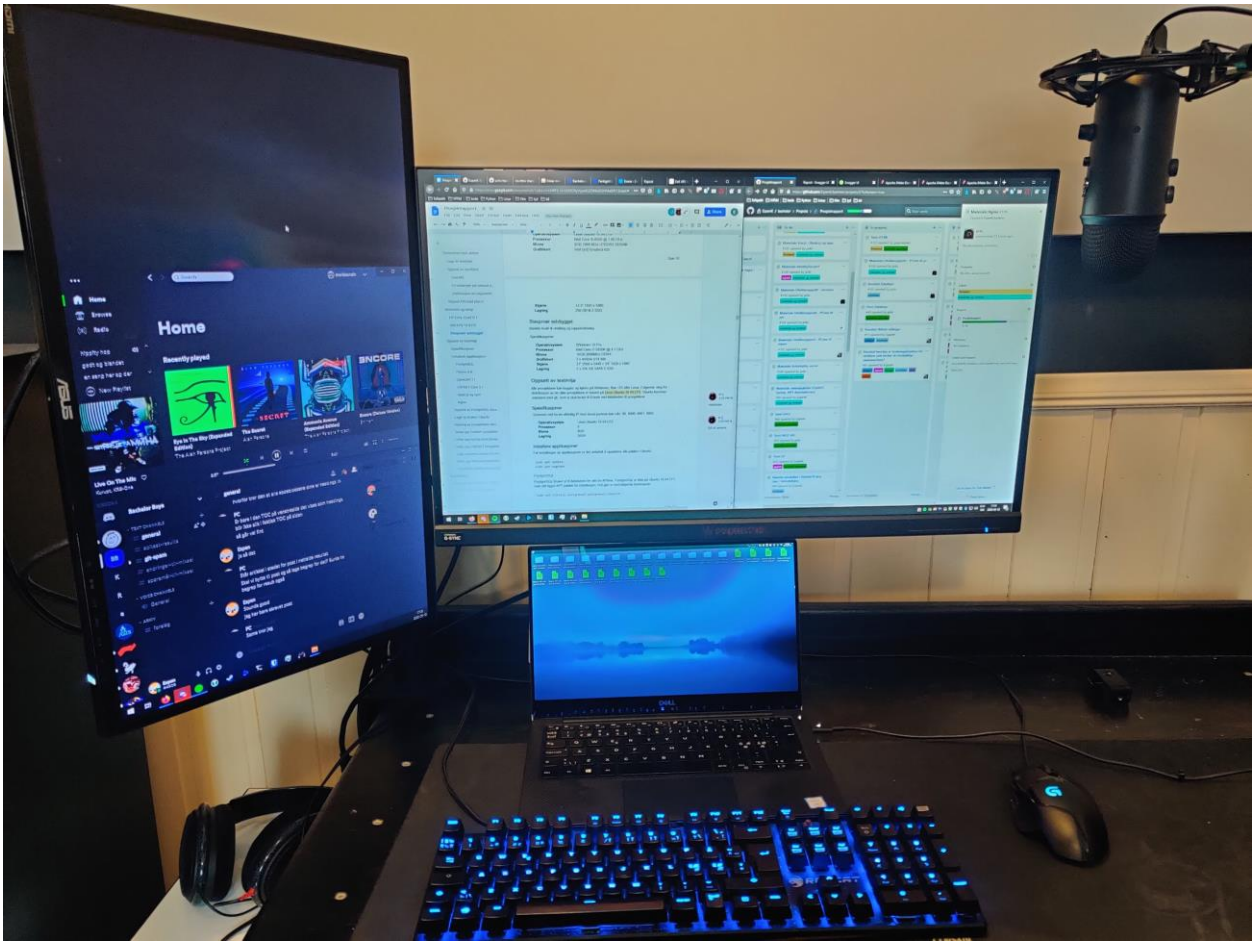
Figur 22. Dell XPS 13 9370

3.6.4 Stasjonær selvbygget

Maskin brukt til utvikling og rapportskriving.

Spesifikasjoner:

Operativsystem	Windows 10 Pro
Prosessor	Intel Core i7-5930K @ 4.1 GHz
Minne	16GB 2666MHz DDR4
Grafikkort	2 x NVIDIA GTX 980
Skjerm	27” 2560 x 1440 + 24” 1920 x 1080
Lagring	2 x 256 GB SATA 3 SSD



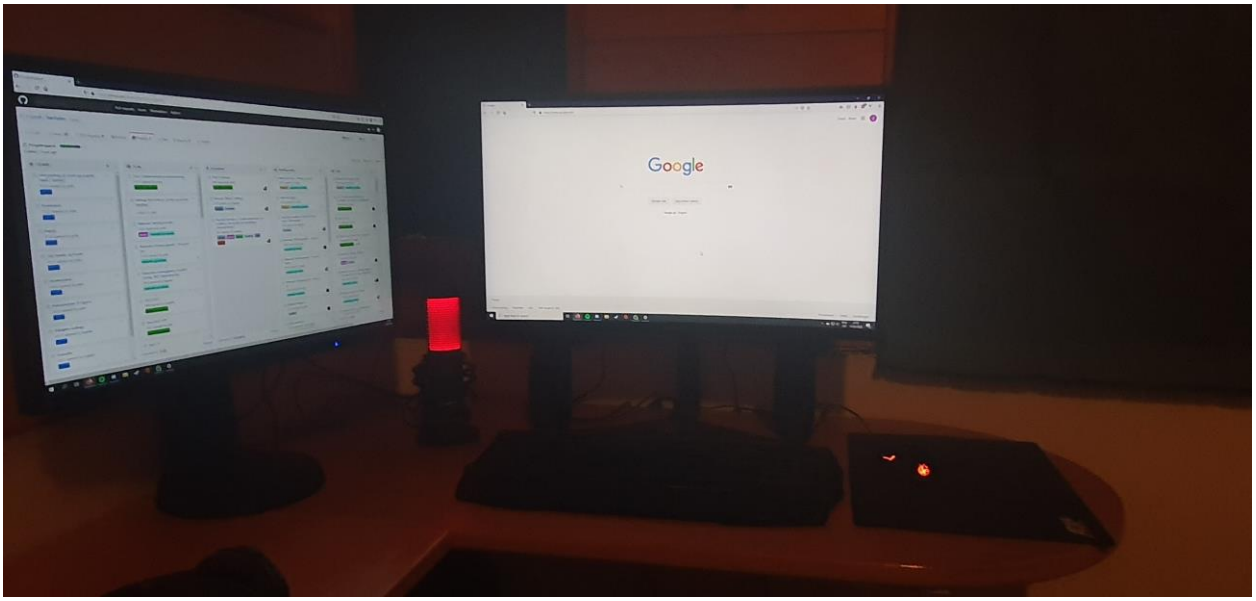
Figur 23. Skrivebord hvor Stasjonær selvbygget er koblet opp

3.6.5 Stasjonær selvbygget 2

Maskin brukt til utvikling og rapportskriving.

Spesifikasjoner:

Operativsystem	Windows 10 Pro
Prosessor	Intel Core i7-9700K @ 3.6 GHz
Minne	16GB 32000 MHz DDR4
Grafikkort	NVIDIA GeForce RTX 2080 SUPER
Skjerm	2 x 1920 x 1080
Lagring	1 TB M.2 SSD



Figur 24. Skrivebord hvor Stasjonær selvbygget 2 er koblet opp

3.6.6 Supermicro 6028U-E1CNRT

Dette er maskinen som kjører de virtuelle maskinene vi bruker hos NTNU i Ålesund. Vår virtuelle maskin hvor vi kjører API-ene og ytelsestester med JMeter tjenes av denne.

Spesifikasjoner:

Operativsystem	ESXi 6.0
Prosessor	2 x Intel Xeon E5-2687Wv4 12-Core @ 3.00 GHz
Minne	768 GB (24 x 32 GB)
Lagring	2 x 480 GB SSD & 6 x 6 TB SATA HDD

Figur 25. Supermicro 6028U-E1CNRT (foto: bsicomputer.com)

3.7 Oppsett av testmiljø

Alle prosjektene kan bygges og kjøres på Windows, Mac OS eller Linux. Følgende steg for distribusjon av de ulike prosjektene er basert på Linux Ubuntu 18.04 LTS. Ubuntu kommer standard med git, som vi skal bruke til å laste ned kildekoden til prosjektene.

3.7.1 Spesifikasjoner

Serveren må ha en offentlig IP hvor disse portene kan nås: 80, 8000, 8001, 8002. Spesifikasjonene nedenfor er basert på en virtuell maskin som tjenes ved NTNU i Ålesund. Se Supermicro 6028U-E1CNRT (3.6.6) for spesifikasjonene til maskina som kjører den virtuelle maskinen.

Operativsystem	Linux Ubuntu 18.04 LTS
Prosessor	8 kjerner
Minne	8 GB
Lagring	50 GB

3.7.2 Installere applikasjoner

Før installasjon av applikasjoner er det anbefalt å oppdatere alle pakker i Ubuntu.

```
sudo apt update
sudo apt upgrade
```

3.7.2.1 PostgreSQL

PostgreSQL bruker vi til databasen for alle tre API-ene. PostgreSQL er ikke på Ubuntu 18.04 LTS, men det ligger APT pakker for installasjon. Det gjør vi med følgende kommando:

```
sudo apt install postgresql postgresql-contrib
```

3.7.2.2 Python 3.8

FastAPI prosjektet er laget med Python 3.8, og dermed burde samme versjon installeres her. Versjonen av Python som kommer pakket med Ubuntu 18.04 LTS er Python 3.6. Altså må vi installere en nyere versjon av denne. Samtidig trenger vi også pakker for venv, slik at vi kan lage en egen instans av Python miljøet for prosjektet. pip må også installeres her, og kreves for installasjon av eksterne Python bibliotek. Disse pakkene installerer vi med APT:

```
sudo apt install python3.8-dev python3.8-venv python3-pip
```

3.7.2.3 OpenJDK 11

OpenJDK 11 brukes til Spring Boot prosjektet. Ubuntu inkluderer ingen versjon av Java JDK. Derfor installerer vi OpenJDK versjon 11 for vårt prosjekt ved å bruke APT:

```
sudo apt install openjdk-11-jdk-headless
```

3.7.2.4 ASP.NET Core 3.1

Ubuntu inkluderer ikke ASP.NET Core 3.1, og disse finnes heller ikke i APT. Derfor må vi legge til Microsoft Repository for å kunne få tilgang til disse (Microsoft, 2020b). Det gjøres med følgende kommando:

```
wget https://packages.microsoft.com/config/ubuntu/18.04/packages-microsoft-
prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

Etter repository er lagt til må vi oppdatere alle pakker tilgjengelig i APT og så installere .NET SDK 3.1 slik:

```
sudo apt update
sudo apt install dotnet-sdk-3.1
```

3.7.2.5 Node.js og npm

Node.js og npm brukes til front-end og kan installeres med APT med følgende kommando:

```
sudo apt install nodejs-dev npm
```

3.7.2.6 Nginx

Nginx brukes som webserver til front-end. Den tjener web filene som vi skal generere med Node.js og npm. Nginx installeres med følgende APT kommando:

```
sudo apt install nginx
```

Nginx kjører på *root* brukeren i systemet, og derfor burde vi konfigurere tjenesten samtidig. Først fjerner vi standardkonfigurasjonen i Nginx med følgende kommando:

```
sudo rm /etc/nginx/sites-enabled/default
```

Nå kan vi legge inn vår egen konfigurasjon. Denne filen finner vi i Vedlegg 2, og vi bruker et tekstredigeringsverktøy for å endre den. Bruk kommandoen under eller en annen metode for å lagre fila på riktig sted.

```
sudo nano /etc/nginx/sites-available/repost
```

Til slutt må vi aktivere konfigurasjonen. Dette gjør vi ved å lage en symbolsk lenke til fila i *sites-enabled* mappen, hvor vi også fjernet standardkonfigurasjonen. Deretter restarter vi Nginx slik at den finner og tar i bruk den nye konfigurasjonen.

```
sudo ln -s /etc/nginx/sites-available/repost /etc/nginx/sites-
enabled/repost
sudo systemctl restart nginx
```

3.7.3 Oppsett av PostgreSQL database

Først må vi lage en PostgreSQL bruker (også kalt rolle) som kan være tilknyttet flere databaser. Vi skal lage én database for hver API. Brukeren i PostgreSQL skal ha samme navn som en bruker i Ubuntu. Her kan vi få ekstra sikkerhet ved å lage en ny bruker spesielt egnet til databasen, men ettersom brukeren vår skal være for analyse, ikke produksjon, så setter vi opp databasen på samme bruker. Uansett har vi også sørget for at denne brukeren ikke har *sudo* rettigheter.

Vi bytter til *postgres* brukeren for å gjøre administrative endringer i postgres.

```
sudo -i -u postgres
```

Deretter lager vi den nye PostgreSQL brukeren. I kommandoen som lager brukeren må også et nytt passord bli oppgitt. Dette skal brukes for å koble til databasen i de forskjellige API-ene. Databasen skal også kjøre lokalt, så dette behøver ikke å være et sikkert passord, men merk at det skal skrives i klartekst i API konfigurasjonene.

```
createuser --pwprompt --no-superuser repost
```

Deretter lager vi alle databasene og oppgir at eieren er vår *repost* bruker.

```
createdb --owner repost repost_fastapi
createdb --owner repost repost_spring
createdb --owner repost repost_aspnet
```

Nå kan vi logge ut av *postgres* brukeren.

```
exit
```

3.7.4 Lage ny bruker i Ubuntu

Nå må vi lage den samme brukeren i Ubuntu. Skriv kommandoen nedenfor og fyll ut begge passord feltene med et nytt passord. Resten av feltene kan hoppes over ved å trykke **Enter**. Kommandoene kjøres med *sudo*, og passordet til brukeren som er logget inn må derfor skrives først.

```
sudo adduser repost
```

Vi legger også til brukeren i *systemd-journal* gruppen slik at logger kan leses.

```
sudo usermod -aG systemd-journal repost
```

For at systemd skal kjøre på denne nye brukeren etter brukeren logger ut må vi skru på *linger* for brukeren (ArchWiki, 2020). Dette steget kan hoppes over om det ikke er behov for å la prosessene kjøre i bakgrunnen når brukeren ikke er logget inn.:

```
sudo loginctl enable-linger repost
```

Til slutt må vi logge ut av vår nåværende bruker og logge inn på den nye. Dette er viktig for at systemd tjenesten for brukeren startes. Logg ut ved å skrive `exit`

Merk at denne brukeren ikke får *sudo* rettigheter, så globale pakker må altså installeres med en annen bruker. Derfor gjorde vi dette steget i kapitlene over.

3.7.5 Kloning av prosjektene med git

Først må vi klone alle prosjektene ved bruk av verktøyet git i kommandolinjen. Først lager vi en mappe *apis* for alle API-ene og kloner disse. Dersom det ikke er mulighet for å bruke GitHub ligger disse også i Vedlegg 4.

```
mkdir ~/apis
cd ~/apis
git clone https://github.com/pckv/repost-fastapi.git
git clone https://github.com/EspenK/repost-spring.git
git clone https://github.com/pckv/repost-aspnet.git
```

git clone kommandoene lager nye mapper med kildekoden for API-ene. For å oppdatere kan vi også gjøre *git pull* i den mappen vi ønsker å laste ned ny kildekode i. Videre distribusjon skal ta sted i disse mappene.

Nå kan vi også klone front-end prosjektet. Her lager vi en ny mappe *web*:

```
mkdir ~/web
cd ~/web
git clone https://github.com/jonsondrem/repost-frontend
```

Vi må også laste ned testplaner for JMeter. Disse filene vil vi plassere i *test* mappen.

```
mkdir ~/test
cd ~/test
git clone https://github.com/EspenK/repost-testplan.git
```

3.7.6 Sette opp FastAPI prosjektet

Før vi setter opp prosjektet må vi lage et Python venv, altså et virtuelt Python miljø. Hensikten er å ha et eget Python miljø for prosjektet, slik at man ikke får konflikt mellom pakker som muligens er installert for andre prosjekter.

Vi lager det virtuelle miljøet i samme mappen som kildekoden slik at vi har god oversikt over hvor miljøet for Python ligger for dette prosjektet. Kommandoene under lager det nye miljøet og bytter til det.

```
cd ~/apis/repost-fastapi
python3.8 -m venv venv
```

Nå som miljøet er laget kan vi installere alle pakkene for prosjektet. Disse installerer vi med pip. Alle pakkene som kreves for prosjektet er definert i fila *requirements.txt*, så vi må fortelle pip at den skal se på denne fila etter pakkene og hvilke versjonsnummer de har. Her må vi bruke versjonen av pip som ligger i *venv* mappen. Dette gjøres med følgende kommando:

```
venv/bin/pip install -r requirements.txt
```

3.7.7 Sette opp Spring Boot prosjektet

Vi bygger prosjektet med Apache Maven. Kildekoden inkluderer en kjørbar versjon av Maven slik at vi slipper å installere denne. Først må denne fila gjøres kjørbar i Ubuntu, og så kan vi kjøre *package* kommandoen. Denne kommandoen vil både installere alle eksterne pakker som kreves for prosjektet og bygge applikasjonen. Informasjon om pakkene ligger i *pom.xml* fila, og Maven leter etter denne når vi kjører *package* kommandoen.

```
cd ~/apis/repost-spring
chmod +x mvnw
./mvnw package
```

3.7.8 Sette opp ASP.NET prosjektet

Vi bygger kildekoden ved å kjøre dotnet publish. Dette lager en konfigurasjon for Release.

```
cd ~/apis/repost-aspnet
dotnet publish --configuration Release
```

3.7.9 Lage systemd service for API-er

Vi setter opp *systemd* service filer for å kjøre alle API-ene. Dette lar API-ene kjøre i bakgrunnen som en tjeneste. Først må vi lage en service fil for hvert av de tre API-ene. Ettersom service funksjonene vi lager skal tilhøre *repost* brukeren skal de legges i *~/.config/systemd/user*

mappen. Disse mappene er vanligvis ikke opprettet på forhånd, så vi må gjøre det med følgende kommando:

```
mkdir -p ~/.config/systemd/user
```

Deretter kan vi lage alle service filene i denne nye mappen. Filene ligger i Vedlegg 1, så følg videre instruksjoner for oppsett der.

Vi kan bruke nano for å skrive inn disse filene. Alternativt kan man bruke et annet program for tekst, eller kopiere filene over til serveren på andre måter. Skriv kommandoene under én om gangen og lim inn innholdet fra de riktige filene fra Vedlegg 1.

```
nano ~/.config/systemd/user/repost-fastapi.service
nano ~/.config/systemd/user/repost-spring.service
nano ~/.config/systemd/user/repost-aspnet.service
```

Etter alle service filene ligger på plass må vi fortelle systemd at filene er lagt til. Skriv følgende kommando:

```
systemctl --user daemon-reload
```

Deretter aktiverer vi alle tjenestene. Dette gjør også at de skrus på automatisk ved systemoppstart.

```
systemctl --user enable repost-fastapi
systemctl --user enable repost-spring
systemctl --user enable repost-aspnet
```

Og til slutt starter vi alle tjenestene.

```
systemctl --user start repost-fastapi
systemctl --user start repost-spring
systemctl --user start repost-aspnet
```

Nå skal alle API-ene være oppe. For å teste at alle er oppe og svarer på API forespørsler kan vi nå gå til følgende adresse i en nettleser:

```
http://SERVER_IP:PORT/api/swagger
```

Erstatt **SERVER_IP** med IP til maskina som tjener serverene. **PORT** må også erstattes, og de følgende portene er i bruk.

8000	FastAPI
8001	Spring Boot
8002	ASP.NET

Dersom en av serverne ikke fungerer som den skal kan loggen leses med kommandoen nedenfor. Erstatt **TJENESTE** med navnet på tjenesten som ikke fungerer, hvor **TJENESTE** er en av *repost-fastapi*, *repost-spring* eller *repost-aspnet*.

```
journalctl --user-unit TJENESTE
```

3.7.10 Sette opp front-end nettsiden

Vi bruker npm til å bygge front-end. Først må vi installere alle eksterne pakker som kreves. Dette gjør vi slik:

```
cd ~/web/repost-frontend
npm install
```

Før vi kan bygge prosjektet må vi koble opp alle API-ene. Dette gjør vi ved å redigere *.env* fila som ligger i hovedmappen til prosjektet. Det eneste som trenger å gjøres er å endre alle **127.0.0.1** med den offentlige IPen av serveren. Det kan gjøres manuelt med et tekstredigeringsverktøy som *nano*, men vi bruker kommandoen nedenfor for å gjøre erstatninger. Bytt ut **SERVER_IP** med serverens offentlige IP.

```
sed -i 's/127.0.0.1/SERVER_IP/g' .env
```

Merk at IPene her ikke kan være lokale for systemet, ettersom de skal bli kalt av en nettleser.

Deretter bygger vi prosjektet, som resulterer i at alle web filene havner i *dist/* mappen.

```
npm run build
```

Nginx vil nå laste inn disse filene om vi kobler til nettsiden. Dette kan vi nå gjøre ved å navigere til **SERVER_IP** i en nettleser. Derfra kan vi teste at alle API-ene fungerer ved å bruke dropdown menyen øverst til høyre.

3.7.11 Oppsett av JMeter for testing

Vi bruker JMeter som testverktøy. Det er skrevet i Java og trenger derfor at Java er installert på maskinen der det skal kjøre. Siden vi har OpenJDK 11 for å kjøre Spring Boot trenger vi ikke å installere noen andre programmer før vi begynner.

På nedlastingssiden (*Apache JMeter - Download Apache JMeter*, no date) til JMeter finner vi lenker til kildekoden og til ferdigbygget kode. Vi vil laste ned den ferdigbygde koden. Det anbefales å verifisere (*Verifying Apache Software Foundation Releases*, no date) filen som blir lastet ned for å være sikker på at det er riktig fil.

Last ned programmet og hashfilen. Lag ei ny mappe for testene *~/test* og navigerer til denne.

```
cd ~/test
```

```
wget https://apache.uib.no//jmeter/binaries/apache-jmeter-5.2.1.tgz
wget https://www.apache.org/dist/jmeter/binaries/apache-jmeter-
5.2.1.tgz.sha512
```

Sjekk at filen som er lastet ned er riktig. Kommandoen nedenfor skal vise *apache-jmeter-5.2.1.tgz: OK* om alt er i orden.

```
sha512sum -c apache-jmeter-5.2.1.tgz.sha512
```

Om checksum stemmer skal vi pakke ut filene.

```
tar xf apache-jmeter-5.2.1.tgz
```

Nå er JMeter satt opp og kan brukes fra *~/test/apache-jmeter-5.2.1*. For at genererte rapporter skal se like ut som våre rapporter må vi endre verdien for hvor ofte det lages et datapunkt som brukes på grafer. Vi endrer fra 60000 ms til 1000 ms med følgende kommando:

```
sed -i
's/#jmeter.reportgenerator.overall_granularity=60000/jmeter.reportgenerator
.overall_granularity=1000/g' ~/test/apache-jmeter-5.2.1/bin/user.properties
```

Altså fjerner vi # tegnet fra linja og endrer verdien fra 60000 til 1000. Dette kan også gjøres manuelt med et tekstredigeringsverktøy.

Om testplanene er klonet med git som beskrevet over i Kloning av prosjektene med git (3.7.5) vil de ligge i mappen *~/test/repost-testplan*. Vi går utifra at det er her planene ligger, om du har plassert testplanene en annen plass må stien endres i kommandoene som vises under.

For å kjøre en testplan må vi ha tilgang til JMeter og testplanene. Om alt er satt opp slik som beskrevet skal vi ha tilgang til de i *~/test*. Om man skal lage flere rapporter er det lurt å lage en hovedmappe der alle rapportene blir lagt.

```
mkdir ~/test/reports
```

Kjør kommandoen nedenfor for å utføre *Repost API load plan A FastAPI*.

```
cd ~/test
apache-jmeter-5.2.1/bin/jmeter -n -t 'repost-testplan/Repost API load plan
A FastAPI.jmx' -l 'reports/Repost API load plan A FastAPI results.csv' -e -
o reports/repost-api-load-plan-a-fastapi
```

Med denne kommandoen kjører vi JMeter uten GUI. Testplanen som brukes er *Repost API load plan A FastAPI.jmx*, bestemt av flagget *-t*. Resultatene blir lagret i mappen *reports* med filnavnet *Repost API load plan A FastAPI.csv* bestemt av *-l* flagget. Når testen er ferdig blir det generert en rapport basert på csv filen, denne rapporten blir lagt i undermappen *repost-api-load-plan-a-fastapi* i mappen *reports*. For å se resultatene i rapporten må *index.html* filen åpnes i en nettleser.

For å kjøre andre tester må disse flaggene endres. Sjekk *~/test/repost-testplan* mappen for resten av *jmx* filene. Sørg for at navnet på resultat filene stemmer overens med testplanen slik at det blir lite forvirring av hvilke tester som er hva.

4 Resultater

I dette kapittelet viser vi resultatene av ferdige produkter, brukeropplevelse og ytelsestestene.

4.1 Resultat av produktene - API-er

I introduksjonen har vi en kravspesifikasjon der vi beskriver et API for et forum nettsted. Vi har klart å implementere dette API-et i de tre rammeverkene FastAPI, Spring Boot og ASP.NET Core Web APIs.

Her gjengir vi kravspesifikasjonen igjen og under hvert punkt legger vi endepunktet som tar seg av aktiviteten som er beskrevet.

En ikke-autentisert bruker skal kunne

- Lage en bruker

  `POST /api/users/`

- Logge inn for å få en autentiserings token

  `POST /api/auth/token`

- Få informasjon om en bruker

  `GET /api/users/{username}`

- Få informasjon om en resub

  `GET /api/resubs/{resub}`

- Få informasjon om en post

  `GET /api/posts/{post_id}`

- Få informasjon om en kommentar

  Kommentarer kan ikke hentes ut enkeltvis, men som en liste kommentarer av en bruker eller i en post.

- Få en liste over resuber eid av en bruker

  `GET /api/users/{username}/resubs`

- Få en liste over poster skrevet av en bruker

  `GET /api/users/{username}/posts`

- Få en liste over kommentarer skrevet av en bruker

  `GET /api/users/{username}/comments`

- Få en liste over resuber

  `GET /api/resubs/`

- Få en liste over poster i en resub

  `GET /api/resubs/{resub}/posts`

- Få en liste over kommentarer i en post

  `GET /api/posts/{post_id}/comments`

En autentisert bruker skal kunne

- Lage en resub

  `POST /api/resubs/`

- Redigere sin egen resub
  - Overføre eierskap av resuben til en annen bruker

  `PATCH /api/resubs/{resub}`

  Overføring av eierskap skjer ved at man legger til brukernavnet til den nye brukeren i feltet `new_owner_username` i data. Om det eksisterer en bruker med det brukernavnet blir den brukeren satt som eier.

- Slette sin egen resub

  `DELETE /api/resubs/{resub}`

- Lage en post i en resub

  `POST /api/resubs/{resub}/posts`

- Redigere sin egen post

  `PATCH /api/posts/{post_id}`

- Slette sin egen post

  `DELETE /api/posts/{post_id}`

- Slette en post i sin egen resub

  `DELETE /api/posts/{post_id}`

Siden vi gjør en sjekk på om brukeren som sendte forespørselen har lov til å slette posten kan vi bruke samme endepunkt for sletting av egen post eller sletting av en annens post i sin egen resub.

- Gi positiv eller negativ stemme til en post

  `PATCH /api/posts/{post_id}/vote/{vote}`

  For å gi negativ stemme er verdien for vote -1, for positiv stemme er verdien 1.

- Fjern stemme fra en post

  `PATCH /api/posts/{post_id}/vote/{vote}`

  For å fjerne stemmen fra en post settes verdien av vote til 0.

- Lage en kommentar i en post

  `POST /api/posts/{post_id}/comments`

- Redigere sin egen kommentar

  `PATCH /api/comments/{comment_id}`

- Slette sin egen kommentar

  `DELETE /api/comments/{comment_id}`

- Slette en kommentar i sin egen resub

  `DELETE /api/comments/{comment_id}`

  Her er det samme prinsipp som for sletting av post. Siden vi tar en sjekk i API-et for å bestemme om brukeren som sender forespørselen har lov til å slette kommentaren så kan samme endepunkt brukes for å slette sin egen kommentar eller for å slette en annen brukers kommentar i sin egen resub.

- Gi positiv eller negativ stemme til en kommentar

  `PATCH /api/comments/{comment_id}/vote/{vote}`

  For å gi negativ stemme er verdien for vote -1, for positiv stemme er verdien 1.

- Fjerne stemme fra en kommentar

  `PATCH /api/comments/{comment_id}/vote/{vote}`

  For å fjerne stemmen fra en kommentar settes verdien av vote til 0.

For å få mer detaljer rundt hvert endepunkt må man bruke Swagger UI på `/api/swagger`. Her ligger alt som er med i API-et med alle de forskjellige mulige statuskodene, parametre, mediatyper, eksempel på data og responser.

Vi har altså implementert den samme kravspesifikasjonen i alle tre rammeverkene slik at de har samme funksjonalitet med samme endepunkter og konvensjoner. Dette gjør at alle tre kan brukes uten noen forandring i klienten. For at vi skal sørge for at alle implementasjoner fungerer likt har vi laget og brukt vårt eget testverktøy, beskrevet i Implementasjon av testverktøyet (4.7.5). Testverktøyet gjør alle aktivitetene som er beskrevet i kravspesifikasjonen og vil ikke fungere om det er forskjeller mellom API-ene.

For å demonstrere at man kan bruke samme klient for alle API-ene har vi laget en nettside der man kan bytte mellom de forskjellige API-ene med ett klikk. Denne delen av nettsiden er beskrevet i Nettside utviklet i Vue.js (4.4).

4.2 Resultat av målinger med JMeter

Siden vi har kjørt flere forskjellige testplaner vil dette kapittelet bli delt opp etter testplan og API, med en kort oppsummering for hver plan. Alle resultater som vises her er kjørt på serveren som er beskrevet i Oppsett av testmiljø (3.7). Resultatene er hentet fra JMeter Dashboard generert av CSV filen for hver test som er kjørt.

FastAPI blir kjørt både med Uvicorn og med Gunicorn som bruker 17 Uvicorn workers i test plan A, men i de andre testplanene testes kun FastAPI med Gunicorn som bruker 17 Uvicorn workers.

Spring Boot er det rammeverket med best ytelse. Dette er tall fra plan B 25 threads.

GET: ASP.NET har 70 % av ytelsen til Spring Boot, og FastAPI har 30 %. Spring Boot er 1,4 ganger så raskt som ASP.NET og 3.3 ganger så raskt som FastAPI.

POST: ASP.NET har 42 % av ytelsen til Spring Boot, og FastAPI har 25 %. Spring Boot er 2,4 ganger så raskt som ASP.NET og 4 ganger så raskt som FastAPI.

4.2.1 Oppsett av resultater

For hver test vil vi forklare hvordan testplanen er satt opp og hva vi vil teste. Noen av testene vil være nesten like, men vi endrer på noen få variabler for å se om det har noen effekt. Vil vil legge ved bilder av grafer og tabeller. Alle rapportene blir lagt med i Vedlegg 5 (Vedlegg 5).

4.2.2 Repost API load plan A

Plan A blir kjørt av alle tre rammeverkene, men FastAPI blir kjørt på to forskjellige måter.

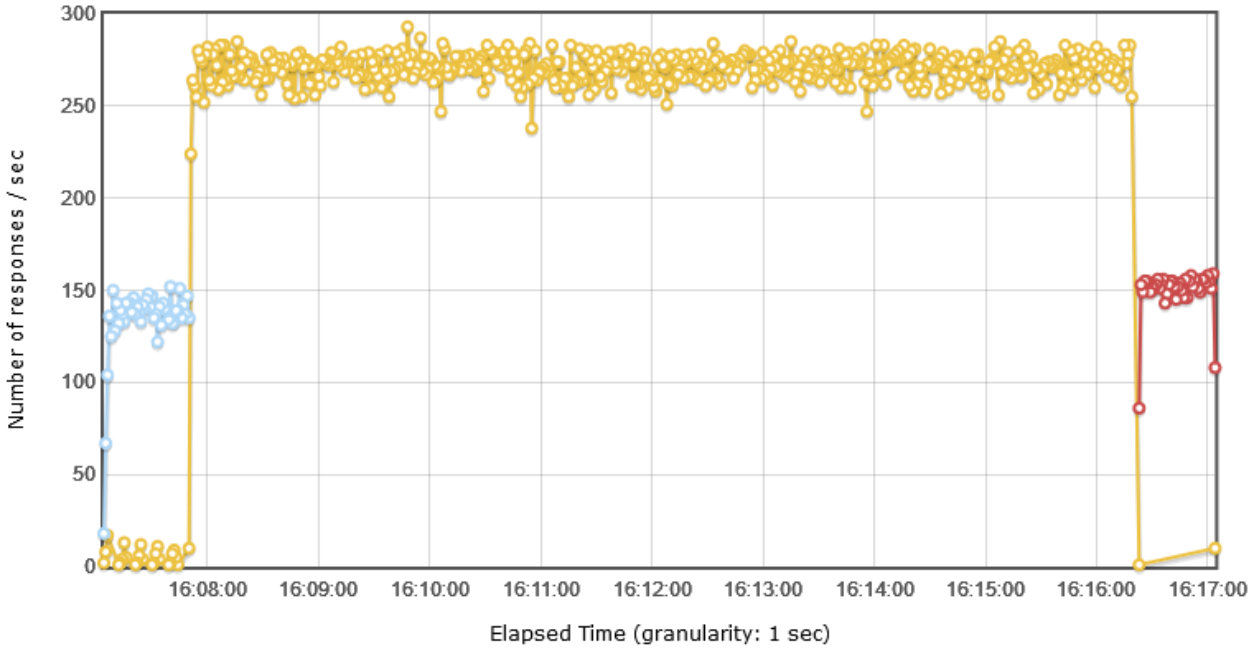
FastAPI kjørt med Uvicorn. Med kun én tråd som jobber er det ikke forventet at vi får særlig gode resultater fra denne testen. Siden FastAPI kjører med én tråd er det ikke store forskjeller mellom DELETE og de andre metodene, selv om testen kjører 25 tråder for alt annet enn DELETE.

Når vi ser på grafen over forespørsler per sekund ser vi at alle metodene holder seg relativt jevnt, men GET er klart den som er raskest med 275 forespørsler per sekund. DELETE ligger på 150 forespørsler per sekund og POST på 140.

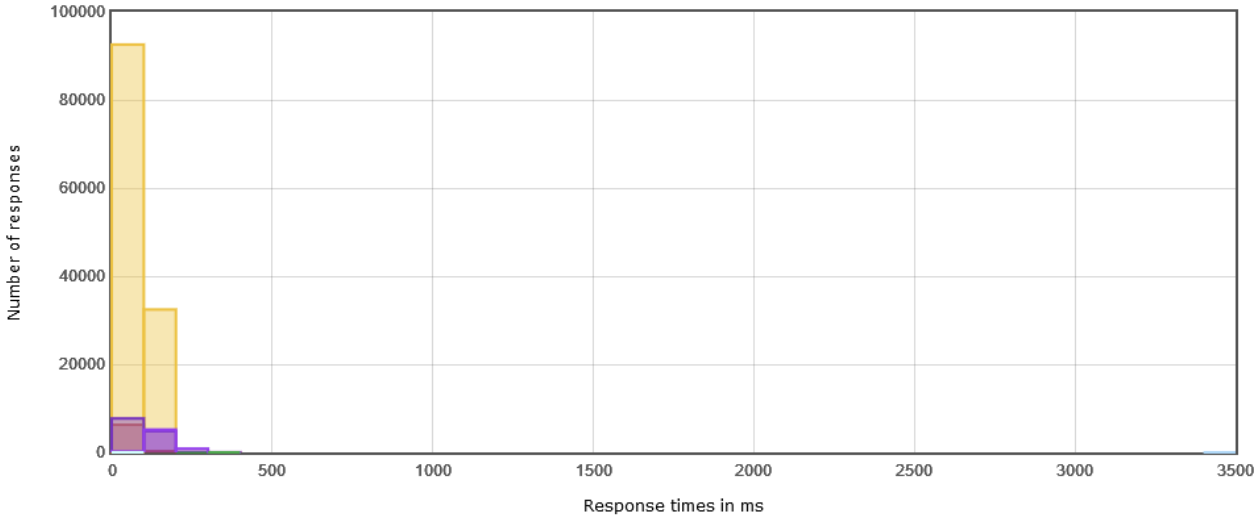
Distribusjon av responstider viser at det er en del forskjeller på responstid. I hovedsak er responstidene rundt 200 mseller mindre, med noen utenfor denne grupperingen. Gjennomsnittlig responstid er 91,37 ms, og responstid for GET posts er 90,94 ms.

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**150491**	**0**	**0.00%**	**91.37**	**1**	**3427**	**106.00**	**113.00**	**130.00**	**250.57**	**189.70**	**88.92**
DELETE /posts/post	6250	0	0.00%	6.47	5	53	7.00	7.00	8.00	152.52	20.26	55.41
DELETE /resubs /resub/posts	50	0	0.00%	4.04	3	5	4.90	5.00	5.00	241.55	32.08	90.58
DELETE /users/me	1	0	0.00%	4.00	4	4	4.00	4.00	4.00	250.00	33.20	90.09
GET /posts/post	125000	0	0.00%	90.94	3	195	111.00	119.00	138.00	245.60	114.64	83.94
GET /resubs /resub/posts	12500	0	0.00%	96.24	9	184	117.00	125.00	143.00	24.56	87.74	9.14
GET /users /user/comments	1	0	0.00%	5.00	5	5	5.00	5.00	5.00	200.00	24.61	75.20
GET /users /user/posts	10	0	0.00%	345.50	3	3427	3084.70	3427.00	3427.00	0.22	48.50	0.08
GET /users /user/resubs	376	0	0.00%	157.99	7	374	258.30	284.00	332.23	0.63	8.30	0.23
POST /auth/token	1	0	0.00%	263.00	263	263	263.00	263.00	263.00	3.80	1.28	1.17
POST /resubs/	50	0	0.00%	9.88	6	16	13.00	14.00	16.00	51.18	19.89	27.49
POST /resubs /resub/posts	6250	0	0.00%	172.03	7	377	210.90	232.00	273.00	138.02	65.10	78.85
POST /users/	1	0	0.00%	336.00	336	336	336.00	336.00	336.00	2.98	0.73	0.73
setUp setProperties	1	0	0.00%	1.00	1	1	1.00	1.00	1.00	1000.00	0.00	0.00

Figur 26. Repost API load plan A FastAPI Uvicorn oversikt



Figur 27. Repost API load plan A FastAPI Uvicorn forespørsler per sekund



Figur 28. Repost API load plan A FastAPI Uvicorn responstider

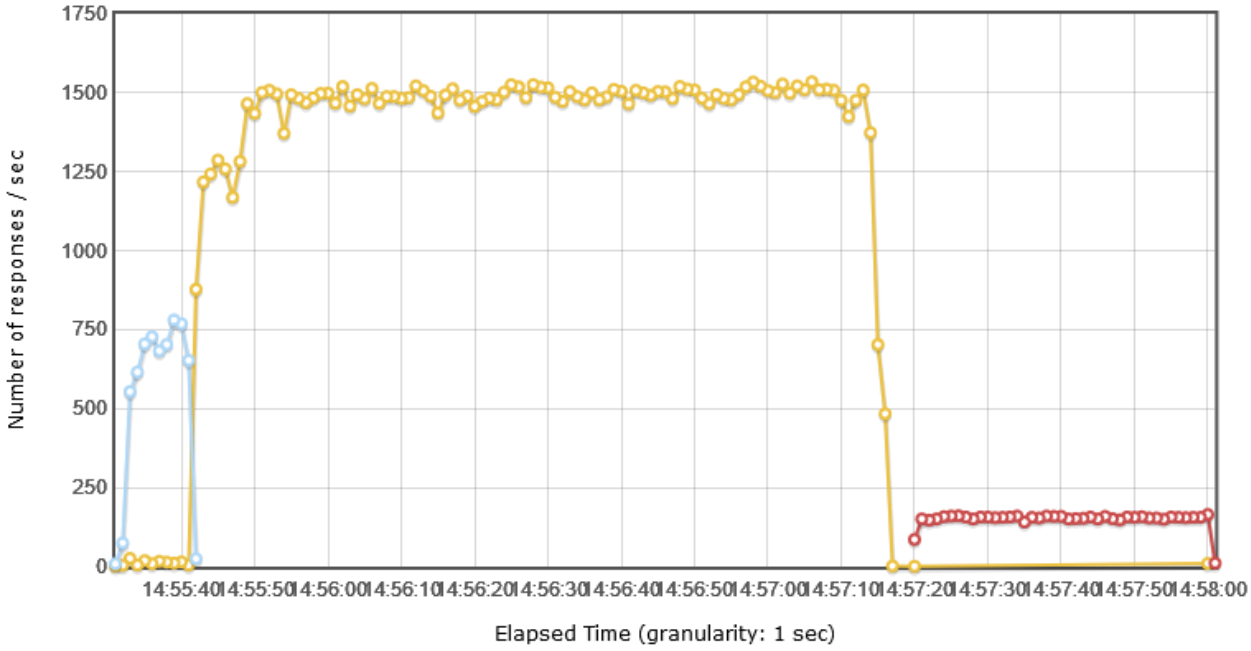
FastAPI kjørt med Gunicorn med 17 Uvicorn workers. Med flere Uvicorn workers går det mye raskere å kjøre POST og GET. DELETE holder seg til omtrent samme verdier som med én Uvicorn worker, noe som er forventet.

Grafen med responstid per metode viser at hastigheten øker litt før en grense blir nådd, men så holder det seg jevnt på dette nivået. POST har omtrent 730 forespørsler per sekund og GET har 1500. DELETE ligger på 150 forespørsler per sekund.

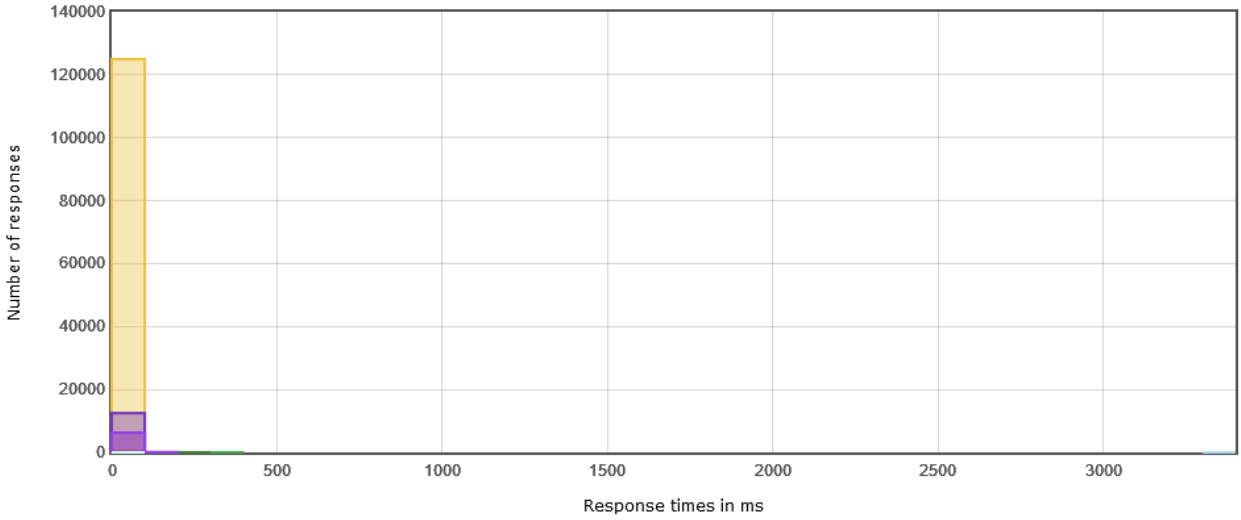
Dette resultatet viser en mer gruppert responstid der nesten alle er under 100 ms. Gjennomsnittlig responstid er 16,55 ms, og responstid for GET posts er 15,01 ms.

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**150491**	**0**	**0.00%**	**16.55**	**2**	**3317**	**20.00**	**26.00**	**39.99**	**1004.14**	**760.20**	**356.32**
DELETE /posts/post	6250	0	0.00%	6.38	5	52	7.00	8.00	9.00	154.52	20.52	56.13
DELETE /resubs /resub/posts	50	0	0.00%	4.36	3	8	5.00	6.00	8.00	228.31	30.32	85.62
DELETE /users/me	1	0	0.00%	4.00	4	4	4.00	4.00	4.00	250.00	33.20	90.09
GET /posts/post	125000	0	0.00%	15.01	2	223	23.00	29.00	44.00	1318.08	615.28	450.52
GET /resubs /resub/posts	12500	0	0.00%	29.54	8	219	45.00	52.00	74.00	131.84	470.95	49.05
GET /users /user/comments	1	0	0.00%	4.00	4	4	4.00	4.00	4.00	250.00	30.76	93.99
GET /users /user/posts	10	0	0.00%	334.40	2	3317	2985.70	3317.00	3317.00	0.23	49.21	0.09
GET /users /user/resubs	376	0	0.00%	28.41	6	171	55.30	72.00	103.91	2.54	33.57	0.91
POST /auth/token	1	0	0.00%	261.00	261	261	261.00	261.00	261.00	3.83	1.29	1.18
POST /resubs/	50	0	0.00%	11.14	5	31	27.90	29.35	31.00	50.05	19.45	26.88
POST /resubs /resub/posts	6250	0	0.00%	30.40	6	134	56.00	64.00	86.00	672.98	317.43	384.47
POST /users/	1	0	0.00%	344.00	344	344	344.00	344.00	344.00	2.91	0.71	0.71
setUp setProperties	1	0	0.00%	2.00	2	2	2.00	2.00	2.00	500.00	0.00	0.00

Figur 29. Repost API load plan A FastAPI Gunicorn oversikt



Figur 30. Repost API load plan A FastAPI forespørsler per sekund



Figur 31. Repost API load plan A FastAPI Gunicorn responstider

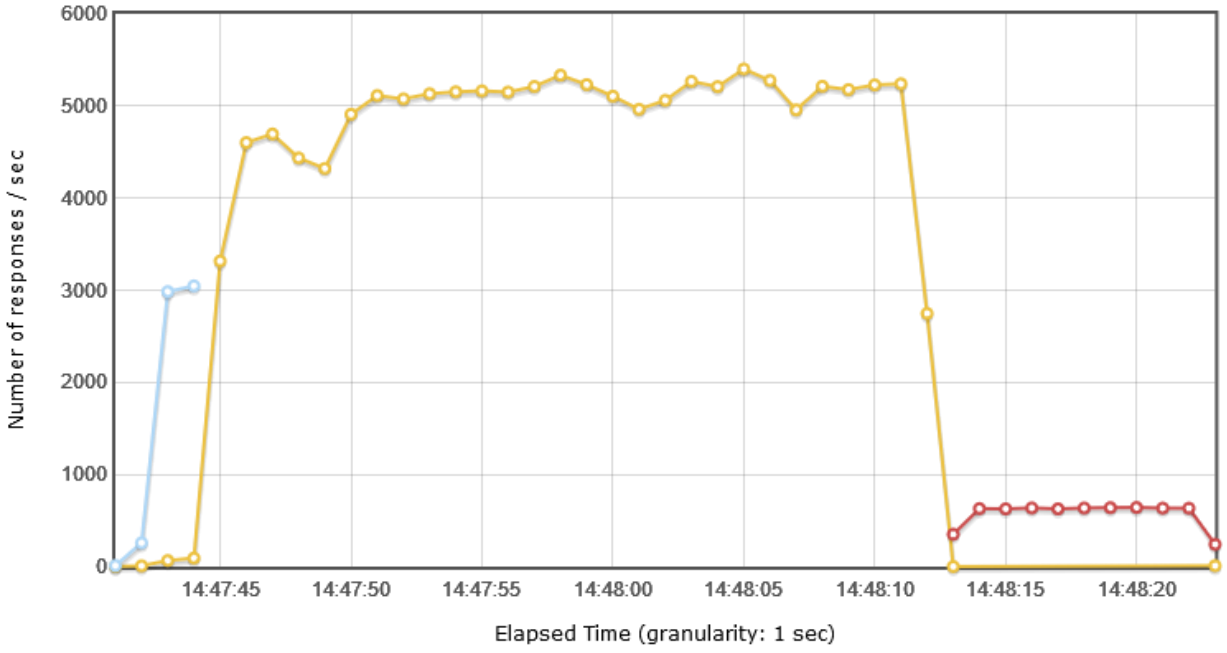
Spring. Spring ser ut til å ha bra resultater. POST og GET er mye raskere enn hos FastAPI, men det er også en stor forskjell på DELETE, selv om det kjøres med bare en tråd av testen.

Grafen for forespørsler per sekund viser at Spring topper ut på 3000 forespørsler per sekund for POST, og med noen variasjoner lander GET på omtrent 5000 forespørsler per sekund. DELETE ligger på 600 forespørsler per sekund.

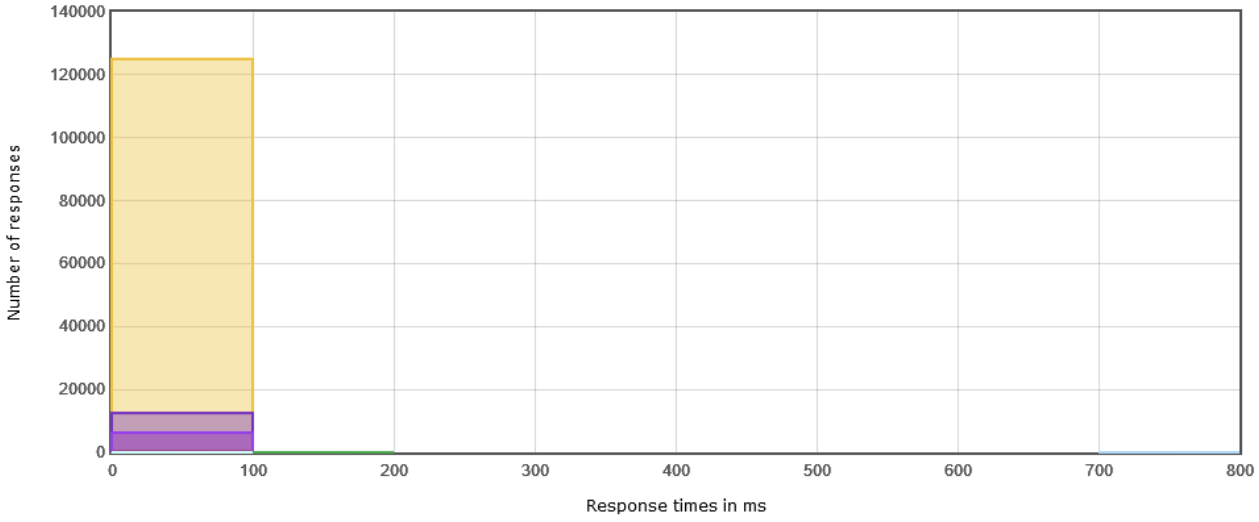
Gruppering av responstider er veldig bra, med nesten ingen over 100 ms. Gjennomsnittlig responstid er 4,64 ms, og responstid for GET posts er 4,5 ms.

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**150491**	**0**	**0.00%**	**4.64**	**0**	**719**	**7.00**	**8.00**	**12.00**	**3586.96**	**3785.09**	**1713.76**
DELETE /posts/post	6250	0	0.00%	1.52	1	8	2.00	2.00	2.00	628.71	228.84	305.76
DELETE /resubs /resub/posts	50	0	0.00%	1.60	1	4	2.00	2.00	4.00	574.71	209.02	285.67
DELETE /users/me	1	0	0.00%	2.00	2	2	2.00	2.00	2.00	500.00	182.13	241.21
GET /posts/post	125000	0	0.00%	4.50	0	41	7.00	9.00	12.00	4515.90	3490.95	2099.19
GET /resubs /resub/posts	12500	0	0.00%	7.34	2	30	11.00	13.00	17.00	451.65	1730.97	223.18
GET /users /user/comments	1	0	0.00%	2.00	2	2	2.00	2.00	2.00	500.00	217.29	249.02
GET /users /user/posts	10	0	0.00%	73.00	1	719	647.30	719.00	719.00	0.94	199.30	0.46
GET /users /user/resubs	376	0	0.00%	4.90	1	36	8.00	10.15	18.23	9.28	122.82	4.47
POST /auth/token	1	0	0.00%	177.00	177	177	177.00	177.00	177.00	5.65	4.54	1.74
POST /resubs/	50	0	0.00%	3.50	2	7	5.00	5.45	7.00	51.65	35.61	34.05
POST /resubs /resub/posts	6250	0	0.00%	4.93	1	55	9.00	11.00	18.00	2934.27	2268.48	2034.51
POST /users/	1	0	0.00%	130.00	130	130	130.00	130.00	130.00	7.69	4.18	1.88
setUp setProperties	1	0	0.00%	2.00	2	2	2.00	2.00	2.00	500.00	0.00	0.00

Figur 32. Repost API load plan A Spring Boot oversikt



Figur 33. Repost API load plan A Spring Boot forespørsler per sekund



Figur 34. Repost API load plan A Spring Boot responstider

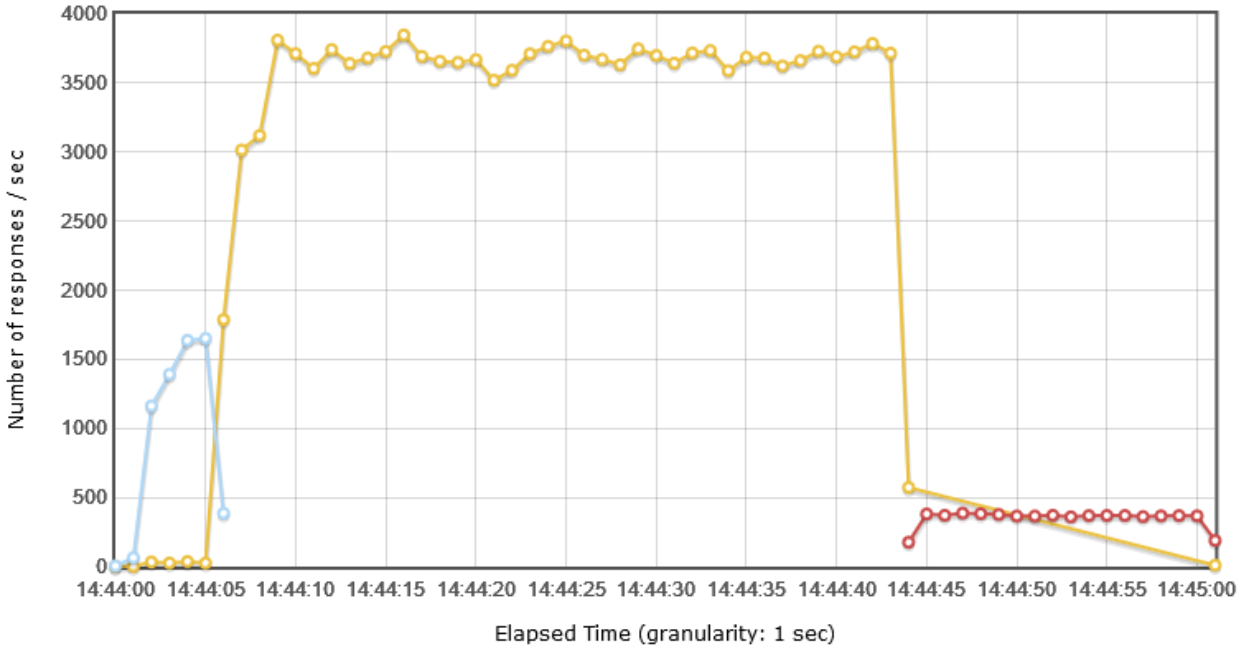
ASP.NET. Her har vi ganske bra resultat, men ikke like bra som Spring. Her har vi også flere forespørsler per sekund for DELETE enn vi har for FastAPI.

Grafen for forespørsler per sekund viser at det tar noen sekunder før en grense er nådd her også. For POST ser det ut til å være en grense rundt 1600 forespørsler per sekund, men det er for lite punkter til å si sikkert. GET har mindre variasjoner enn Spring og ligger på 3750 forespørsler per sekund, og delete rundt 400.

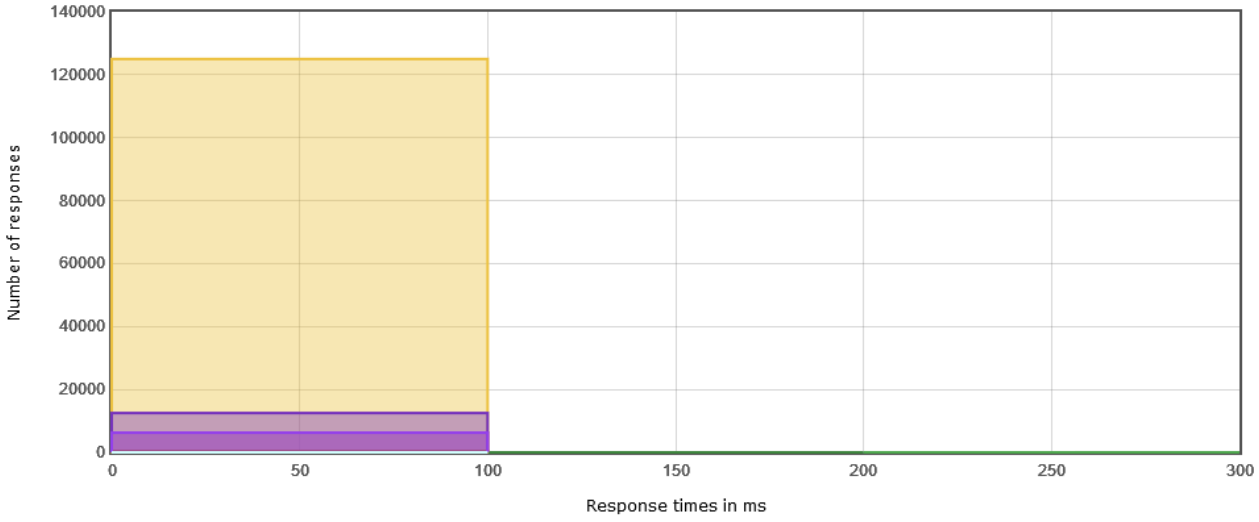
Grupperingen av responstid er den beste med minst avvik. Gjennomsnittlig responstid er på 6,79 ms, og responstid for GET posts er 6,48 ms.

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**150491**	**0**	**0.00%**	**6.79**	**1**	**246**	**8.00**	**9.00**	**10.00**	**2465.17**	**1920.93**	**2143.13**
DELETE /posts/post	6250	0	0.00%	2.63	2	19	3.00	3.00	4.00	370.48	29.31	325.26
DELETE /resubs /resub/posts	50	0	0.00%	2.18	2	3	3.00	3.00	3.00	446.43	35.31	396.73
DELETE /users/me	1	0	0.00%	2.00	2	2	2.00	2.00	2.00	500.00	39.55	437.01
GET /posts/post	125000	0	0.00%	6.48	1	33	8.00	9.00	10.00	3293.46	1630.28	2820.67
GET /resubs /resub/posts	12500	0	0.00%	8.36	2	30	11.00	12.00	14.00	329.47	1171.11	291.82
GET /users /user/comments	1	0	0.00%	4.00	4	4	4.00	4.00	4.00	250.00	39.06	222.41
GET /users /user/posts	10	0	0.00%	26.80	2	246	221.70	246.00	246.00	0.58	123.94	0.52
GET /users /user/resubs	376	0	0.00%	9.55	2	45	14.00	16.15	22.00	6.32	81.84	5.52
POST /auth/token	1	0	0.00%	196.00	196	196	196.00	196.00	196.00	5.10	4.92	1.57
POST /resubs/	50	0	0.00%	4.36	2	8	6.00	7.00	8.00	51.65	24.66	54.28
POST /resubs /resub/posts	6250	0	0.00%	13.79	3	55	21.00	23.00	29.00	1439.10	792.48	1561.36
POST /users/	1	0	0.00%	233.00	233	233	233.00	233.00	233.00	4.29	1.42	1.05
setUp setProperties	1	0	0.00%	1.00	1	1	1.00	1.00	1.00	1000.00	0.00	0.00

Figur 35. Repost API load plan A ASP.NET oversikt



Figur 36. Repost API load plan A ASP.NET forespørsler per sekund



Figur 37. Repost API load plan A ASP.NET responstider

4.2.2.1 Oppsummering av Repost API load plan A

I denne testen er det klart Spring som har de beste resultatene for forespørsler per sekund, med nest best gruppering av responstid og den laveste gjennomsnittlige responstiden. ASP.NET ligger ikke veldig langt bak Spring når det kommer til GET, men hastigheten på POST er nesten halvert. ASP.NET hadde minst avvik på responstid, men gjennomsnittlig var den litt tregere enn Spring. FastAPI er tregest, både med én Uvicorn worker og 17 Uvicorn workers. I forhold til ASP.NET er FastAPI halvparten så rask til POST og noe under halvparten så rask til GET.

4.2.3 Repost API load plan B 10 threads

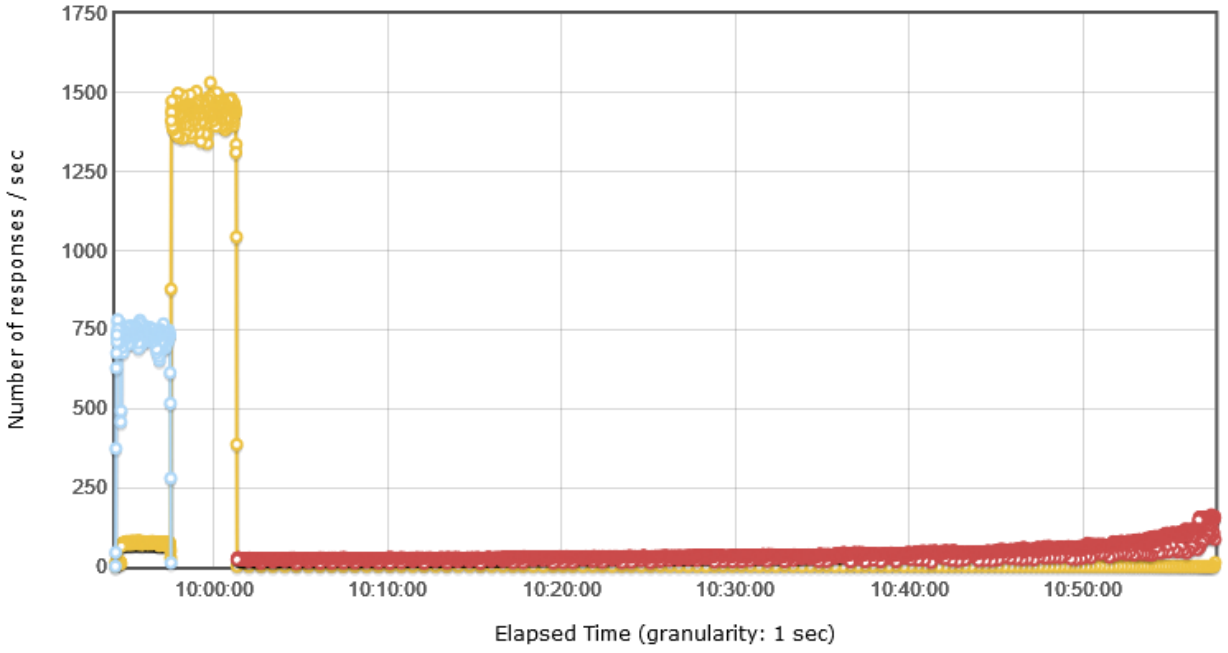
FastAPI. Disse resultatene ser relativt like ut som for plan A for FastAPI, med ett stort unntak. DELETE er tregere i denne testen.

Siden DELETE tar så lang tid er det litt vanskelig å se graden tydelig. Vi kan se at POST ligger litt i underkant av 750, kanskje 725 forespørsler per sekund. POST ser ut til å ha en del variasjoner, og ligger på 1400 forespørsler per sekund. DELETE er veldig tregt men vi kan se at det går raskere mot slutten av testen.

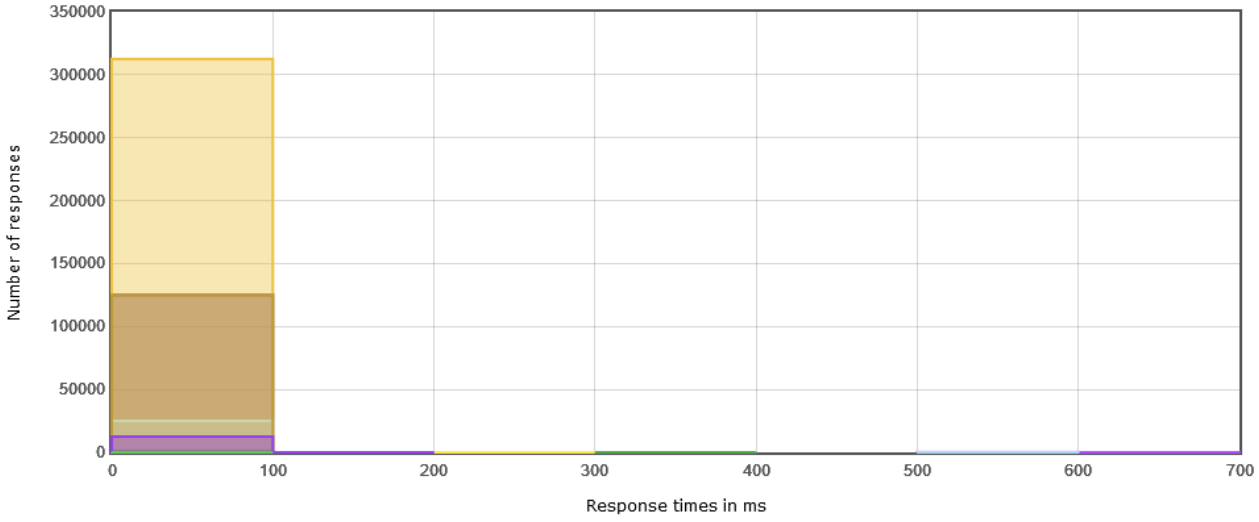
De fleste responstidene er gruppert under 100 millisekunder, med noen få avvik. Gjennomsnittlig responstid er 12,12 ms, og responstiden for GET posts er 6,01 ms.

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**613505**	**0**	**0.00%**	**12.12**	**1**	**658**	**10.00**	**11.00**	**13.00**	**161.53**	**115.03**	**62.58**
DELETE /comments /comment	125000	0	0.00%	25.50	7	83	14.00	15.00	18.00	38.12	5.06	13.96
DELETE /posts/post	12500	0	0.00%	7.28	5	51	9.00	10.00	12.00	126.91	16.86	45.99
DELETE /resubs /resub/posts	50	0	0.00%	4.10	3	6	5.00	5.45	6.00	233.64	31.03	87.62
DELETE /users/me	1	0	0.00%	3.00	3	3	3.00	3.00	3.00	333.33	44.27	120.12
GET /posts/post	312500	0	0.00%	6.01	2	107	9.00	10.00	13.00	1373.94	638.85	467.11
GET /resubs /resub/posts	25000	0	0.00%	21.09	8	137	36.00	39.00	46.00	62.57	383.18	23.28
GET /users /user/comments	125	0	0.00%	570.92	522	658	613.00	627.00	656.70	0.04	12.88	0.01
GET /users /user/posts	25	0	0.00%	279.52	3	579	574.40	577.80	579.00	0.25	43.41	0.09
GET /users /user/resubs	751	0	0.00%	11.47	6	47	16.00	19.00	34.40	0.20	2.61	0.07
POST /auth/token	1	0	0.00%	266.00	266	266	266.00	266.00	266.00	3.76	1.27	1.16
POST /posts /post/comments	125000	0	0.00%	12.00	6	110	17.00	19.00	25.00	726.05	336.73	373.52
POST /resubs/	50	0	0.00%	9.44	6	36	12.90	29.00	36.00	53.19	20.67	28.57
POST /resubs /resub/posts	12500	0	0.00%	13.00	6	100	19.00	21.00	32.00	700.63	329.87	400.26
POST /users/	1	0	0.00%	350.00	350	350	350.00	350.00	350.00	2.86	0.70	0.70
setUp setProperties	1	0	0.00%	1.00	1	1	1.00	1.00	1.00	1000.00	0.00	0.00

Figur 38. Repost API load plan B 10 threads FastAPI oversikt



Figur 39. Repost API load plan B 10 threads FastAPI forespørsler per sekund



Figur 40. Repost API load plan B 10 threads FastAPI responstider

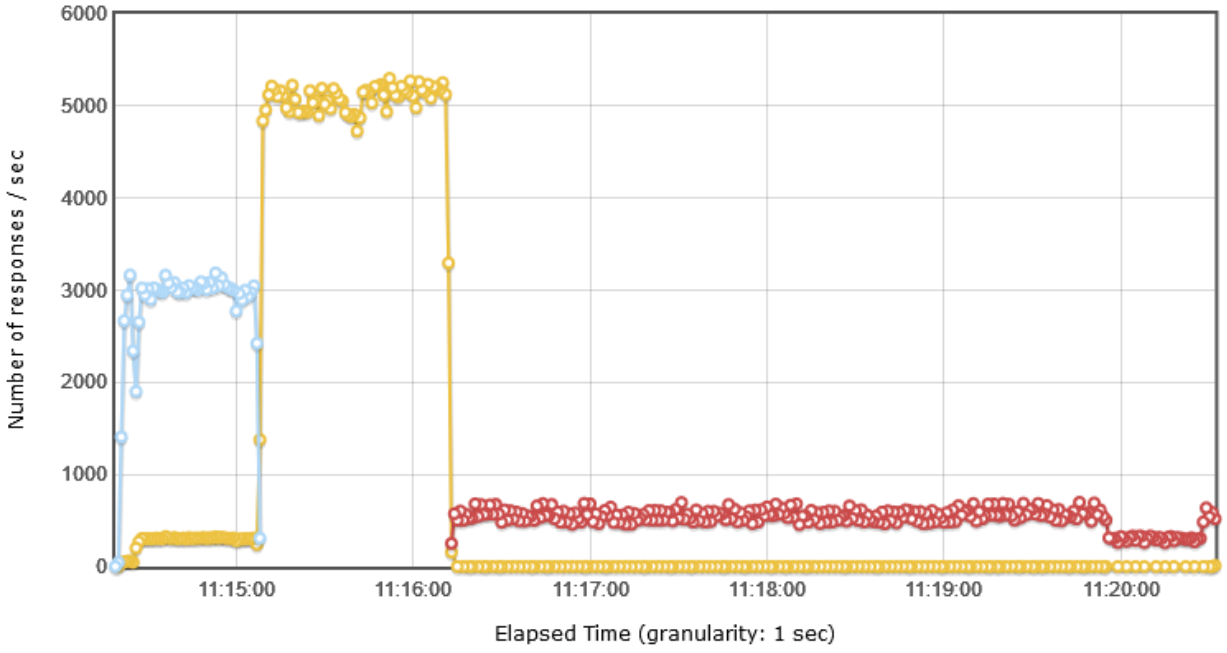
Spring. Veldig solide resultater i denne testen også. Det ser ikke ut som store forskjeller fra plan A, og DELETE ser like raskt ut i denne testen.

I grafen for forespørsler per sekund ser vi at det er et lite fall i hastigheten på POST, men det går raskt opp til grensen på 3000 forespørsler per sekund igjen. GET ligger rundt 5000 forespørsler per sekund igjen, men gjennomsnittet er nok litt høyere enn i plan A. DELETE ser ut til å ligge på omtrent 500 forespørsler per sekund, som er så vidt lavere enn sist.

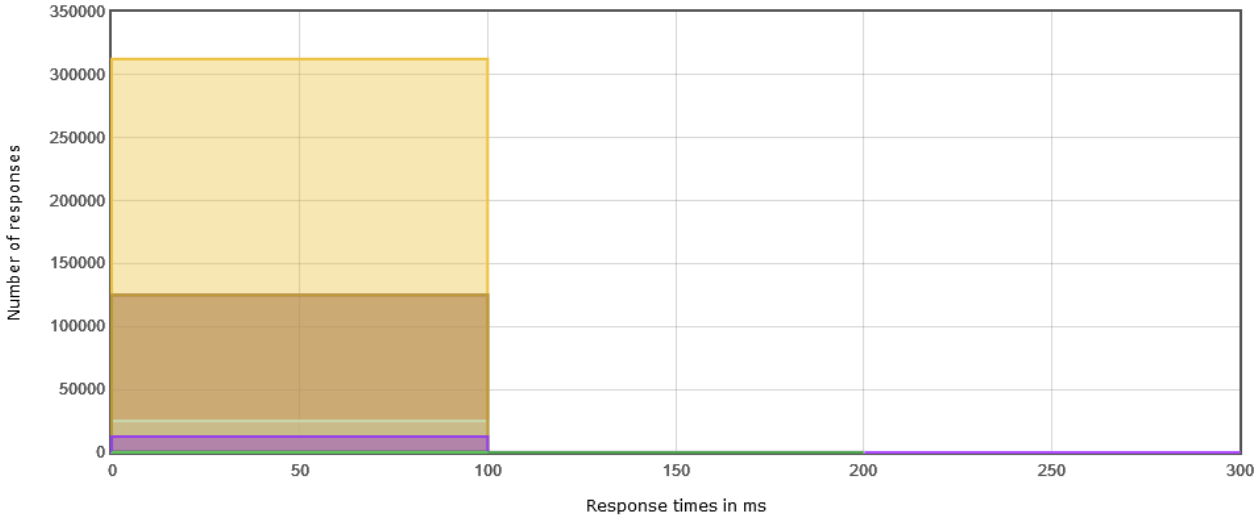
Nesten alle forespørsler har responstid under 100 ms, bare noen få er utenfor grupperingen. Gjennomsnittlig responstid er 2,11 ms, og responstid for GET posts er 1,66 ms.

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**613505**	**0**	**0.00%**	**2.11**	**0**	**207**	**3.00**	**4.00**	**5.00**	**1644.66**	**1627.25**	**841.68**
DELETE /comments /comment	125000	0	0.00%	1.54	1	10	2.00	2.00	3.00	562.14	204.61	275.03
DELETE /posts/post	12500	0	0.00%	2.78	1	13	4.00	4.00	5.00	338.55	123.22	164.65
DELETE /resubs /resub/posts	50	0	0.00%	1.60	1	3	2.00	2.45	3.00	625.00	227.31	310.67
DELETE /users/me	1	0	0.00%	2.00	2	2	2.00	2.00	2.00	500.00	182.13	241.21
GET /posts/post	312500	0	0.00%	1.66	0	26	2.00	3.00	4.00	4824.91	3729.84	2242.83
GET /resubs /resub/posts	25000	0	0.00%	5.52	2	29	9.00	10.00	12.00	233.06	1487.92	115.16
GET /users /user/comments	125	0	0.00%	181.71	150	207	195.40	201.00	206.22	0.57	176.09	0.28
GET /users /user/posts	25	0	0.00%	61.88	1	128	125.40	127.40	128.00	0.67	114.97	0.33
GET /users /user/resubs	751	0	0.00%	2.44	1	19	4.00	5.00	8.48	2.02	26.74	0.97
POST /auth/token	1	0	0.00%	163.00	163	163	163.00	163.00	163.00	6.13	4.92	1.89
POST /posts /post/comments	125000	0	0.00%	2.79	1	29	4.00	4.00	9.00	2940.62	2187.09	1880.96
POST /resubs/	50	0	0.00%	3.24	2	8	4.00	5.00	8.00	54.53	37.59	35.94
POST /resubs /resub/posts	12500	0	0.00%	2.97	1	32	5.00	6.00	10.00	2609.60	2017.36	1809.39
POST /users/	1	0	0.00%	126.00	126	126	126.00	126.00	126.00	7.94	4.32	1.94
setUp setProperties	1	0	0.00%	2.00	2	2	2.00	2.00	2.00	500.00	0.00	0.00

Figur 41. Repost API load plan B 10 threads Spring Boot oversikt



Figur 42. Repost API load plan B 10 threads Spring Boot forespørsler per sekund



Figur 43. Repost API load plan B 10 threads Spring Boot responstider

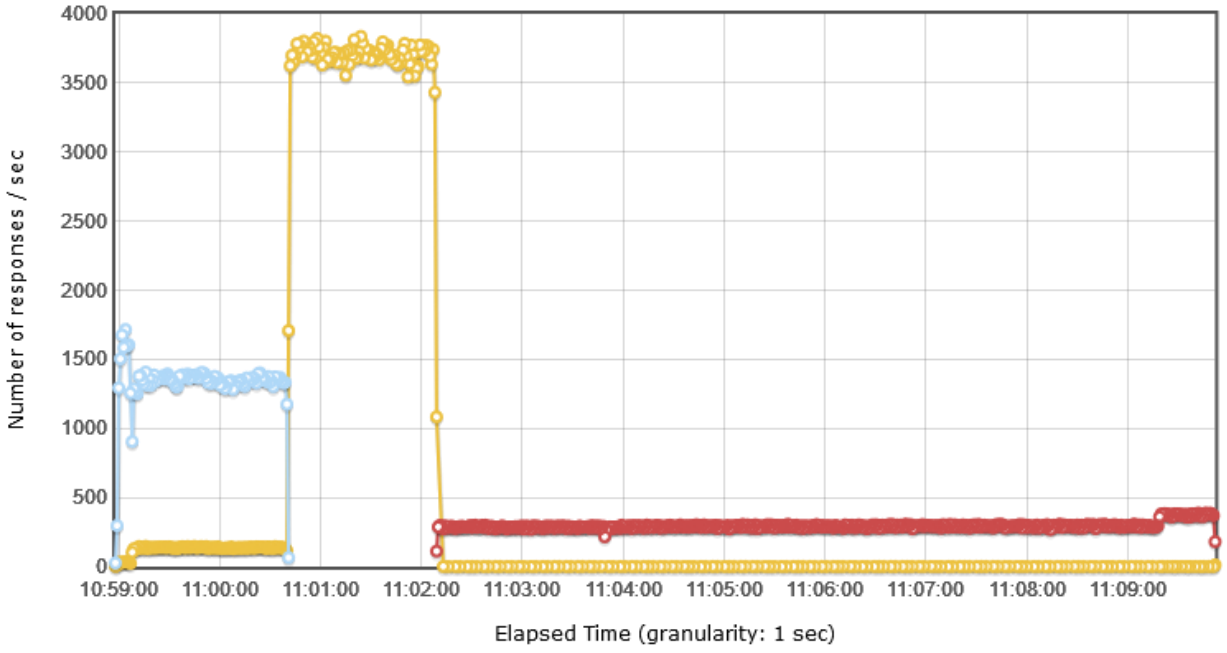
ASP.NET. Ser ut som relativt gode resultater med lite forskjeller fra plan A, men jevnt over litt lavere hastighet.

På grafen ser vi at POST øker kraftig til 1750 forespørsler per sekund, og så faller den ned og stabiliserer seg på 1350, med et punkt som er lavere. GET holder seg jevnt rundt 3700 forespørsler per sekund, litt under det vi fikk i plan A. DELETE er noe tregere i denne testen, rundt 300 forespørsler per sekund.

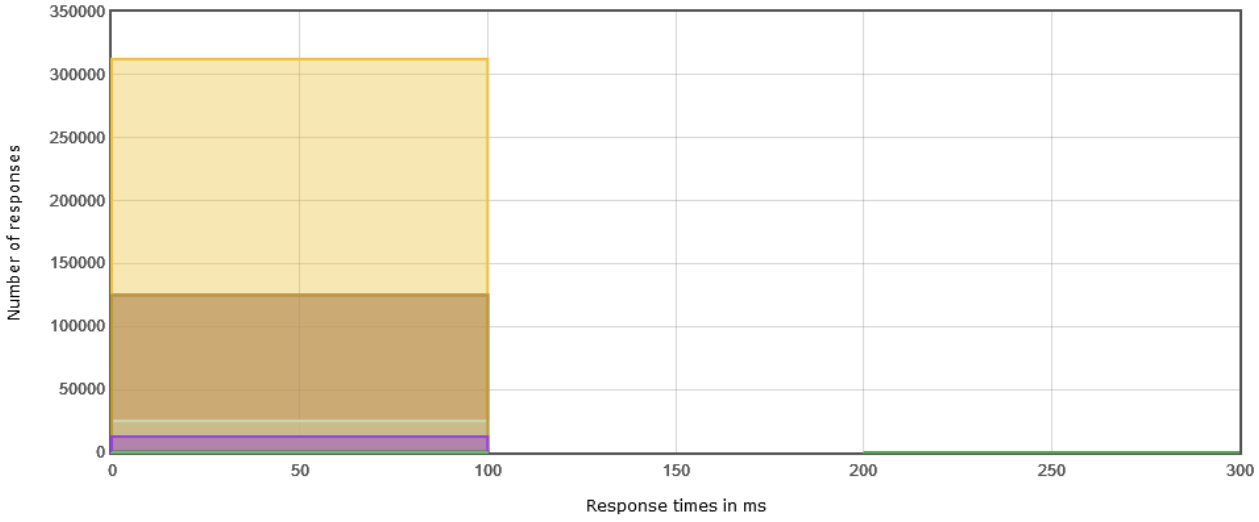
Grupperingen på responstider er utrolig bra. Gjennomsnittlig responstid er 3,74 ms, og responstid for GET posts er 2,53 ms.

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**613505**	**0**	**0.00%**	**3.74**	**1**	**237**	**4.00**	**4.00**	**4.00**	**937.44**	**685.33**	**846.18**
DELETE /comments /comment	125000	0	0.00%	3.33	2	27	4.00	4.00	5.00	291.06	23.02	256.38
DELETE /posts/post	12500	0	0.00%	2.58	2	10	3.00	3.00	4.00	375.69	29.72	329.46
DELETE /resubs /resub/posts	50	0	0.00%	2.34	2	3	3.00	3.00	3.00	413.22	32.69	367.22
DELETE /users/me	1	0	0.00%	5.00	5	5	5.00	5.00	5.00	200.00	15.82	174.80
GET /posts/post	312500	0	0.00%	2.53	1	32	4.00	4.00	5.00	3537.75	1747.79	3026.43
GET /resubs /resub/posts	25000	0	0.00%	4.67	2	22	6.00	7.00	9.00	137.72	838.19	121.99
GET /users /user/comments	125	0	0.00%	41.31	37	62	47.00	51.00	60.96	0.29	98.05	0.26
GET /users /user/posts	25	0	0.00%	14.60	2	29	28.00	28.70	29.00	0.75	127.22	0.67
GET /users /user/resubs	751	0	0.00%	4.37	2	12	6.00	7.00	9.00	1.15	14.89	1.00
POST /auth/token	1	0	0.00%	222.00	222	222	222.00	222.00	222.00	4.50	4.34	1.39
POST /posts /post/comments	125000	0	0.00%	6.82	3	31	9.00	10.00	13.00	1340.93	734.48	1381.52
POST /resubs/	50	0	0.00%	4.00	2	9	6.00	7.45	9.00	54.59	26.06	57.36
POST /resubs /resub/posts	12500	0	0.00%	5.92	3	32	8.00	9.00	13.00	1478.94	811.52	1604.59
POST /users/	1	0	0.00%	237.00	237	237	237.00	237.00	237.00	4.22	1.40	1.03
setUp setProperties	1	0	0.00%	1.00	1	1	1.00	1.00	1.00	1000.00	0.00	0.00

Figur 44. Repost API load plan B 10 threads ASP.NET oversikt



Figur 45. Repost API load plan B 10 threads ASP.NET forespørsler per sekund



Figur 46. Repost API load plan B 10 threads ASP.NET responstider

4.2.3.1 Oppsummering av Repost API load plan B 10 threads

Siden denne testen kjøres med 10 tråder i testen, og ikke 25 som i plan A, blir det litt vanskelig å sammenligne resultatene. Generelt sett hadde alle API-ene lavere responstid i denne testen enn de hadde i plan A. Det er Spring som er raskest med flest forespørsler per sekund og fortsatt lavest responstid. ASP.NET tar andreplass igjen, og FastAPI kommer sist. Denne gangen er FastAPI mye tregere på DELETE enn de to andre.

4.2.4 Repost API load plan B 25 threads

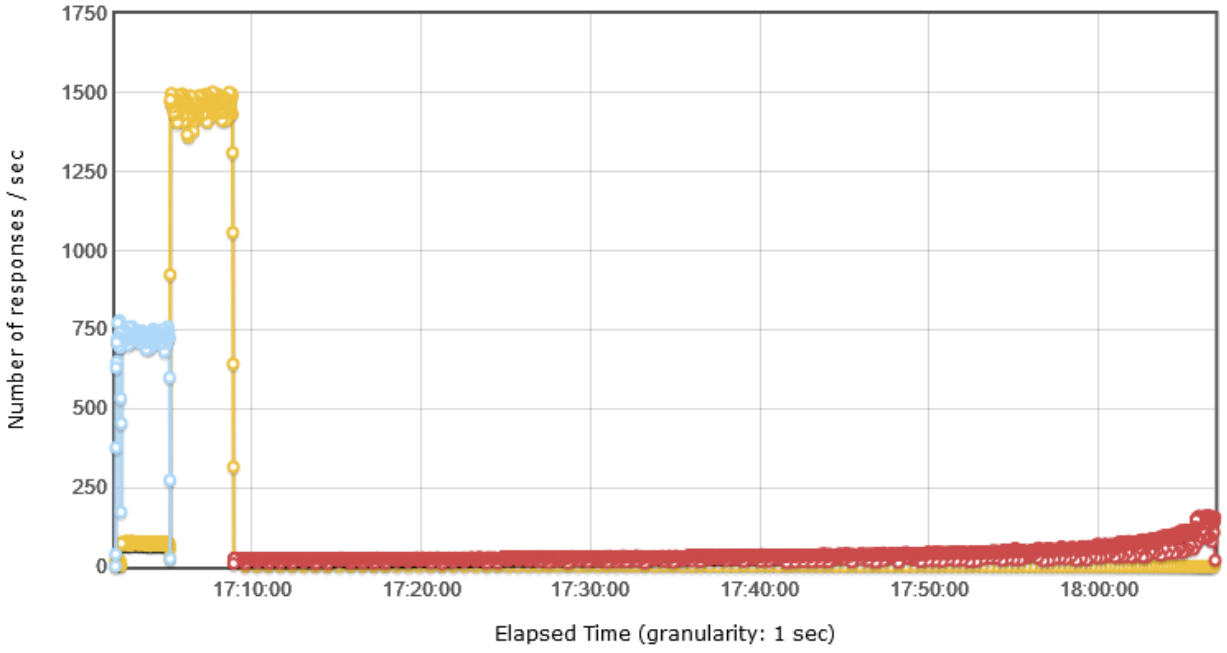
FastAPI. For FastAPI får vi nesten likt resultat for plan B med 10 og 25 tråder. Det er fortsatt veldig lav hastighet på DELETE, noe som går på én tråd i testen, men det er mye tregere enn i plan A.

POST ligger på 750 forespørsler per sekund og GET ligger på 1400. Vi ser at DELETE blir litt raskere mot slutten av testen her også. POST og GET har lite variasjon.

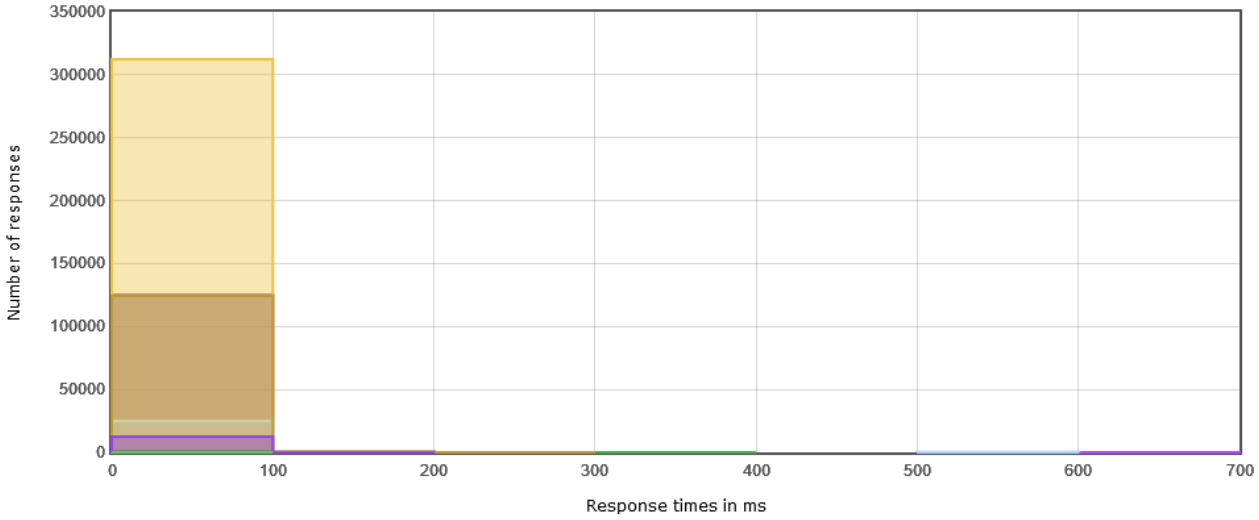
Gjennomsnittlig responstid er 21,93 ms, og responstid for GET posts er 15,35 ms. Dette er dårligere enn samme test med 10 tråder.

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**613505**	**0**	**0.00%**	**21.93**	**1**	**653**	**10.00**	**11.00**	**14.00**	**157.49**	**112.71**	**61.26**
DELETE /comments /comment	125000	0	0.00%	26.29	7	160	14.00	16.00	19.00	36.98	4.91	13.58
DELETE /posts/post	12500	0	0.00%	7.04	5	48	9.00	10.00	12.00	131.11	17.41	47.63
DELETE /resubs /resub/posts	50	0	0.00%	4.44	4	8	5.90	7.00	8.00	222.22	29.51	83.33
DELETE /users/me	1	0	0.00%	4.00	4	4	4.00	4.00	4.00	250.00	33.20	90.09
GET /posts/post	312500	0	0.00%	15.35	2	207	20.00	26.00	47.00	1389.84	648.77	475.04
GET /resubs /resub/posts	25000	0	0.00%	42.06	8	245	68.00	78.00	99.00	62.85	387.00	23.38
GET /users /user/comments	125	0	0.00%	584.56	529	653	629.00	639.70	651.18	0.04	12.60	0.01
GET /users /user/posts	25	0	0.00%	273.88	2	582	563.60	577.20	582.00	0.26	44.95	0.10
GET /users /user/resubs	751	0	0.00%	31.11	6	152	63.00	83.40	110.00	0.19	2.55	0.07
POST /auth/token	1	0	0.00%	264.00	264	264	264.00	264.00	264.00	3.79	1.28	1.17
POST /posts /post/comments	125000	0	0.00%	29.92	6	253	46.00	57.00	77.99	722.63	337.32	373.31
POST /resubs/	50	0	0.00%	11.76	6	32	28.90	29.90	32.00	51.39	19.97	27.60
POST /resubs /resub/posts	12500	0	0.00%	30.96	6	190	57.00	68.00	95.00	659.35	311.00	376.68
POST /users/	1	0	0.00%	338.00	338	338	338.00	338.00	338.00	2.96	0.72	0.72
setUp setProperties	1	0	0.00%	1.00	1	1	1.00	1.00	1.00	1000.00	0.00	0.00

Figur 47. Repost API load plan B 25 threads FastAPI oversikt



Figur 48. Repost API load plan B 25 threads FastAPI forespørsler per sekund



Figur 49. Repost API load plan B 25 threads FastAPI responstider

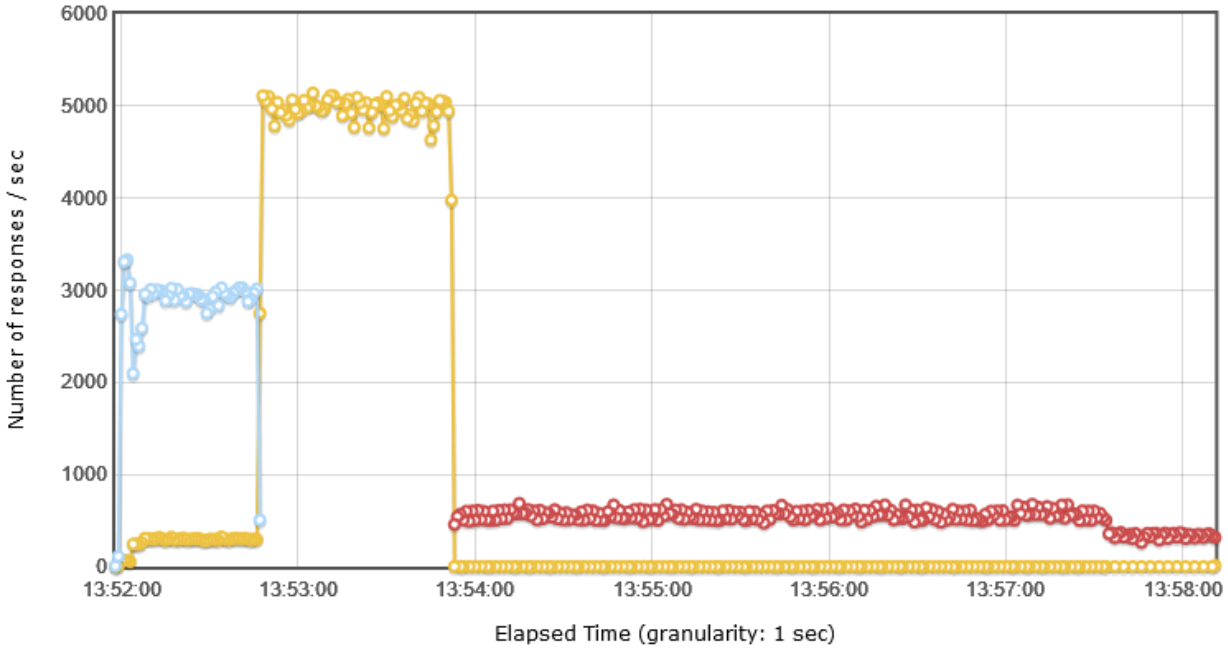
Spring. Resultatene for 25 tråder ser veldig like ut som for 10 tråder.

Som i tidligere tester er Spring litt ujevn de første sekundene, men jevner seg for ut på høye verdier. POST ligger rundt 3000 forespørsler per sekund, GET på 5000. DELETE er også ganske lik og er på omtrent 500 forespørsler per sekund igjen.

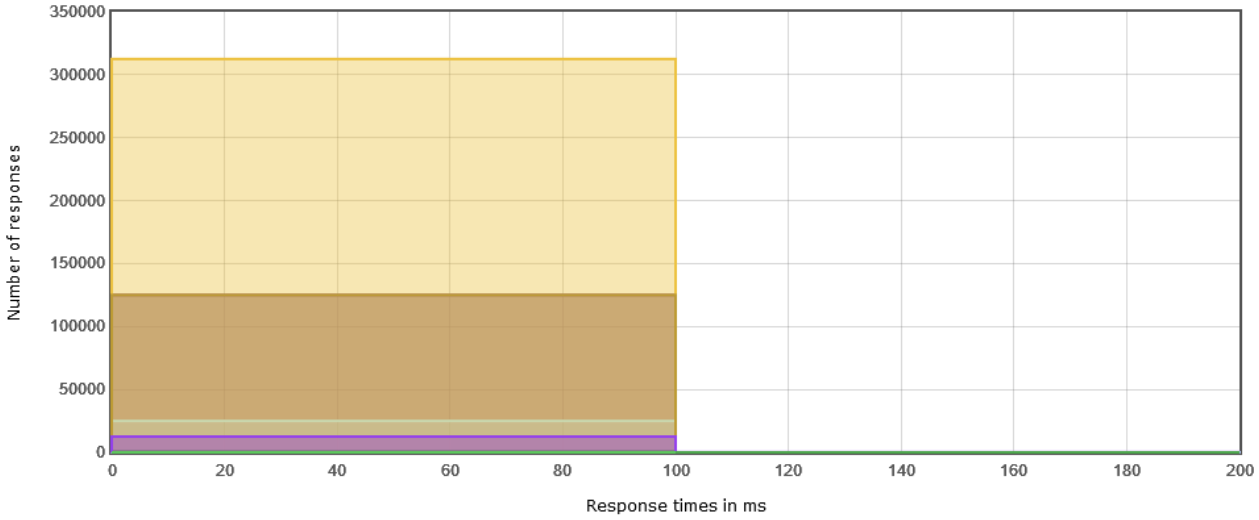
Bare et par forespørsler har over 100 ms responstid, noe som gjør grupperingen veldig bra. Gjennomsnittlig responstid er 4,81 ms, og responstid for GET posts er 4,58 ms.

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**613505**	**0**	**0.00%**	**4.81**	**0**	**191**	**3.00**	**4.00**	**4.00**	**1643.33**	**1625.93**	**841.00**
DELETE /comments /comment	125000	0	0.00%	1.56	1	9	2.00	2.00	3.00	563.72	205.18	275.81
DELETE /posts/post	12500	0	0.00%	2.77	2	9	3.00	4.00	4.00	339.11	123.43	164.92
DELETE /resubs /resub/posts	50	0	0.00%	2.70	2	4	3.00	4.00	4.00	349.65	127.16	173.80
DELETE /users/me	1	0	0.00%	3.00	3	3	3.00	3.00	3.00	333.33	121.42	160.81
GET /posts/post	312500	0	0.00%	4.58	0	50	8.00	9.00	13.00	4762.05	3681.23	2213.61
GET /resubs /resub/posts	25000	0	0.00%	10.47	2	55	16.00	18.00	23.00	229.54	1465.44	113.42
GET /users /user/comments	125	0	0.00%	154.90	142	191	167.00	173.70	189.44	0.57	176.64	0.28
GET /users /user/posts	25	0	0.00%	62.20	1	137	129.00	135.50	137.00	0.68	115.18	0.33
GET /users /user/resubs	751	0	0.00%	5.78	1	36	10.00	12.00	21.00	2.02	26.72	0.97
POST /auth/token	1	0	0.00%	170.00	170	170	170.00	170.00	170.00	5.88	4.72	1.81
POST /posts /post/comments	125000	0	0.00%	7.43	1	59	12.00	14.00	20.00	2886.44	2146.78	1846.30
POST /resubs/	50	0	0.00%	3.42	1	6	5.00	5.00	6.00	51.65	35.61	34.05
POST /resubs /resub/posts	12500	0	0.00%	6.04	1	54	11.00	14.00	21.00	3104.05	2399.56	2152.22
POST /users/	1	0	0.00%	124.00	124	124	124.00	124.00	124.00	8.06	4.39	1.97
setUp setProperties	1	0	0.00%	1.00	1	1	1.00	1.00	1.00	1000.00	0.00	0.00

Figur 50. Repost API load plan B 25 threads Spring Boot oversikt



Figur 51. Repost API load plan B 25 threads Spring Boot forespørsler per sekund



Figur 52. Repost API load plan B 25 threads Spring Boot forespørsler per sekund

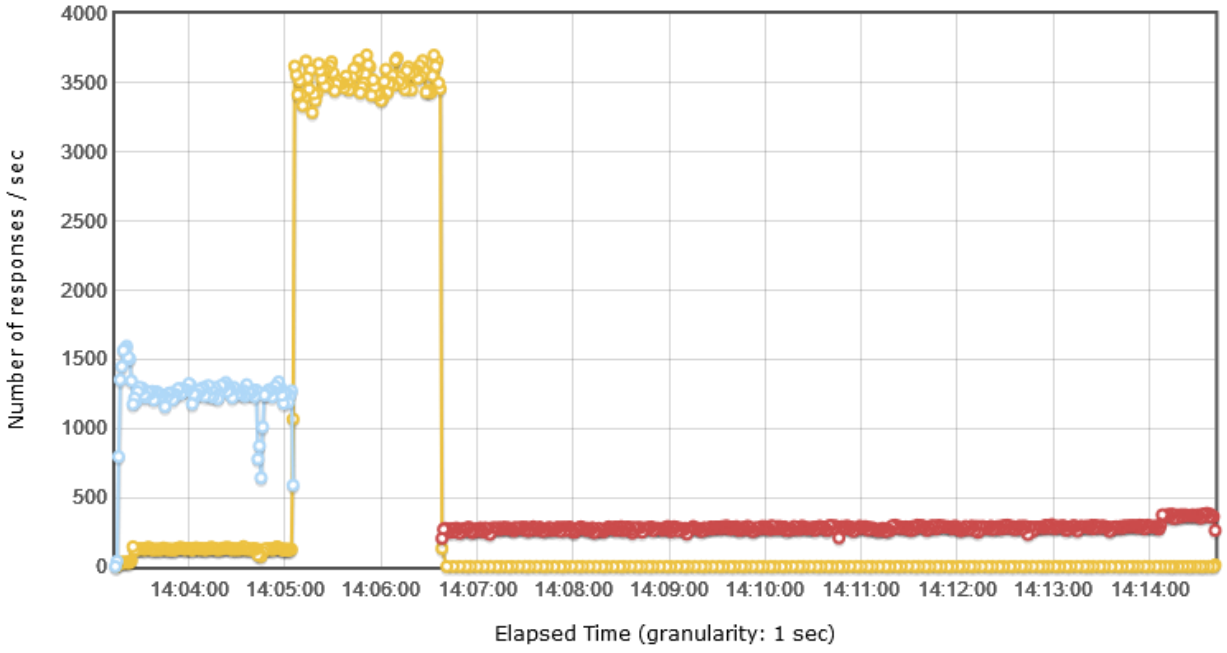
ASP.NET. Disse resultatene ser ganske like ut som samme test med 10 tråder, med noen endringer.

På grafen ser vi at POST øker kraftig til 1500 forespørsler per sekund, og så faller den ned til 1250, med et punkt som er lavere. GET holder seg jevnt rundt 3500 forespørsler per sekund, litt under det vi fikk i plan A. DELETE ligger rundt 300 forespørsler per sekund.

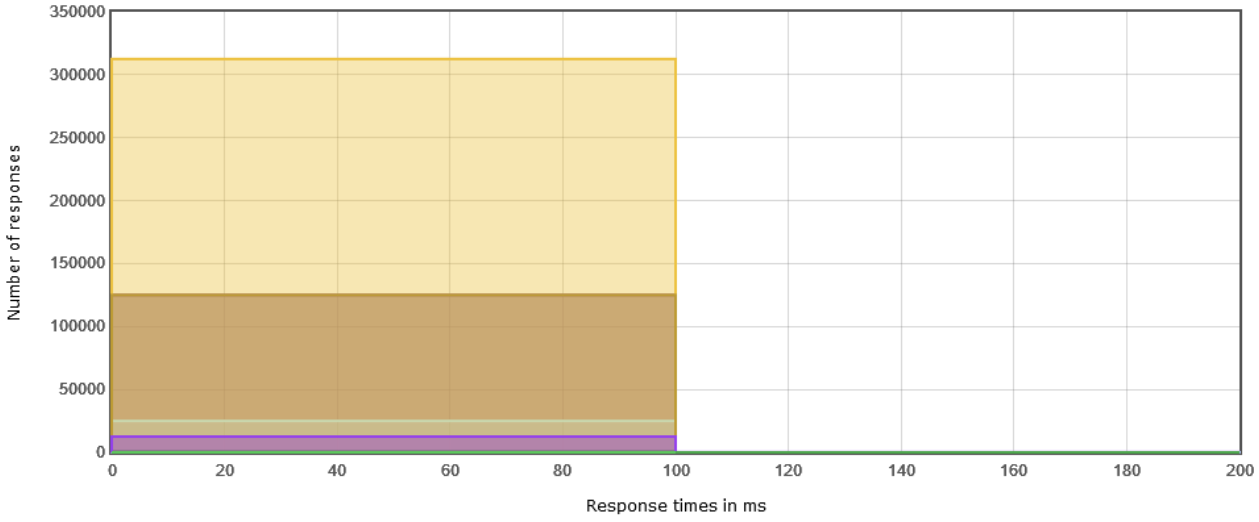
Grupperingen av responstider er nærmest perfekt med bare to kall over 100 ms. Gjennomsnittlig responstid er 8,82 ms, og responstid for GET posts er 6,86 ms.

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**613505**	**0**	**0.00%**	**8.82**	**1**	**244**	**4.00**	**4.00**	**5.00**	**892.93**	**654.25**	**806.64**
DELETE /comments /comment	125000	0	0.00%	3.48	3	82	4.00	4.00	5.00	278.07	22.00	244.94
DELETE /posts/post	12500	0	0.00%	2.62	2	9	3.00	3.00	4.00	369.36	29.22	324.28
DELETE /resubs /resub/posts	50	0	0.00%	2.54	2	4	3.00	4.00	4.00	370.37	29.30	329.14
DELETE /users/me	1	0	0.00%	8.00	8	8	8.00	8.00	8.00	125.00	9.89	109.25
GET /posts/post	312500	0	0.00%	6.86	1	34	9.00	9.00	11.00	3377.10	1671.68	2892.30
GET /resubs /resub/posts	25000	0	0.00%	12.57	2	71	18.00	21.00	26.00	129.29	789.08	114.52
GET /users /user/comments	125	0	0.00%	42.07	37	62	48.40	55.70	61.48	0.28	93.92	0.25
GET /users /user/posts	25	0	0.00%	16.76	3	41	31.20	38.60	41.00	0.74	125.36	0.65
GET /users /user/resubs	751	0	0.00%	12.48	3	39	20.00	22.00	29.00	1.10	14.18	0.96
POST /auth/token	1	0	0.00%	244.00	244	244	244.00	244.00	244.00	4.10	3.95	1.26
POST /posts /post/comments	125000	0	0.00%	18.27	3	85	24.00	26.00	29.00	1239.48	680.12	1278.21
POST /resubs/	50	0	0.00%	4.26	2	10	5.90	6.45	10.00	52.19	24.92	54.84
POST /resubs /resub/posts	12500	0	0.00%	14.94	3	65	21.00	24.00	29.00	1473.71	811.53	1598.92
POST /users/	1	0	0.00%	243.00	243	243	243.00	243.00	243.00	4.12	1.36	1.00
setUp setProperties	1	0	0.00%	1.00	1	1	1.00	1.00	1.00	1000.00	0.00	0.00

Figur 53. Repost API load plan B 25 threads ASP.NET oversikt



Figur 54. Repost API load plan B 25 threads ASP.NET forespørsler per sekund



Figur 55. Repost API load plan B 25 threads ASP.NET responstider

4.2.4.1 Oppsummering av Repost API load plan B 25 threads

Det er noe høyere responstid på alle API-ene når testen kjøres med 25 tråder i stedet for 10. Spring har samme antall forespørsler per sekund, men litt høyere responstid. ASP.NET har litt lavere ytelse, både i form av forespørsler per sekund og responstid. FastAPI er fortsatt sist, men har ikke særlig forskjellige resultater mellom 10 og 25 tråder i testen.

4.2.5 Repost API load plan B 50 threads

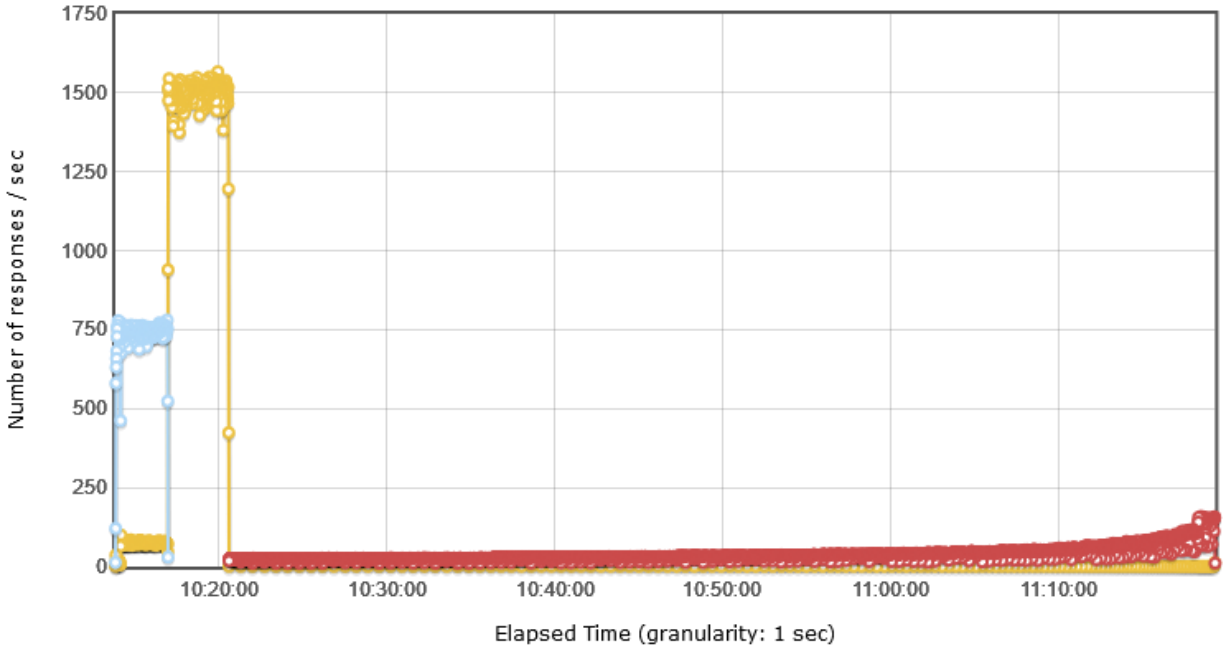
FastAPI. Her har vi litt mer variasjon, men det ser ut som at det totalt sett er raskere enn testene med 10 og 25 tråder.

POST er litt ustabil men ligger på 750 forespørsler per sekund, GET på 1500. GET er mer variert enn ved tidligere tester. DELETE har samme problem her som på alle FastAPI resultatene for plan B testen.

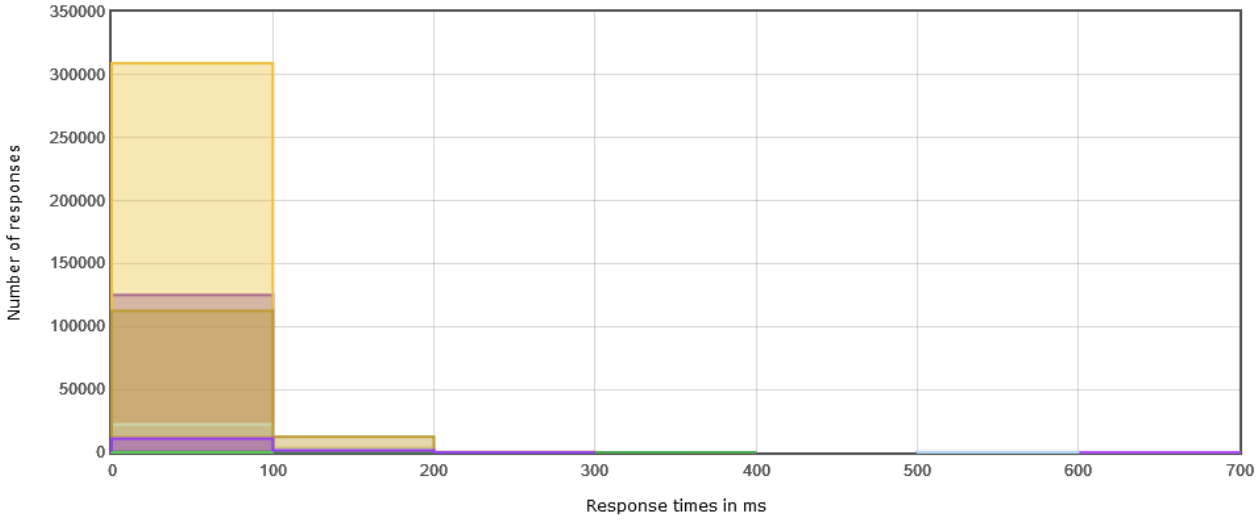
Grupperingen på responstider er dårligere her, med en del responser på 200 ms og noen flere avvik. Gjennomsnittlig responstid er 37,52 ms, og gjennomsnittlig responstid for GET posts er 31,09 ms.

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**613505**	**0**	**0.00%**	**37.52**	**1**	**654**	**11.00**	**12.00**	**15.00**	**156.15**	**111.76**	**60.74**
DELETE /comments /comment	125000	0	0.00%	26.61	7	82	15.00	17.00	20.00	36.54	4.85	13.42
DELETE /posts/post	12500	0	0.00%	7.49	5	53	9.00	10.00	12.00	123.46	16.40	44.85
DELETE /resubs /resub/posts	50	0	0.00%	4.36	4	6	5.90	6.00	6.00	227.27	30.18	85.23
DELETE /users/me	1	0	0.00%	3.00	3	3	3.00	3.00	3.00	333.33	44.27	120.12
GET /posts/post	312500	0	0.00%	31.09	3	277	42.00	54.00	83.00	1436.81	670.70	491.10
GET /resubs /resub/posts	25000	0	0.00%	65.03	8	288	101.00	116.00	154.00	64.73	398.59	24.08
GET /users /user/comments	125	0	0.00%	589.89	526	654	624.40	631.00	651.14	0.04	12.46	0.01
GET /users /user/posts	25	0	0.00%	279.64	2	590	582.40	589.70	590.00	0.25	42.34	0.09
GET /users /user/resubs	751	0	0.00%	63.83	6	264	122.00	146.00	193.44	0.19	2.53	0.07
POST /auth/token	1	0	0.00%	276.00	276	276	276.00	276.00	276.00	3.62	1.22	1.11
POST /posts /post/comments	125000	0	0.00%	58.84	6	308	91.00	110.00	144.00	740.69	345.75	382.64
POST /resubs/	50	0	0.00%	10.02	8	18	12.90	15.45	18.00	50.45	19.61	27.10
POST /resubs /resub/posts	12500	0	0.00%	61.72	7	356	109.00	126.00	171.00	706.89	333.43	403.84
POST /users/	1	0	0.00%	326.00	326	326	326.00	326.00	326.00	3.07	0.75	0.75
setUp setProperties	1	0	0.00%	1.00	1	1	1.00	1.00	1.00	1000.00	0.00	0.00

Figur 56. Repost API load plan B 50 threads FastAPI oversikt



Figur 57. Repost API load plan B 50 threads FastAPI forespørsler per sekund



Figur 58. Repost API load plan B 50 threads FastAPI responstider

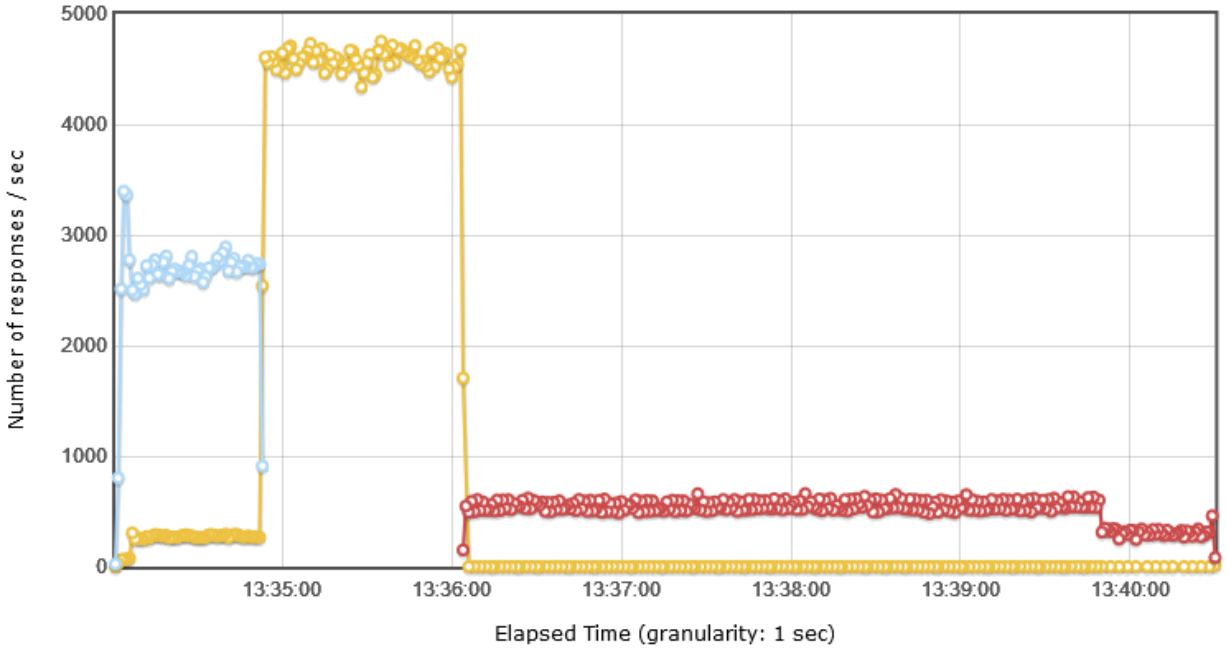
Spring. Disse resultatene ser ut til å være noe lavere enn for samme test med 10 og 25 tråder.

Forespørsler per sekund ligger på 2700 for POST og 4600 for GET. DELETE ser ut til å være som før, 500 forespørsler per sekund.

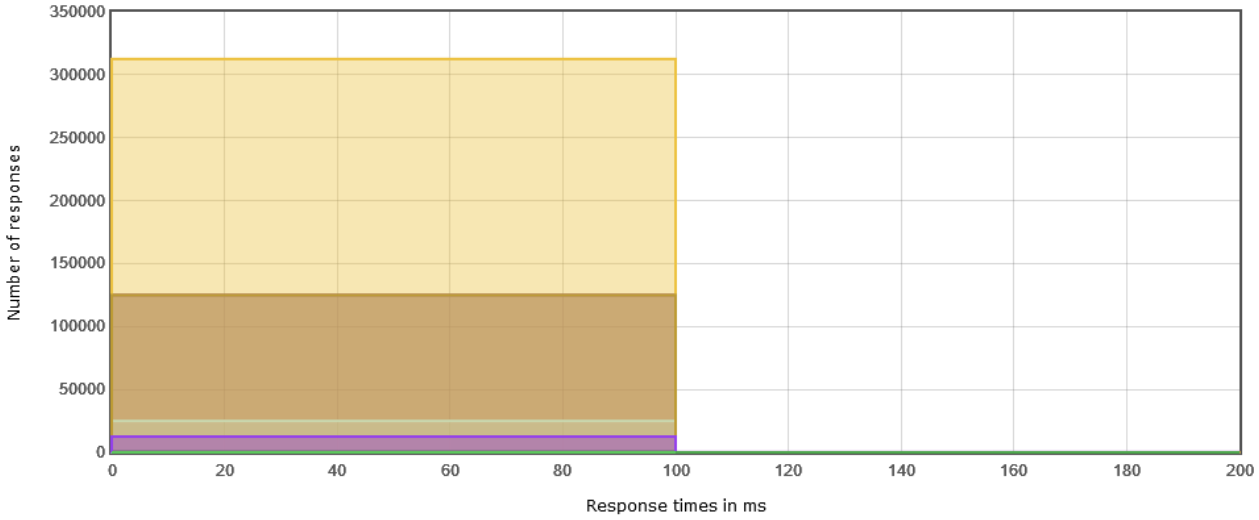
Selv om gjennomsnittet økte litt er grupperingen fortsatt veldig bra med nesten alle responser under 100 millisekunder. Gjennomsnittlig responstid er 10,01 ms, og gjennomsnittlig responstid for GET posts er 10,37 ms.

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**613505**	**0**	**0.00%**	**10.01**	**0**	**179**	**4.00**	**4.00**	**5.00**	**1577.45**	**1560.72**	**807.28**
DELETE /comments /comment	125000	0	0.00%	1.60	1	15	2.00	2.00	2.00	554.55	201.84	271.32
DELETE /posts/post	12500	0	0.00%	3.01	1	11	4.00	4.00	5.00	312.87	113.88	152.16
DELETE /resubs /resub/posts	50	0	0.00%	1.44	1	2	2.00	2.00	2.00	666.67	242.46	331.38
DELETE /users/me	1	0	0.00%	2.00	2	2	2.00	2.00	2.00	500.00	182.13	241.21
GET /posts/post	312500	0	0.00%	10.37	0	116	20.00	26.00	40.00	4398.19	3399.92	2044.47
GET /resubs /resub/posts	25000	0	0.00%	17.51	2	124	30.00	37.00	52.00	212.52	1356.75	105.02
GET /users /user/comments	125	0	0.00%	145.74	127	179	153.40	165.60	178.74	0.56	173.70	0.28
GET /users /user/posts	25	0	0.00%	61.68	1	134	124.80	131.60	134.00	0.62	106.31	0.31
GET /users /user/resubs	751	0	0.00%	12.34	1	84	26.00	32.00	47.96	1.94	25.64	0.93
POST /auth/token	1	0	0.00%	164.00	164	164	164.00	164.00	164.00	6.10	4.89	1.88
POST /posts /post/comments	125000	0	0.00%	16.28	1	149	29.00	38.00	52.00	2682.58	1995.17	1715.91
POST /resubs/	50	0	0.00%	4.22	3	7	5.00	6.00	7.00	50.76	34.99	33.46
POST /resubs /resub/posts	12500	0	0.00%	13.14	1	121	26.00	34.00	55.00	2971.94	2297.62	2060.63
POST /users/	1	0	0.00%	122.00	122	122	122.00	122.00	122.00	8.20	4.46	2.00
setUp setProperties	1	0	0.00%	1.00	1	1	1.00	1.00	1.00	1000.00	0.00	0.00

Figur 59. Repost API load plan B 50 threads Spring Boot oversikt



Figur 60. Repost API load plan B 50 threads Spring Boot forespørsler per sekund



Figur 61. Repost API load plan B 50 threads Spring Boot responstider

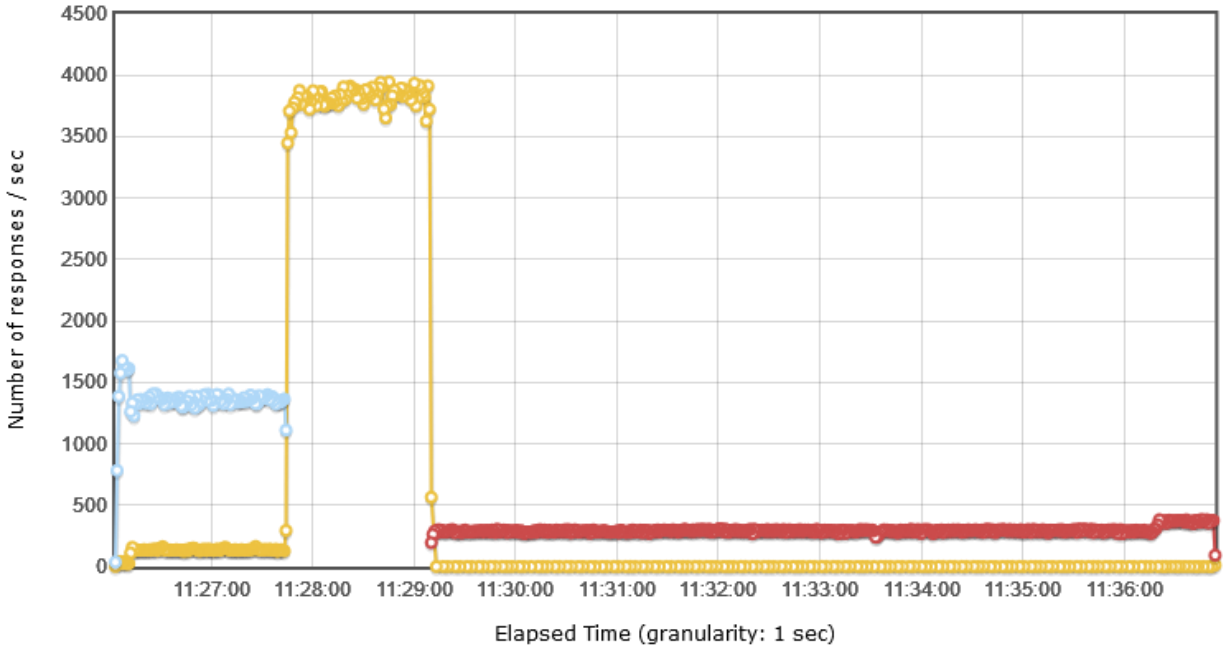
ASP.NET. Resultatene for ASP.NET ser bedre ut med 50 tråder enn med 10 eller 25 tråder.

POST ligger på 1400 forespørsler per sekund og GET er rundt 3750. DELETE holder seg likt, 300 forespørsler per sekund.

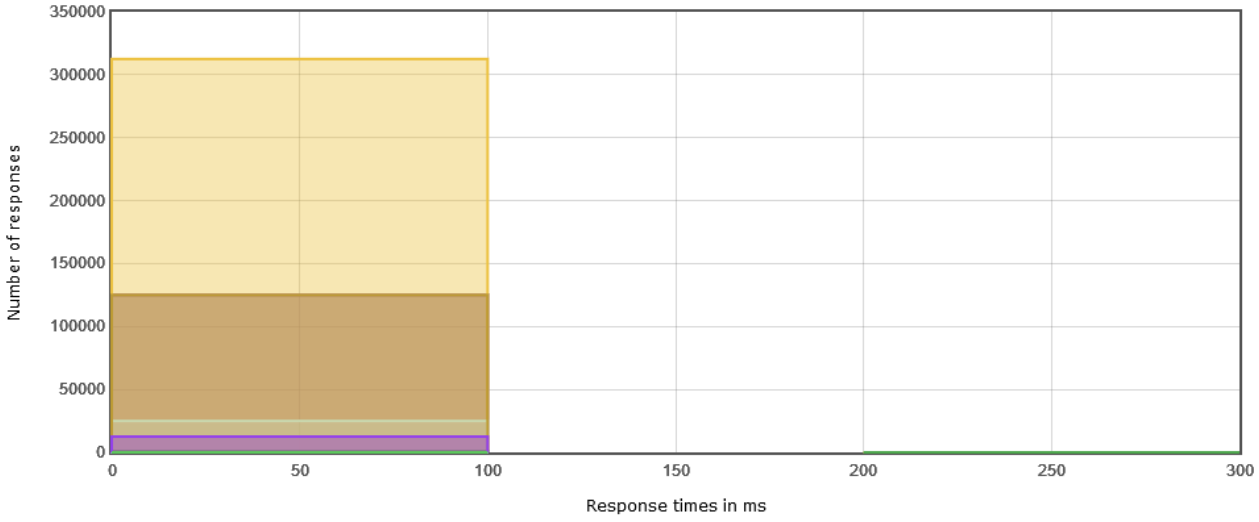
Gruppering på responstid er veldig bra med bare et par responser over 100 ms. Gjennomsnittlig responstid er 15,6 ms, og responstid for GET posts er 12,8 ms.

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**613505**	**0**	**0.00%**	**15.60**	**1**	**235**	**4.00**	**4.00**	**4.00**	**941.71**	**688.44**	**850.03**
DELETE /comments /comment	125000	0	0.00%	3.33	2	31	4.00	4.00	5.00	290.60	22.99	255.98
DELETE /posts/post	12500	0	0.00%	2.61	2	9	3.00	3.00	4.00	370.59	29.31	324.99
DELETE /resubs /resub/posts	50	0	0.00%	2.30	2	3	3.00	3.00	3.00	427.35	33.80	379.77
DELETE /users/me	1	0	0.00%	4.00	4	4	4.00	4.00	4.00	250.00	19.78	218.51
GET /posts/post	312500	0	0.00%	12.80	1	41	16.00	18.00	20.00	3664.36	1810.31	3134.74
GET /resubs /resub/posts	25000	0	0.00%	23.18	3	79	32.00	39.00	47.00	140.66	856.07	124.59
GET /users /user/comments	125	0	0.00%	38.24	34	61	43.00	44.70	60.74	0.29	97.88	0.26
GET /users /user/posts	25	0	0.00%	14.72	2	32	28.40	31.10	32.00	0.74	125.51	0.66
GET /users /user/resubs	751	0	0.00%	23.18	2	53	37.00	42.00	49.00	1.16	14.96	1.01
POST /auth/token	1	0	0.00%	201.00	201	201	201.00	201.00	201.00	4.98	4.80	1.53
POST /posts /post/comments	125000	0	0.00%	33.33	3	80	45.00	47.00	53.00	1351.63	740.35	1392.55
POST /resubs/	50	0	0.00%	5.56	4	8	7.00	7.00	8.00	50.81	24.26	53.39
POST /resubs /resub/posts	12500	0	0.00%	28.30	3	74	39.00	43.00	49.00	1543.78	847.10	1674.94
POST /users/	1	0	0.00%	235.00	235	235	235.00	235.00	235.00	4.26	1.40	1.04
setUp setProperties	1	0	0.00%	2.00	2	2	2.00	2.00	2.00	500.00	0.00	0.00

Figur 62. Repost API load plan B 50 threads ASP.NET oversikt



Figur 63. Repost API load plan B 50 threads ASP.NET forespørsler per sekund



Figur 64. Repost API load plan B 50 threads ASP.NET responstider

4.2.5.1 Oppsummering for Repost API load plan B 50 threads

I denne testen, med 50 tråder, var det en del variasjoner fra resultatene med 10 og 25 tråder. FastAPI hadde litt høyere ytelse med tanke på forespørsler per sekund, men hadde den største endringen i gruppering av responstid. Selv om alle API-ene hadde lengre gjennomsnittlig responstid med flere tråder i testen hadde de fortsatt ikke merkverdig mange responser som ikke var under 100 millisekunder. ASP.NET hadde også en god forbedring på antall forespørsler per sekund som kunne håndteres. Responstiden gikk også litt opp for ASP.NET, men det var nesten ingen responser over 100 millisekund av den grunn. Igjen så er det Spring som er raskest, men i denne testen gikk ytelsen ned en god del i forhold til de tidligere testene. Spring hadde fortsatt den beste responstiden i gjennomsnitt, og flest forespørsler per sekund.

4.3 Brukeropplevelse for en utvikler som bruker de tre rammeverkene

I innledningen satte vi noen kriterier for at det skulle være enkelt å bruke rammeverket for en utvikler. Nå som vi har brukt rammeverkene til å lage API-er kan vi se om alle kriteriene er møtt.

- OpenAPI dokumentasjon med støtte for Swagger UI

  I alle tre rammeverkene fikk vi integrert OpenAPI og Swagger UI.

  FastAPI: Bra. OpenAPI og Swagger UI er lagt inn som standard i FastAPI og det fungerer fint.

  Spring Boot: For det meste bra. Vi må legge til en avhengighet og så fungerer det fint, med ett unntak.

  ASP.NET: Bra. Vi må legge til en avhengighet og endre noe konfigurasjon for at tekst skal bli inkludert, det fungerer fint.

- OAuth2 autentisering med password flow

  FastAPI: Bra. God oppskrift på hvordan det settes opp.

  Spring Boot: Vanskelig. Vanskelig å finne dokumentasjon for OAuth2 med den konfigurasjonen vi ville bruke.

  ASP.NET: Greit. Litt vanskelig å finne riktig dokumentasjon.

- ORM for å lage tabeller og relasjoner fra klasser

  FastAPI: Bra. Enkelt å lage datamodeller.

  Spring Boot: Bra. Det er veldig enkelt å lage datamodeller.

  ASP.NET: Veldig bra. Enkelt å lage datamodeller. Veldig integrert med språket.

- ORM for å lage spørringer til databasen

  FastAPI: Bra. Her kan vi lage spørringer med å spesifisere parametre og modellen som skal brukes.

  Spring Boot: Bra. For å lage spørringer lager man en metode i datarepositorien og metoden blir automatisk implementert for oss.

  ASP.NET: Greit. Problemer med bruk av lazy loading og dermed må alle queries spesifisere hvilke relasjoner som må lastes inn.

- God og veiledende dokumentasjon for bruk av rammeverket

FastAPI: Bra. Enkel dokumentasjon som har gode veiledninger for nybegynnere og avanserte brukere. Ingen komplett API referanse.

Spring Boot: Greit. Veldig mye dokumentasjon med mange gode kilder, men også en del utdatert materiale som kan gjøre det vanskeligere å finne den beste løsningen på et problem.

ASP.NET: Bra. Mye veiledende dokumentasjon laget spesifikt for nyeste versjon.

- Legger opp for lite gjenbruk av kode

  FastAPI: Bra. Dependency systemet i FastAPI gjør det enkelt å gjenbruke kode. Kan ikke kombinere API og database modeller.

  Spring Boot: Bra. Databasemodeller og API skjema kan knyttes mot samme objekt.

  ASP.NET: Bra. Databasemodeller og API skjema kan knyttes mot samme objekt, og kan enkelt gjenbruke kode i kontrollere.

- Raskt og enkelt å teste nye endringer i koden

  Denne funksjonaliteten er tilgjengelig i alle rammeverkene.

  FastAPI: Bra.

  Spring Boot: Bra.

  ASP.NET: Bra.

- Enkelt å modellere endepunkter og parametere i endepunktene

  Alle rammeverkene gjør det enkelt å modellere endepunktene slik vi vil, med de parametrene vi ønsker.

  FastAPI: Bra.

  Spring Boot: Bra.

  ASP.NET: Bra.

- Enkelt å spesifisere formatering av navn i API skjema uavhengig av programmeringsspråket

  Alle rammeverkene har støtte for å ha forskjellig navn på API skjema og interne navn på feltene.

  FastAPI: Bra.

  Spring Boot: Bra.

ASP.NET: Bra.

- Enkelt å bygge og kjøre applikasjonen i et testmiljø

  FastAPI: Greit. For å sette opp applikasjonen i et eget miljø må man manuelt sette opp et virtuelt miljø. Ellers enkelt.

  Spring Boot: Bra. Veldig enkelt.

  ASP.NET: Bra. Veldig enkelt.

- God støtte for ekstra funksjonalitet gjennom tredjepartsbiblioteker

  Alle rammeverkene har god støtte for tredjepartsbiblioteker. Om vi manglet noe funksjonalitet var det ikke noe problem å finne en godt dokumentert løsning gjennom en tredjepart.

  FastAPI: Bra.

  Spring Boot: Bra.

  ASP.NET: Bra.

- Støtte for alle populære operativsystem slik at utvikleren kan jobbe i sitt foretrukne miljø

  Alle rammeverkene har støtte for alle de populære operativsystemene. Vi har brukt både Linux og Windows som utviklingsmiljø, og Linux som testmiljø. Det eneste som ikke er støttet i Windows er Gunicorn som brukes for å kjøre FastAPI med flere Uvicorn workers. Dette gjør at det er mindre publiseringsmuligheter om man velger Windows som server.

  FastAPI: Bra.

  Spring Boot: Bra.

  ASP.NET: Bra.

Alle kriteriene som var satt ble møtt i de forskjellige rammeverkene. For det meste var det bra støtte for det vi ønsket at skulle være med. Det er noen punkter som ikke var like bra i alle rammeverkene, men alt tatt i betraktning fikk vi laget API-et slik vi ville i alle tre rammeverkene.

Et annet punkt vi nevnte var at vi ville se antall linjer koder for hver implementasjon. Med pluginen Statistic får vi hentet ut disse tallene.

API	Fil	Totalt linjer	Kode linjer	Kode %	Kommentar linjer	Kommentar %	Blanke linjer	Blank %
FastAPI	Totalt	1217	712	59 %	170	14 %	335	28 %
Spring Boot	Totalt	2264	1751	77 %	214	9 %	299	13 %
ASP.NET	Totalt	1619	1298	80 %	120	7 %	201	12 %

Oppsummeringen av tallene viser at Spring Boot har flest linjer totalt, så kommer ASP.NET og FastAPI. Antall prosent av linjene som er kode er minst for FastAPI. Spring Boot og ASP.NET har nesten lik prosent av linjer som er kode. FastAPI har flere linjer med kommentarer og blanke linjer relativt til totalt antall linjer. Her er Spring Boot og ASP.NET relativt like igjen.

Når vi ser hvilke filer som har flest linjer i de forskjellige rammeverkene ser vi at det er trender blant de tre. Det er kontrollere som har flest linjer kode i alle rammeverkene. Spring Boot sine service filer og FastAPI sine CRUD filer er deretter nest høyest oppe. Videre er det forskjeller, med noen som har flere konfigurasjonsfiler og så videre.

4.4 Nettside utviklet i Vue.js

Dette kapittelet viser resultatet av demo nettsiden utviklet i Vue.js og de forskjellige sidene og komponentene som ble laget.

4.4.1 Forside

Forsiden til nettsiden består av en navigasjonsbar på toppen med en logo på venstre side. Navigasjonsbaren har lenke til en liste med andre resuber som brukeren kan besøke om de ikke finner det i forsiden. Her er også menyvalg for å logge inn, lage en ny bruker eller skifte til en annen back-end API.

Dersom brukeren er logget inn vil navigasjonsbaren vise fram brukernavnet istedenfor login og sign up menyvalgene. Brukeren kan klikke på brukernavnet i menyen for en oversikt over sin egen profil.

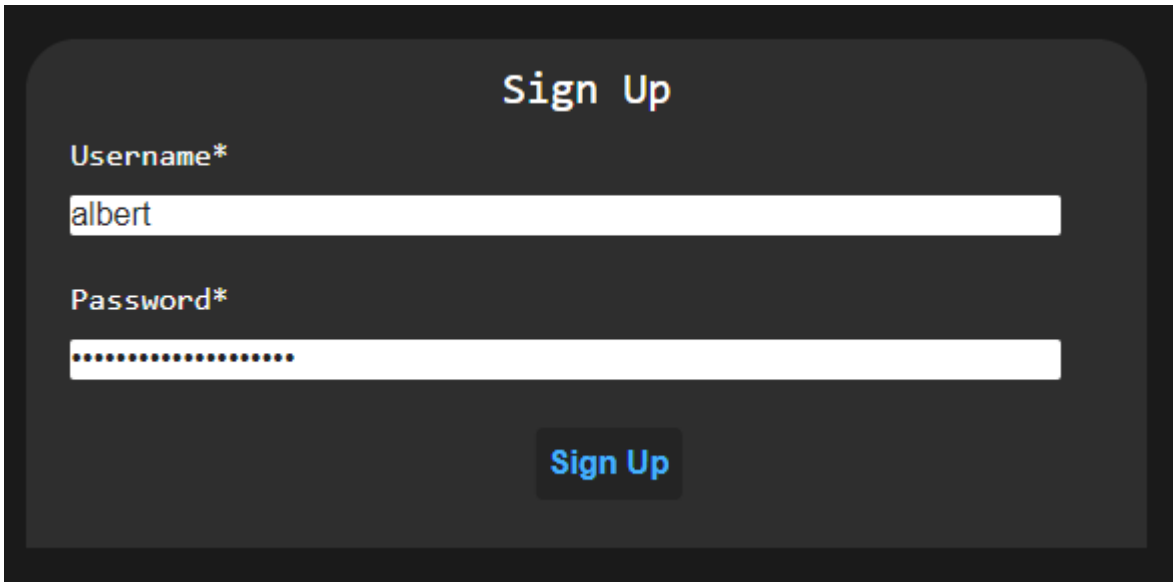
Nedenfor navigasjonsbaren ligger det på venstre side en liste med poster som tilhører forskjellige resuber. Posten kan lastes inn ved å klikke på den, og man kan også klikke på navnet til brukeren som laget posten for å se deres profil. Til høyre for tittelen på disse artiklene kan man se hvor mange stemmer posten har fått.

På høyre side av forsida kan man se den høyest vurderte posten sortert etter hvor mange stemmer den har fått. Denne posten kan også klikkes på for å laste den inn, og man kan navigere videre inn til brukeren som laget posten eller resub som den ble lagt ut i.

Figur 65. Nettside forside

4.4.2 Lage bruker

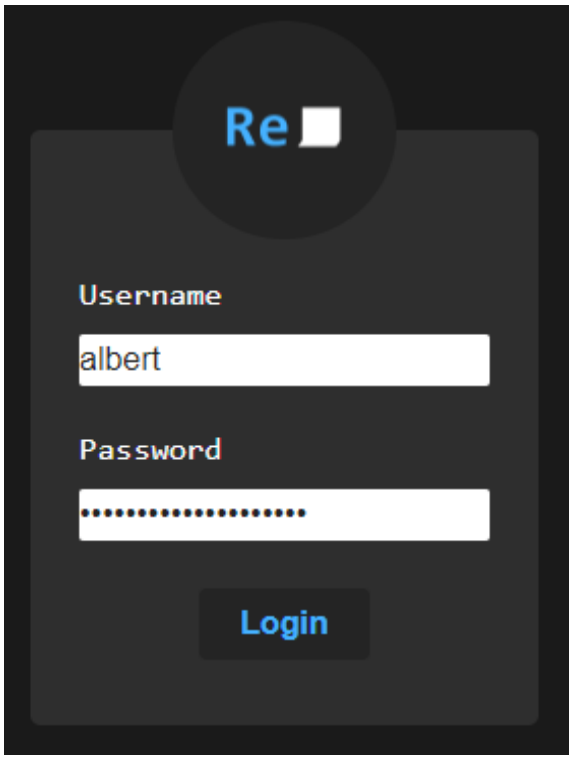
Her er det to felt brukeren må fylle ut for å lage bruker. Er begge feltene fylt inn, kan brukeren klikke på *Sign Up* for å lage brukeren. Brukeren vil deretter bli omdirigert til logg inn siden.



Figur 66. Registreringsskjema

4.4.3 Logge inn siden

I denne siden kan det fylles ut brukernavn og passord for å logge inn. Klikk på *Login* knappen etter å ha fylt ut feltene for å logge inn. Logoen kan også klikkes og sender brukeren til forsiden. På denne siden finnes det ikke en navigasjonsbar.

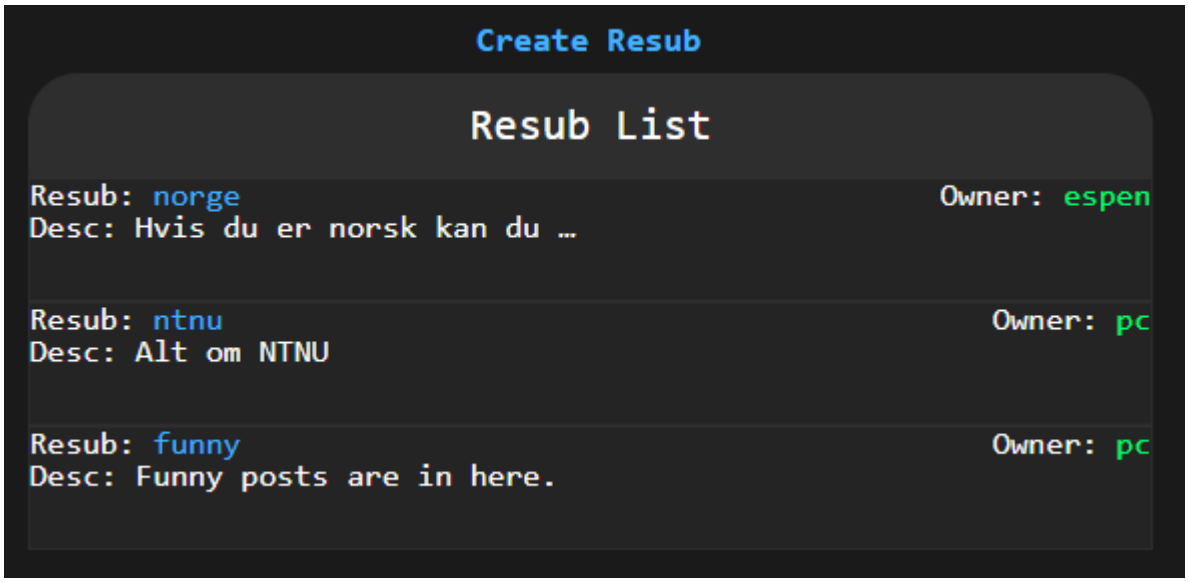


Figur 67. Skjema for å logge inn

4.4.4 Resub Liste

Her blir det vist en liste med resuber. Brukeren kan se tittelen, hvem som eier den og en del av beskrivelsen. Brukeren kan klikke seg inn på resuben for å gå inn på resubens side. Brukeren kan også velge å klikke på eierens navn for å gå til deres profil.

Dersom brukeren er logget inn vises et valg for å lage en ny resub. Ved å klikke denne knappen blir brukeren sendt videre til skjemaet for å lage en ny resub.



Figur 68. Liste av resuber

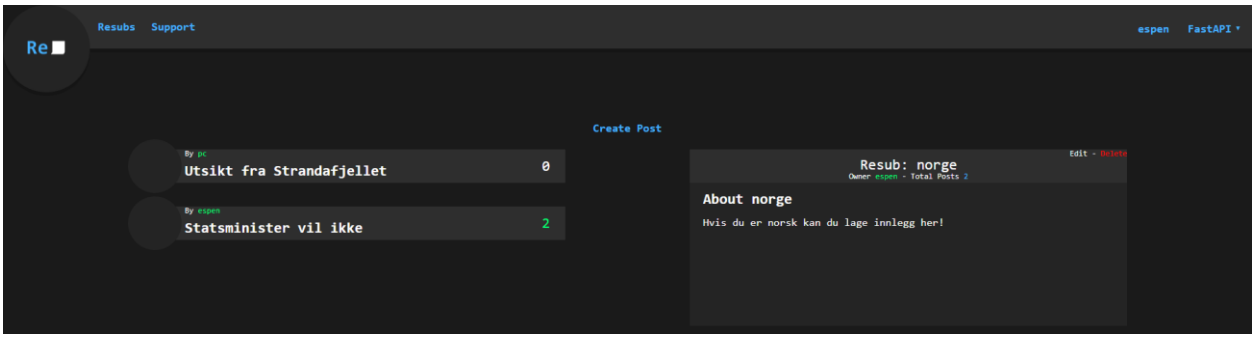
4.4.5 Resub

Til venstre kan brukeren se en liste med poster som tilhører resuben. Brukeren kan klikke på posten for å lese den, eller klikke på forfatteren for å se deres profil. Til høyre for tittelen kan brukeren se hvor mange stemmer posten har fått.

Til høyre for brukeren lese informasjon om resuben. Brukeren kan også få se hvor totalt antall poster som ligger i resuben og hvem som eier den. Brukeren kan klikke på navnet til eieren for å se deres profil.

I midten ligger det et valg som drar brukeren til et skjema der de kan lage sin egen post til resuben. Dette valget vises kun om brukeren er logget inn.

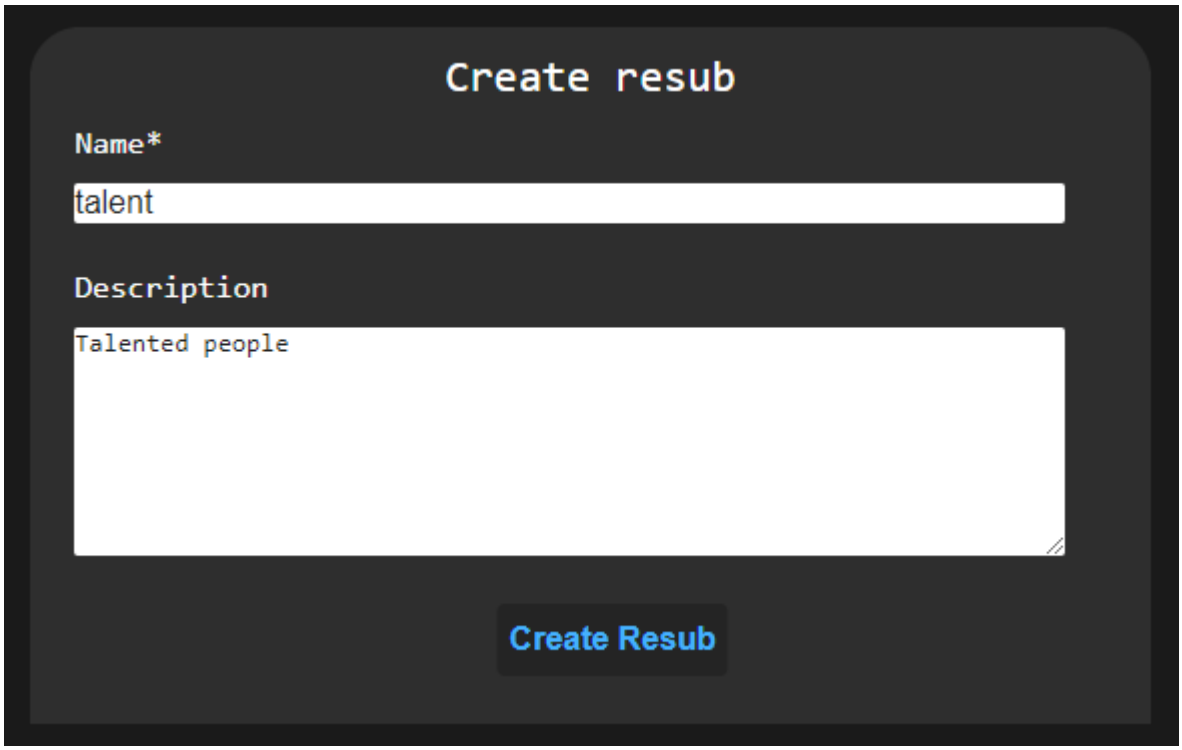
Om den innloggede brukeren er eier av resuben kan de endre på informasjonen eller eierskapet til resuben ved å klikke på *Edit* knappen i informasjonspanelet til høyre. Her kan brukeren også slette resuben ved å trykke på *Delete* knappen. Disse knappene vises kun når den innloggede brukeren er eier av resuben.



Figur 69. Visning av en resub, poster til venstre og informasjon om resuben til høyre

4.4.6 Resub Skjema

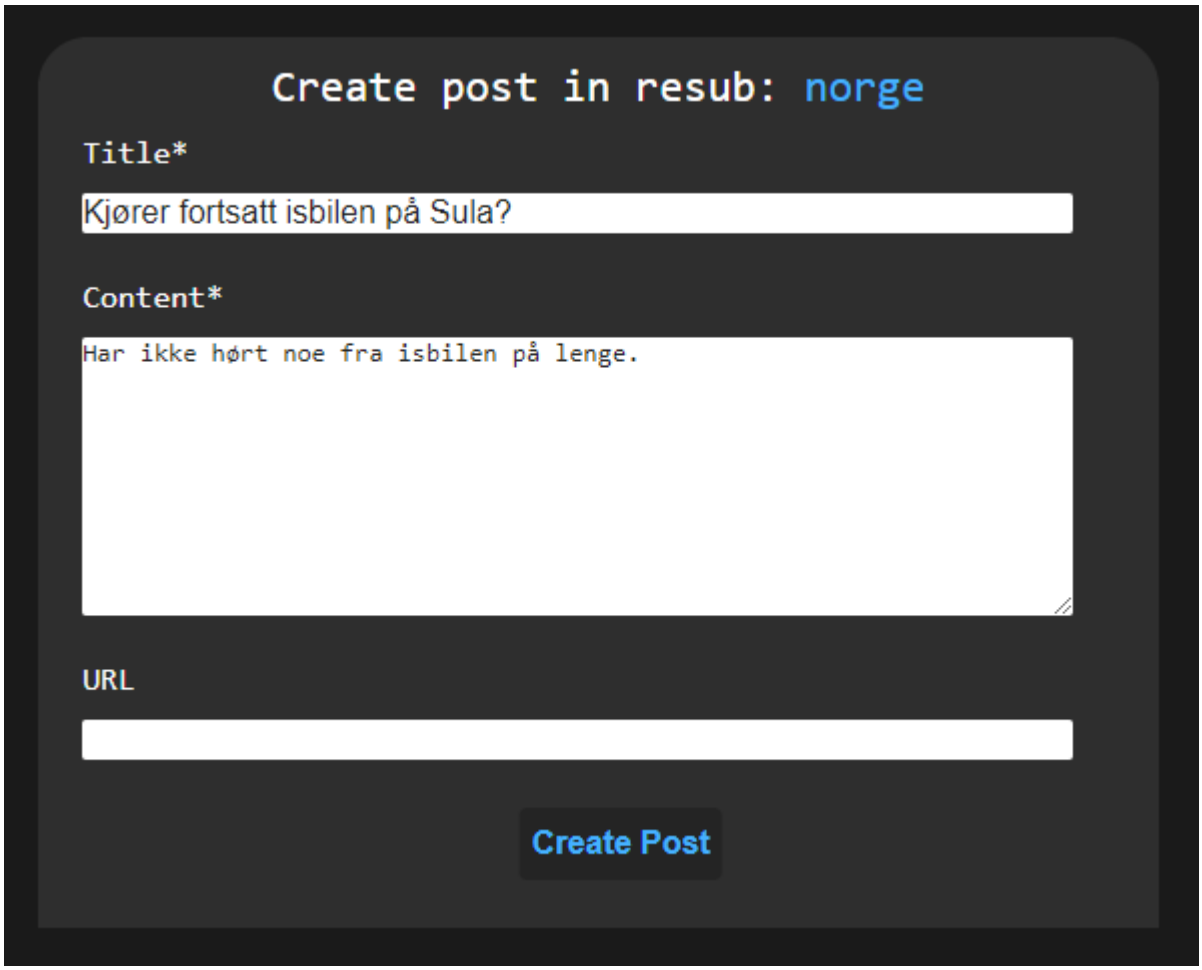
Brukeren blir vist et skjema med to felter: tittel og beskrivelse. På bunnen av skjemaet er det en knapp *Create Resub* som lager resuben om feltene har blitt fylt ut.



Figur 70. Skjema med eksempeltekst for å lage ny resub

4.4.7 Post Skjema

Brukeren blir vist en skjema med tre felter: tittel, innhold og URL. URL blir brukt for å vise et bilde skulle brukeren ønske det. Hvis brukeren lar det være tomt så blir det ikke vist noe bilde. Etter skjema er ferdig fylt ut kan brukeren klikke på *Create Post* for å lage posten. Navnet på resuben som man postet til blir lagt i tittelen på skjemaet.

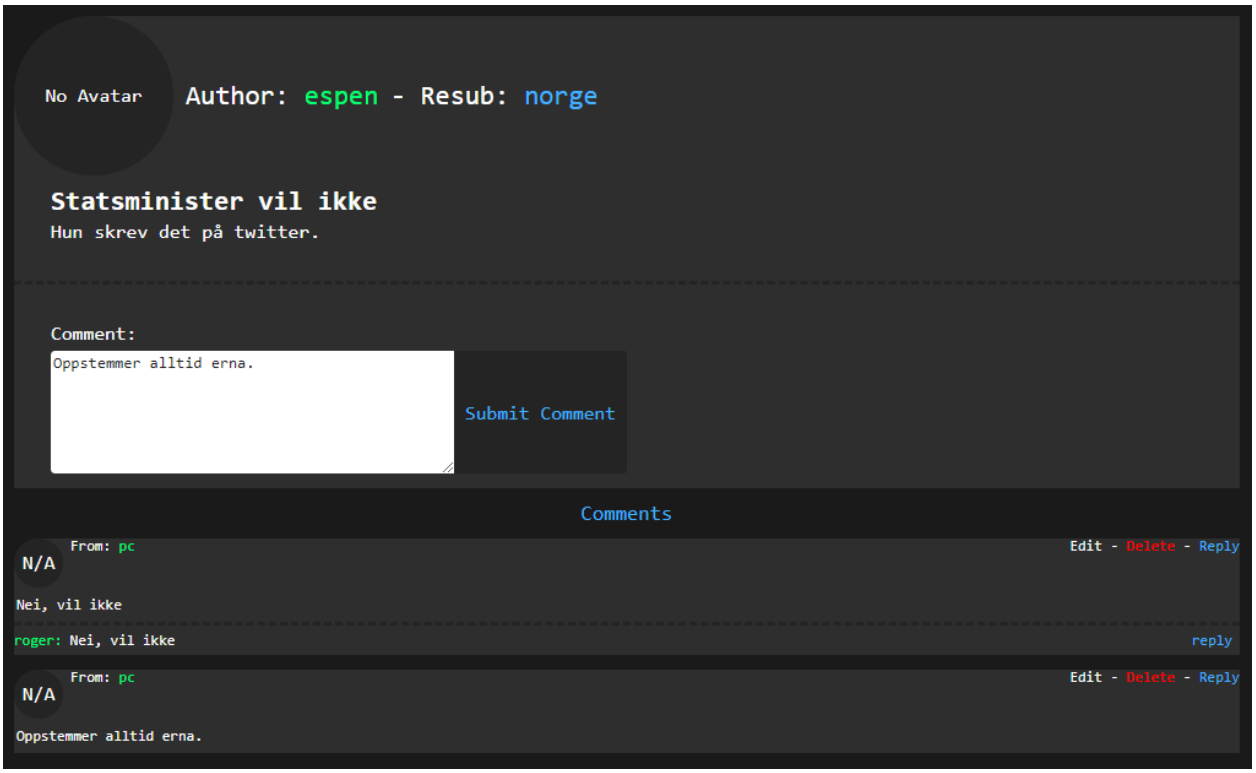


Figur 71. Skjema med eksempeltekst for å lage ny post

4.4.8 Post

På toppen av posten kan brukeren se hvem som har laget posten og hvilken resub posten ligger i. Brukeren kan klikke på disse lenkene for å dra til profilen til eieren eller resuben. Til venstre vil avataren til forfatteren bli vist hvis de har en. Hvis ikke vil det bare stå *No avatar*. Knappen *Delete* brukes til å slette posten og vises kun dersom den innloggede brukeren er eier av posten eller resuben som posten ligger i. *Edit* knappen vises kun for brukeren som laget posten.

Under profilbildet vises tittelen til posten, etterfulgt av innholdet. Om posten har en URL vil bildet fra URLen vises nedenfor. Under innholdet kan man legge til en kommentar ved å skrive inn i feltet og deretter klikke på *Submit Comment*.



Figur 72. Visning av en post med kommentarer

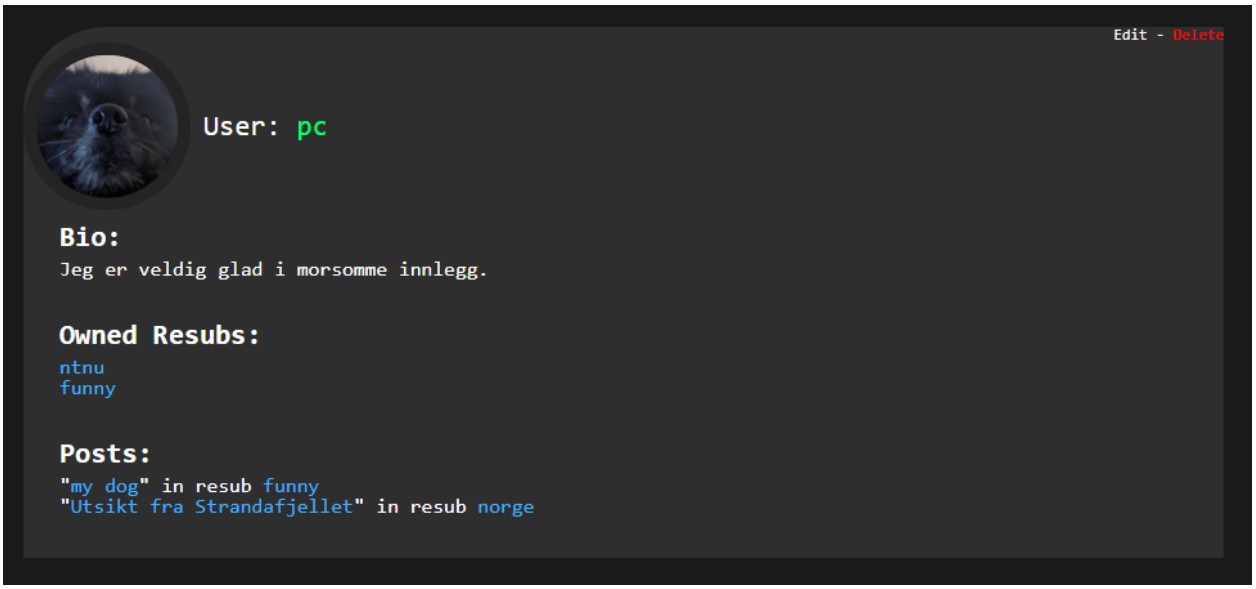
Under posten kan brukeren se kommentarene. Man kan svare på en kommentar ved å klikke på reply. Brukeren må være logget inn for å kunne gjøre dette. Dersom den innloggede bruker skrev kommentaren, eller de er eier av resuben som posten ligger i, vises en knapp *Delete* som lar brukeren slette kommentaren. *Edit* knappen vises kun dersom brukeren er eier av kommentaren, Når en bruker klikker på *Edit* vil det dukke opp et tekstfelt der brukeren kan endre kommentaren. Når brukeren er ferdig klikker de på *Edit Comment* knappen. Brukeren kan også avbryte endringen ved å klikke på “cancel” som skulle ligge til høyre for kommentaren.

Figur 73. Skjema for endring av kommentar

4.4.9 Profil

Denne siden viser all informasjon av en bruker. På toppen vises profilbildet og brukernavnet, etterfulgt av brukerens bio. Her vises også alle resubene brukeren eier, og alle poster de har lagt ut på diverse resuber. Brukeren kan klikke på de blå lenkene for å dra til resuben eller postene.

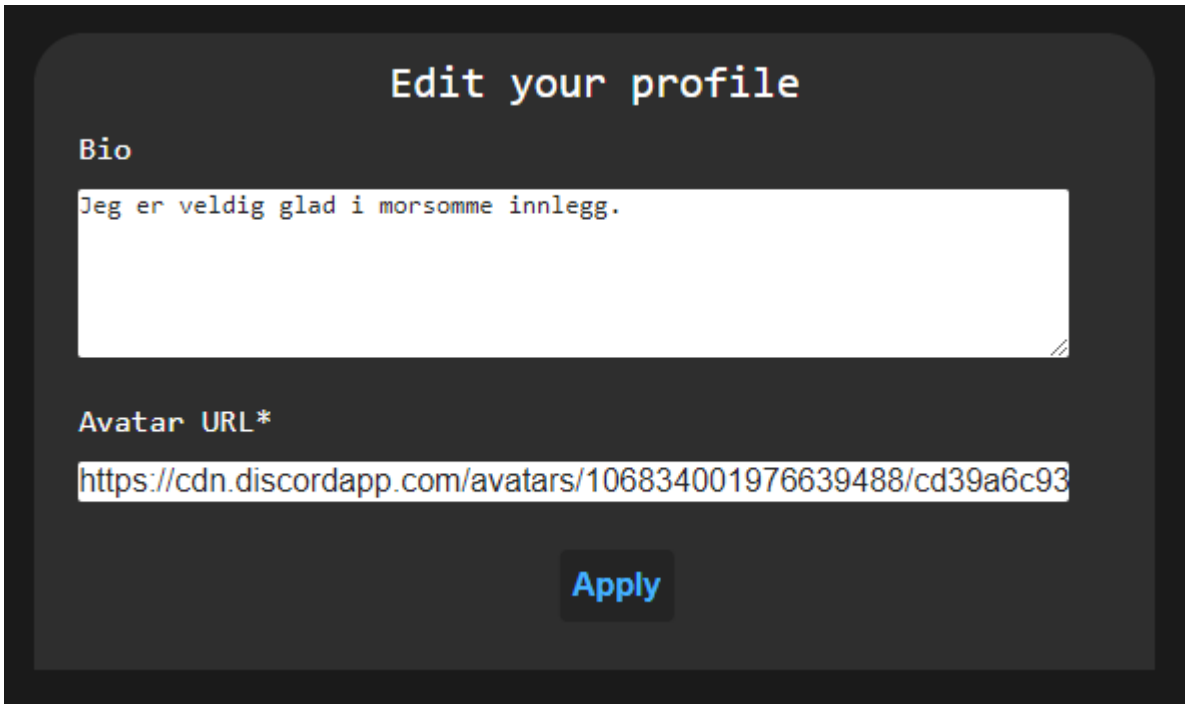
Dersom profilen som er vist er av den innloggede brukeren vises knappene *Edit* og *Delete* øverst til høyre. Ved å klikke *Edit* blir personen dirigert til et skjema der brukeren kan endre bio og avatar. Brukeren kan også slettes fra nettsiden ved å klikke på *Delete* knappen.



Figur 74. Profilsiden til en bruker

4.4.10 Profil Skjema

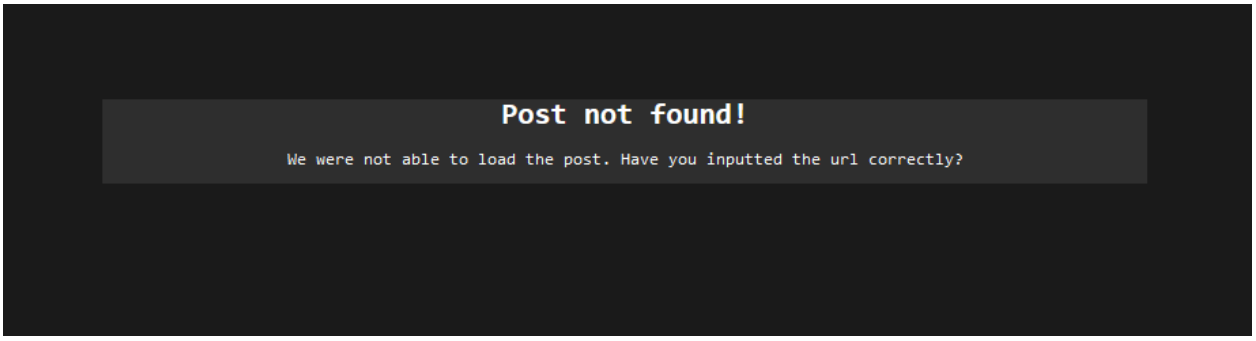
I dette skjemaet blir brukeren vist to felt: bio og avatar URL. Her kan altså brukeren endre sin bio og sette avatar ved å lime inn en lenke til et bilde på en ekstern tjeneste. Ved å klikke *Apply* blir endringene lagret og brukeren blir returnert til sin profil.



Figur 75. Skjema for å endre sin egen profil

4.4.11 Ugyldig post eller resub

Taster brukeren inn feil URL til en post eller resub blir brukeren møtt med en feilmelding. Brukeren kan da bruke navigasjonsbaren for å komme seg tilbake.



Figur 76. Feilmelding når en post ikke er funnet

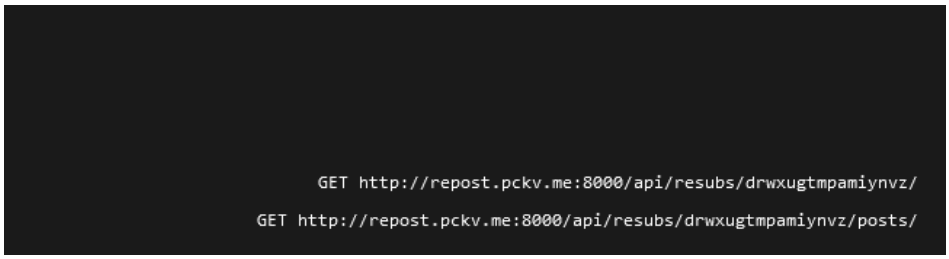
4.4.12 404 (Ikke funnet)

Taster brukeren inn en URL som ikke eksisterer, blir brukeren møtt med en feilmelding som sier at den siden brukeren prøvde å besøke ikke eksisterer. Brukeren kan klikke på *Home* lenken for å navigere tilbake til hovedsiden.

Figur 77. Feilmelding når en ugyldig URL er brukt

4.4.13 Debugging

Ved nedre høyre hjørne viser nettsiden hvilken forespørsler den gjør til backend API serveren. Dette blir da brukt for å se at kommunikasjonen fungerer mellom vue og backend, og for å demonstrere at nettsiden fungere i alle ulike implementasjoene av API-et.

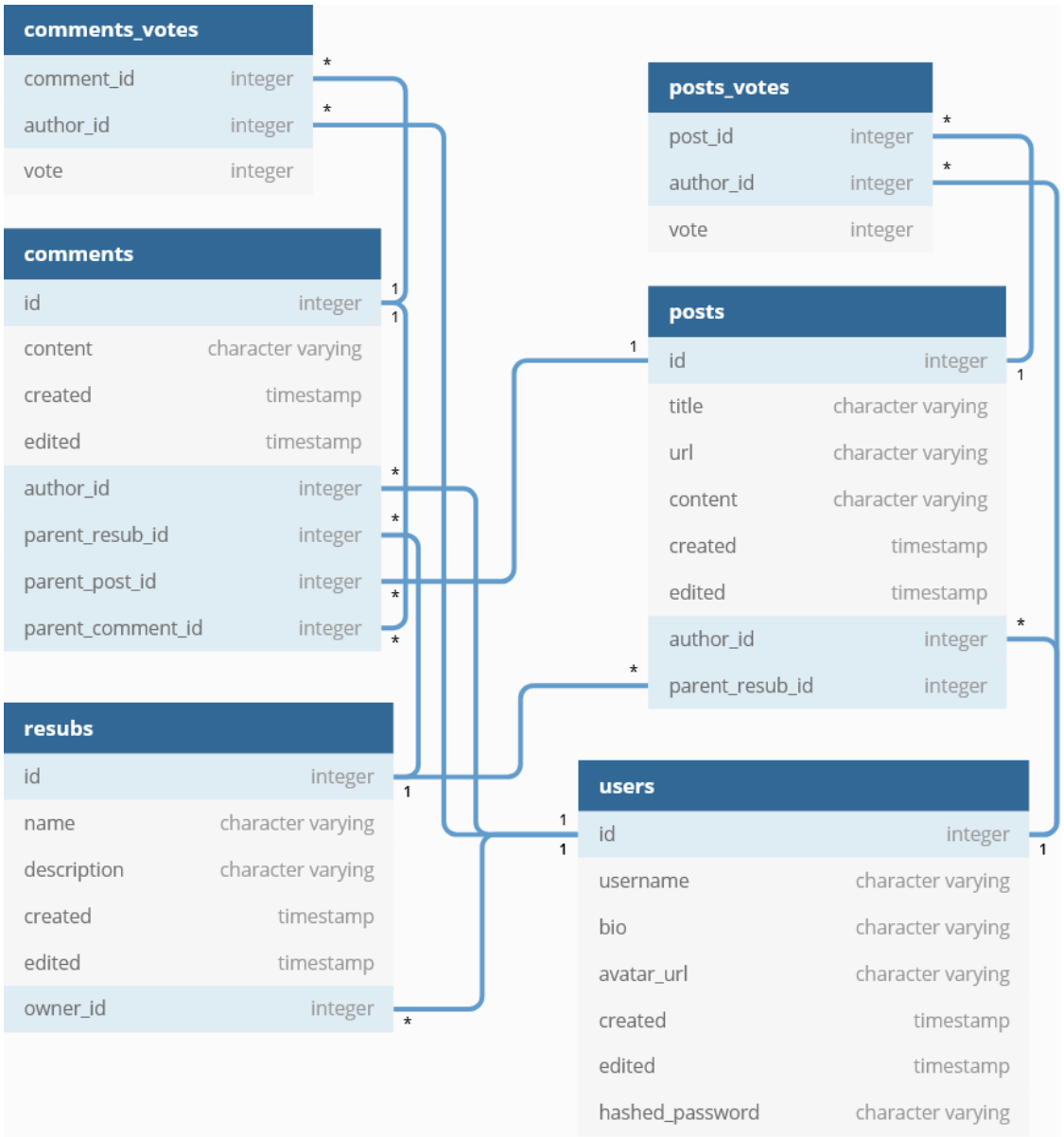


Figur 78. Visning av hvilke forespørsler som blir sendt fra front-end til API-et

4.5 Design av database

Databasen i vårt prosjekt er et resultat av ORM i de forskjellige rammeverkene. Derfor blir implementasjonen litt forskjellig ettersom hvordan ORM prosessen utføres. Strukturen i dette kapittelet tar utgangspunkt i resultatet av ORM i FastAPI prosjektet.

Databasestrukturen endte opp med å være veldig enkel. Primært har vi 4 tabeller for de 4 hovedressursene: *users*, *resubs*, *posts* og *comments.* I tillegg knytter vi opp egne tabeller for votes: *posts_votes*, *comments_votes*.



Figur 79. Databasedesign med relasjoner

Alle tabeller av primærressursene har en *id* nøkkel som er et unikt tall. Dette er primærnøkkelen som vi bruker i databasen til å lage relasjoner mellom tabellene. I API-et brukes ikke *id* nøkkelen til *users* og *resubs* ettersom vi også har unike navn der. For API-ets skyld gir det da god mening å referere til navnet framfor *id* nøkkelen, som også er mer intuitivt for brukeren både av både nettsiden og av API-et.

posts_votes	
post_id	integer
author_id	integer
vote	integer

Tabellene *comments_votes* og *users_votes* bruker ikke *id* nøkler, men er sammensatt av en komposittnøkkel av brukeren som eier stemmen og ressursens *id*. Vi laget det slik fordi stemmer må være tett knyttet mellom brukeren og ressursen, og dermed trenger ikke denne informasjonen å ha sin egen identitet. Siden hver bruker kun kan stemme én gang på en ressurs trenger dette heller ikke å være en many-to-many relasjon.

Hierarkiet starter i *users* tabellen som ikke har noen fremmednøkler. *resubs* tabellen referer til en *owner_id* fra *users* tabellen. Altså har *resubs* et many-to-one forhold med *users* tabellen. Slik er oppsettet for resten av tabellene også, hvor en ressurs alltid kun har behov for én fremmednøkkel og dermed er alle relasjonene many-to-one der det er fremmednøkler.

Alle ressursene utenom *users* har en eier, så disse tabellene har *owner_id* eller *author_id* basert på hvilket navn som passer best til typen ressurs. For noen tilfeller som *comments* har vi inkludert både *parent_resub_id* og *parent_post_id*. En kommentar er knyttet til en post og en post er knyttet til en resub, og dermed er ikke *parent_resub_id* egentlig nødvendig. Vi valgte å modellere det slik ettersom vi også ønsket å ha med denne verdien i API-et. På denne måten kan vi sende med dette uten å måtte gjøre en ekstra query i databasen.

comments	
id	integer
content	character varying
created	timestamp
edited	timestamp
author_id	integer
parent_resub_id	integer
parent_post_id	integer
parent_comment_id	integer

Figur 80. Databasetabell som viser alle felt for kommentarer

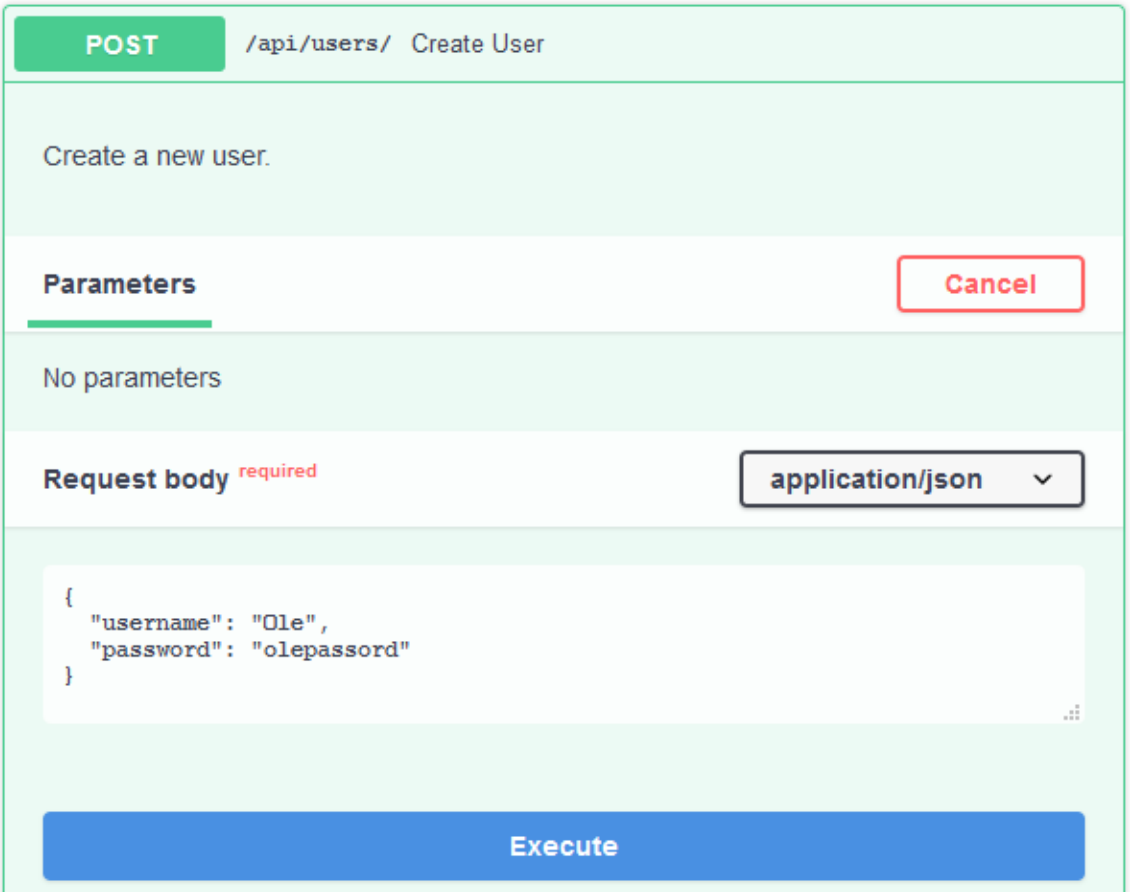
4.6 Bruk av API

API-ene kommuniseres med ved HTTP forespørsler. Som bruker kan vi teste alle endepunktene definert i API-ene med Swagger UI, som ligger på */api/swagger* i API-et. Her vises også *curl* kommandoer for hver forespørsel som blir gjort, og de kan bli kopiert og limt inn i en terminal.

Swagger UI er altså dokumentasjonen til API-et, så videre bruk av API er definert der. Nedenfor viser vi likevel hvordan man gjør forespørsler med noen av de forskjellige metodene i API-ene. Datamodeller og parametere er beskrevet i detalj i Swagger UI og dermed også OpenAPI dokumentet som finnes i Vedlegg 9 (Vedlegg 9).

4.6.1 Opprette ressurser

For å lage en ny ressurs i API-et må et POST endepunkt brukes, med unntak av login endepunktet som også bruker POST. I disse endepunktene sender vi data om ressursen som skal lages i *body* delen av forespørselen, hvor vi bruker *application/json* som mime-type for å sende data som JSON.



Figur 81. Oppretting av ressurs med Swagger UI

I Swagger UI trykker vi Execute for å utføre forespørselen, som vil gi oss en respons fra serveren. Det er her Swagger UI viser *curl* kommandoen, URL til endepunktet som ble brukt i forespørselen, statuskoden på respons, JSON representasjon av ressursen som ble opprettet og headers returnert av serveren. I dette tilfellet laget vi en bruker med brukernavn *Ole*.

Responses

Curl

```
curl -X POST "http://repost.pckv.me:8000/api/users/" -H "accept:
application/json" -H "Content-Type: application/json" -d "{\"username
\":\"Ole\",\"password\":\"olepassord\"}"
```

Request URL

```
http://repost.pckv.me:8000/api/users/
```

Server response

Code	Details
201	Response body

```
{
  "username": "Ole",
  "bio": null,
  "avatar_url": null,
  "created": "2020-05-09T11:35:33.782557+00:00",
  "edited": null
}
```

Download

Response headers

```
access-control-allow-credentials: true
content-length: 106
content-type: application/json
date: Sat09 May 2020 11:35:33 GMT
server: uvicorn
```

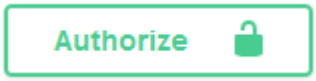
Figur 82. Respons på opprettelse av ressurs med Swagger UI

4.6.2 Innlogging

For å logge inn må vi spørre API-et om en OAuth2 token. I skjemaet for token må vi spesifisere OAuth2 grant type (altså OAuth2 flow), brukernavn, passord, OAuth2 klient ID og OAuth2 scope. JSON skjemaet vil da se slik ut:

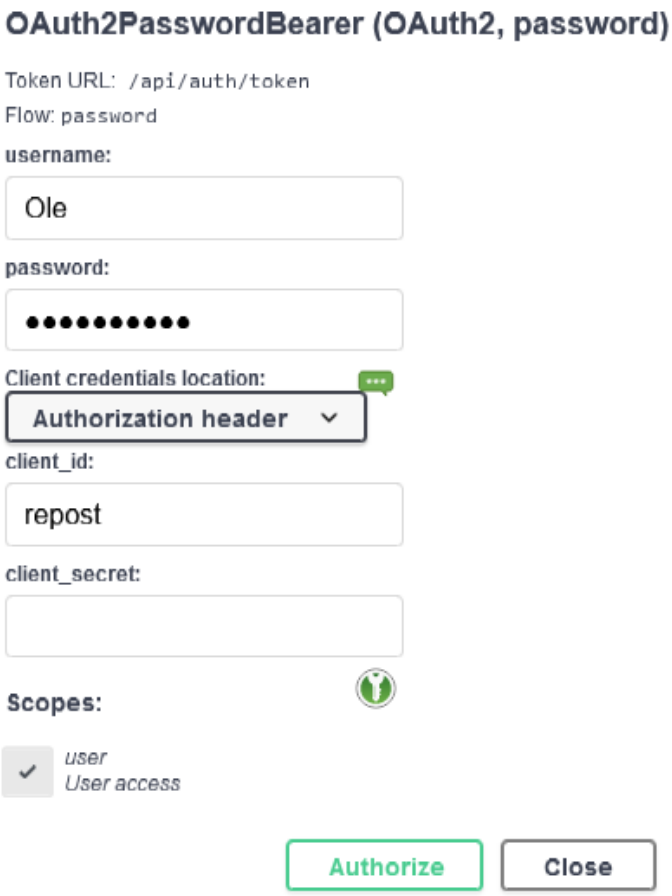
```
{
    "grant_type": "password",
    "username": "BRUKERNAVN",
    "password": "PASSORD",
    "client_id": "repost",
    "scope": "user"
}
```

For testing av API-et i Swagger UI kan vi bruke *Authorize* knappen øverst på siden.



Figur 83. Autoriseringsknapp i Swagger UI

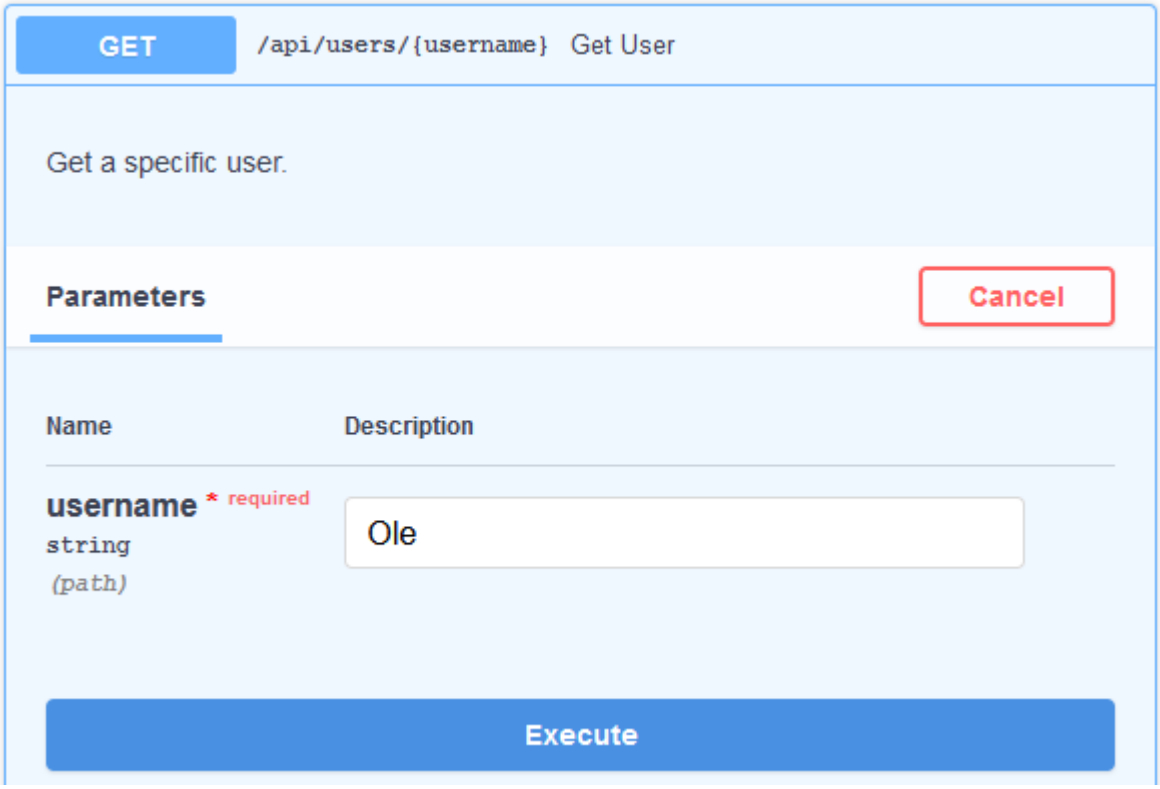
Da får vi en dialog for skjemaet beskrevet ovenfor. Her kan vi fylle ut brukernavn, passord, OAuth2 klient og velge OAuth2 scopes. Ved å fortsette i dialogen vil Swagger UI lagre OAuth2 token og bruke den videre i endepunkt som krever autorisering.



Figur 84. Autoriseringsskjema i Swagger UI

4.6.3 Spørre om ressurser

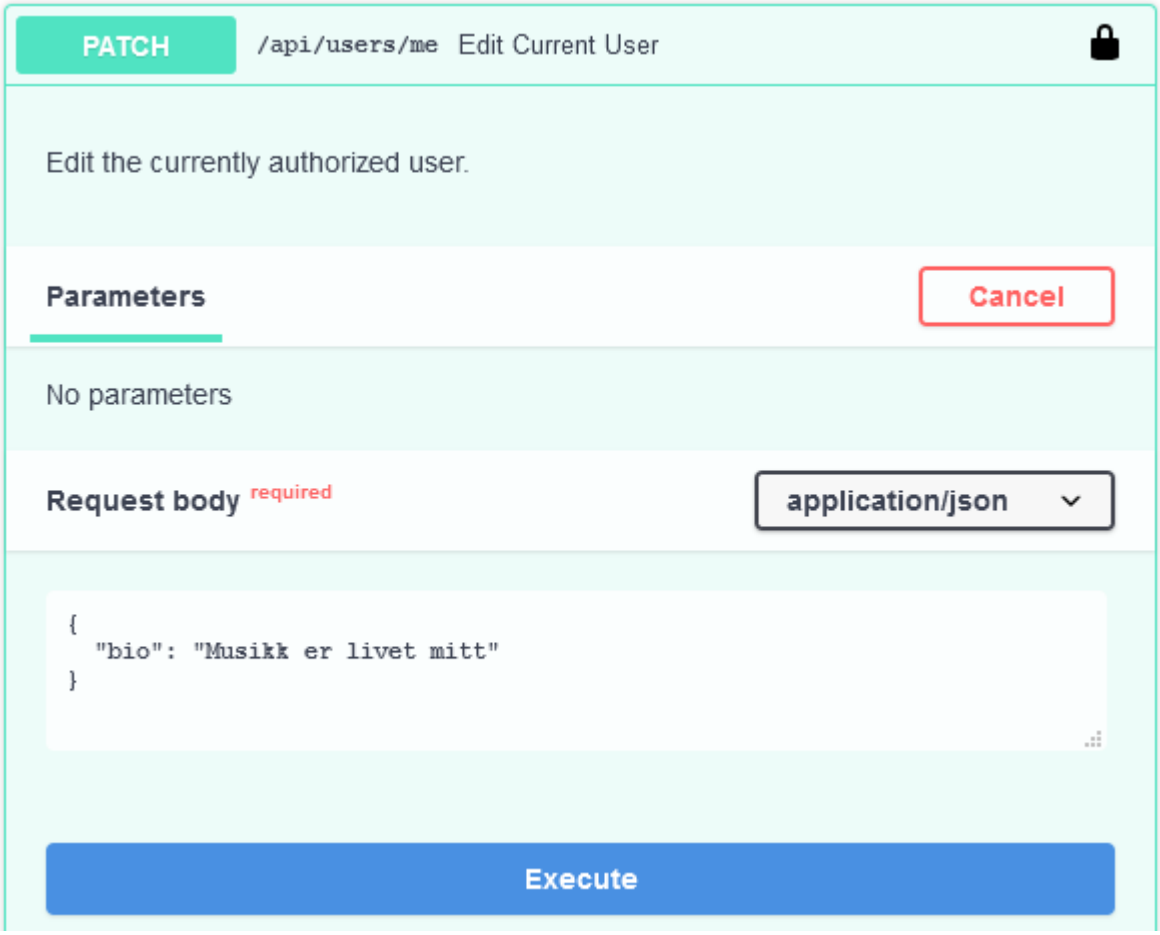
For å sende en forespørsel om en eller flere ressurser bruker vi endepunkt med GET som metode. Alle endepunkt til ressurser vil i flere nivåer kreve et parameter i stien. Figur 85 viser et eksempel på henting av bruker ressursen vi laget tidligere med parameter i sti navnet. Navnene til disse stiene følger en intuitiv måte å vise til ressurser og underressurser, så for å få brukeren med navn Ole skriver man */users/Ole*. I Swagger UI ser vi dette fra navnet på endepunktet med sti */users/{username}*, hvor *{username}* erstattes med brukernavnet til brukeren.



Figur 85. Hente ressurs med Swagger UI

4.6.4 Endre ressurser

Forespørsler om å endre en ressurs sendes med PATCH metoden. Alle ressurser som kan endres krever at brukeren er logget inn og har rettigheter til å endre ressursen. Swagger UI dokumenterer dette med hengelåsikonet til høyre for stien til endepunktet. I disse endepunktene må skjema fylles inn som JSON på samme måte som i endepunktene for å opprette ressurser.



Figur 86. Endre ressurs med Swagger UI

4.6.5 Slette ressurser

Ressurser kan slettes i endepunkter som bruker DELETE metode. Disse tar ingen parametere i APIen, men det kreves at den autoriserte brukeren er eier av ressursen for å slette den.

Figur 87. Slette ressurs med Swagger UI

4.7 Struktur og inndeling av funksjonalitet i API

4.7.1 FastAPI implementasjon

Vi har en mappestruktur som et vanlig Python prosjekt. Prosjektfiler, altså filer uten kode, ligger i rotmappen. Alle kode filene ligger under mappen repost og videre fordelt i undermapper der det er naturlig. Main fila skal kjøres med en ASGI server, i vårt tilfelle Uvicorn, som vil alltid sette opp miljøet slik at pakkene blir funnet.

- repost - pakke for all koden for projektet
    - api - pakke for koden som har med APIen
        - routes - pakke for alle rutere for endepunktene
        - schemas - pakke for pydantic modeller
        - api.py - fil for å samle alle rutere til en hovedruter
        - resolvers.py - samling av funksjoner som blir brukt som dependencies
        - security.py - funksjoner for JWT og OAuth2 oppsett
    - crud - pakke for CRUD metodene som gjør all interaksjon med databasen
    - models - pakke for SQLAlchemy modeller
    - config.py - fil for konfigurasjon av JWT og database
    - database.py - fil for oppkobling av database
    - main.py - filen der FastAPI appen blir instansiert og kjørt
    - password.py - fil for kryptering og validering av passord
- .gitignore - samling av filer som ikke skal versjoners
- config.env - konfigurasjon for JWT og database
- LICENSE - lisensfil
- repost.db - databasefil for lokal testing under utvikling
- requirements.txt - nødvendige avhengigheter for at prosjektet skal fungere

4.7.1.1 SQLAlchemy Modeller

I FastAPI bruker vi SQLAlchemy modeller for de objektene som skal lagres i databasen. Disse modellene har alle feltene for et objekt, og her lager vi også relasjonene mellom forskjellige modeller. Modellene holder også på felt som ikke er tilgjengelige gjennom APIen, for eksempel feltet *hashed_password* er lagret i databasen. Det brukes kun internt for å validere passordet når noen spør om å få en autentiserings token og kan ikke bli hentet ut via noen API forespørsler.

```
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True, index=True)
    username = Column(String, unique=True, index=True)
    bio = Column(String)
    avatar_url = Column(String, nullable=True)
    created = Column(DateTime(timezone=True), server_default=func.now())
    edited = Column(DateTime(timezone=True), onupdate=func.now())

    hashed_password = Column(String)

    resubs = relationship('Resub', back_populates='owner')
    posts = relationship('Post', back_populates='author')
    comments = relationship('Comment', back_populates='author')
```

Figur 88. SQLAlchemy modell for en bruker

4.7.1.2 Pydantic modeller

De modellene som blir eksponert gjennom API-et er Pydantic modeller som vi kaller skjema. Disse skjemaene har ikke alle feltene slik som SQLAlchemy modellene har, de har kun de feltene som er relevante for operasjonene der de skal brukes.

```
class User(BaseModel):
    """Schema for a user account"""
    username: str
    bio: Optional[str]
    avatar_url: Optional[str]
    created: datetime
    edited: Optional[datetime]

    class Config:
        orm_mode = True


class CreateUser(BaseModel):
    """Schema for creating a new user account"""
    username: str
    password: str


class EditUser(BaseModel):
    """Schema for editing a user account"""
    bio: Optional[str] = None
    avatar_url: Optional[str] = None
```

Her kan vi se at for en bruker har vi tre skjema. *User* er skjema som representerer en bruker i API-et og den har de fleste feltene som SqlAlchemy modellen har. Vi ser også at vi har en konfigurasjon *orm_mode = True* som gjør at skjemaet blir koblet sammen med modellen ved hjelp av ORM.

Videre har vi *CreateUser* og *EditUser*. *CreateUser* brukes når vi skal lage en ny bruker. Den eneste informasjonen vi trenger da er brukernavn og passord. Annen informasjon kan bli lagt til senere ved å bruke *EditUser* i PATCH endepunktet.

4.7.1.3 SQLAlchemy ORM og CRUD

Vi bruker SQLAlchemy ORM for å lage databasetabeller og gjøre operasjoner på databasen. Tabellene blir automatisk generert basert på SQLAlchemy modellene. SQLAlchemy gjør det

veldig enkelt å gjøre forespørsler til databasen slik at vi ikke trenger å skrive SQL queries manuelt. Her gjør vi en spørring i databasen etter modellen User og filtrerer etter brukernavn.

```
def get_user(db: Session, *, username: str) -> User:
    """Get the user with the given username."""
    return db.query(User).filter_by(username=username).first()
```

Figur 89. SQLAlchemy databasespørring i en CRUD metode

Disse operasjonene bruker vi til å lage CRUD pakken. Her har vi en samling av funksjoner som snakker med databasen og kan gjenbrukes flere steder i koden.

4.7.1.4 Ruter

Den delen som klienten snakker med er en *APIRouter*. *APIRouter* er en del av FastAPI og instansieres med *router = APIRouter()*.

```
@router.get('/me', response_model=User,
            responses={status.HTTP_400_BAD_REQUEST: {'model': ErrorResponse},
                       status.HTTP_401_UNAUTHORIZED: {'model': ErrorResponse}})
async def get_current_user(current_user: models.User = Depends(resolve_current_user)):
    """Get the currently authorized user."""
    return current_user
```

Figur 90. Endepunkt i kontrolleren for brukere

For å lage endepunkter på denne ruteren bruker vi annoteringer på en funksjon. I annoteringen bestemmer vi hvilken HTTP metode endepunktet lytter etter, og så må vi skrive inn hvilken sti den lytter på. Responsmodell er en Pydantic modell, eller skjema, som er det som skal returneres ved suksess. I tillegg må vi legge inn alternative responser som kan oppstå ved feil. Disse responsene bruker sin egen modell *ErrorResponse*.

I funksjonen vi har annotert henter vi inn nødvendig data fra forespørselen. I eksempelet bruker vi et GET endepunkt, og dette har ingen data inn fra brukeren. Likevel har vi et parameter i funksjonen. Dette parameteret bruker en dependency for å finne brukerobjektet som tilhører brukeren som gjorde forespørselen basert på brukernavnet som er assosiert med autentisering token som ble brukt.

Selve funksjonaliteten som brukes i endepunktene vil vi abstrahere så mye som mulig slik at den blir implementert som gjenbrukbare funksjoner utenom selve ruteren. Vi bruker dermed dependencies og crud funksjoner så mye det lar seg gjøre.

4.7.1.5 Dependencies - hjelpefunksjoner i FastAPI

FastAPI har et system som heter Dependency Injection. Dette systemet gjør at vi kan bruke andre funksjoner som en dependency i et endepunkt i ruteren. Injeksjon i denne sammenhengen betyr at du i endepunkt funksjonen sier at du trenger et objekt for å kunne fortsette, og FastAPI

vil da bruke dependency funksjonen for å skaffe det objektet som trengs (*Dependencies - First Steps - FastAPI*, no date).

Grunnen til å bruke dette systemet er at vi kan lage funksjoner utenom ruteren eller endepunktet som kan gjenbrukes av flere endepunkt. Da unngår vi å skrive den samme logikken i flere endepunkter og dupliserer kode.

I eksempelet bruker vi igjen GET endepunkt for */me*. Her trenger vi objektet *current_user* som er Pydantic *User* skjemaet. Når vi setter *parameter = Depends(resolve_current_user)* så betyr det at FastAPI vil kjøre *resolve_current_user* funksjonen og returnere objekter før vi kommer inn i endepunkt funksjonen.

```
@router.get('/me', response_model=User,
            responses={status.HTTP_400_BAD_REQUEST: {'model': ErrorResponse},
                       status.HTTP_401_UNAUTHORIZED: {'model': ErrorResponse}})
async def get_current_user(current_user: models.User = Depends(resolve_current_user)):
    """Get the currently authorized user."""
    return current_user
```

Figur 91. Bruk av resolve_current_user Dependency i endepunkt i kontrolleren for brukere

```
async def resolve_current_user(username: str = Security(authorize_user, scopes=['user']),
                               db: Session = Depends(get_db)) -> models.User:
    """Resolve the currently authorized User."""
    db_user = crud.get_user(db, username=username)
    if not db_user:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
                            detail=f'The owner of this JSON Web Token no longer exists')

    return db_user
```

Figur 92. Dependency resolve_current_user som returnerer brukeren som tilhører token i forespørselen

Funksjonene som blir brukt som dependencies bruker CRUD funksjonene internt. Det er igjen for å unngå unødvendig duplisering av kode, slik at feilmeldinger som at ressursen ikke eksisterer kan defineres på ett sted.

4.7.1.6 Database

Under utvikling brukte vi en SQLite (*SQLite Home Page*, no date) database lagret som *repost.db*. Under testing med JMeter bruker vi PostgreSQL.

```
url = make_url(config.database_url)
connect_args = {}

# SQLite driver only allows one thread by default to prevent multiple
# connections, but internally we are opening multiple connections so
# multiple threads can be used
if url.drivername == 'sqlite':
    connect_args['check_same_thread'] = False

engine = create_engine(url, connect_args=connect_args)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()
```

Figur 93. Konfigurasjon av databasetilkobling med SQLAlchemy

4.7.1.7 Konfigurasjon

I repost pakken ligger det en fil som heter *config.py*. Her bruker vi *dotenv* biblioteket for å laste inn miljøvariabler fra filen config.env som ligger i rotmappen. Om det ikke eksisterer en slik fil blir det generert en fil automatisk og den blir brukt i stedet.

Verdiene som settes gjennom denne filen er JWT secret og algoritme i tillegg til en koblings URL til databasen.

4.7.1.8 Sikkerhet

I *security.py* som ligger i pakken *repost.api* har vi funksjoner for å håndtere JWT som vi bruker for autentisering. Her finner vi funksjonen *create_jwt_token* som lager en ny token for en bestemt bruker og funksjonen *get_jwt_token_username* som henter brukernavn fra en eksisterende token. Den siste funksjonen *authorize_user* bruker *get_jwt_token_username* for å finne brukernavnet som er i payload i tokenen.

4.7.1.9 Passord

For å hashe og validere passord brukte vi BCrypt algoritmen i passlib. Disse funksjonene brukes kun for å hashe et passord for en ny bruker eller for å verifisere passordet for en bruker som prøver å autentisere seg.

```python
from passlib.context import CryptContext

password_context = CryptContext(schemes=['bcrypt'], deprecated='auto')


def hash_password(password: str) -> str:
    """Hash the given password using the context scheme."""
    return password_context.hash(password)


def verify_password(password: str, hashed_password: str) -> bool:
    """Compare the given password with the given hashed password."""
    return password_context.verify(password, hashed_password)
```

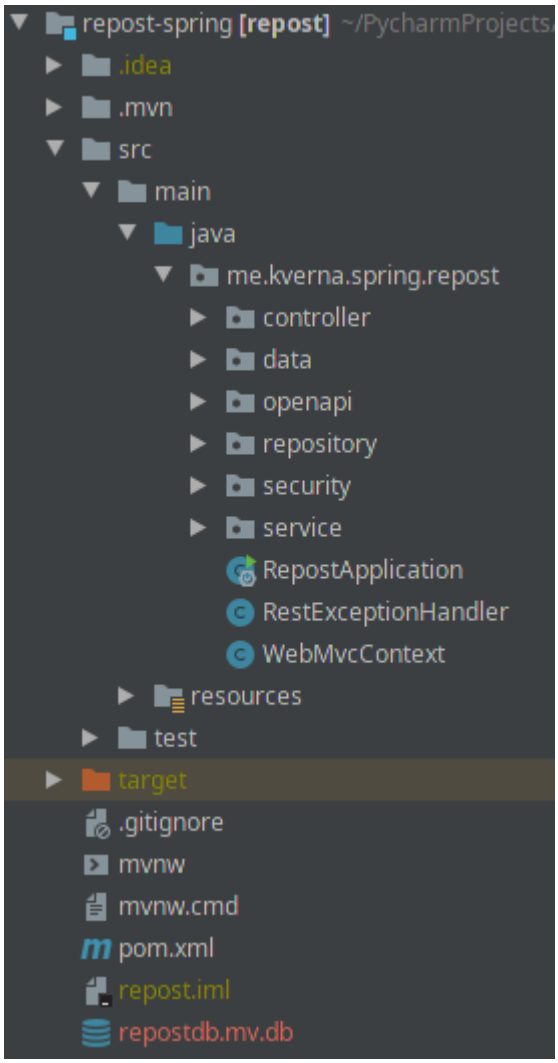
Figur 94. Funksjoner for å lage og validere et BCrypt passord

4.7.1.10 Avhengigheter

Tekstfilen *requirements.txt* inneholder navnet og versjonen på eksterne biblioteker, eller avhengigheter. Disse bibliotekene installeres gjennom verktøyet pip. Kommandoen for å installere alt som er spesifisert i filen er *pip install -r requirements.txt*. Dette sørger for at det er enkelt å sette opp et miljø som er likt vårt eget, ettersom alle versjonene er dokumentert i denne fila.

4.7.2 Spring Boot implementasjon

Prosjektet er satt opp som et typisk Spring Boot prosjekt. Vi brukte IntelliJ for å lage prosjektet, og siden det er støtte for Spring Boot ble vi guidet gjennom oppsettet på lignende måte som Spring Initializr. Her valgte vi versjon av Spring og Java, byggeverktøy som Maven og navn på prosjektet. Vi la ikke inn avhengigheter her, det gjorde vi etterhvert som vi måtte bruke de i koden. GroupID for prosjektet er *me.kverna.spring* og artifactID er *repost*, dermed ligger koden i *src.main.java.me.kverna.spring.repost*, og konfigurasjonsfil og andre ressurser ligger i *src.main.resources*.



Figur 95. Mappestruktur i Spring Boot prosjektet i JetBrains IDE

Videre i dette kapittelet beskrives i funksjonen til de diverse strukturene.

4.7.2.1 controller - API kontrollere

For hver datamodell har vi et API kontroller for å gi tilgang til dataene. For å lage et endepunkt i kontrolleren må vi bruke en annotasjon for å si hvilken HTTP metode, og hvilken sti, vi venter på. I tillegg skriver vi hvilken media type som er godtatt og hva som sendes tilbake.

```
@GetMapping(value = "/me", produces = {"application/json"})
public User getCurrentUser(@CurrentUser User currentUser) {
    return currentUser;
}
```

Figur 96. Endepunkt i kontrolleren for brukere

Vi må også bruke en *@Operation* annotasjon slik at vi kan lage OpenAPI dokumentasjon. Denne annotasjonen inneholder en kort beskrivelse, beskrivelse og alle mulige responser. Dette skal vi beskrive mer i Integrere OpenAPI dokumentasjon (4.8.3).

For å holde mesteparten av logikken på en plass har vi valgt å kun bruke service metoder i kontrolleren i stedet for å skrive logikken på nytt. Det blir dermed enkelt å endre funksjonalitet på ett sted og være sikker på at alle som bruker metoden fungerer på samme måte.

Annotasjonen *@CurrentUser* er en resolver. Dette er en metode utenfor kontrolleren som kan hente et objekt så vi slipper å gjøre det i kontrolleren. I dette tilfellet henter vi et brukerobjekt basert på brukernavn fra autentiserings token på samme måte som resolver i FastAPI.

4.7.2.2 data - Datamodeller

I Spring Boot lager vi klasser for alle ressursene vi skal bruke. Her bruker vi *@Data* annotasjon for å si at det er en dataklasse som gjør at lombok vil generere metoder som get, set og equals. Vi bruker også persistence sin *@Entity* annotering for å si at objektet skal kunne lagres i en database. Her er en klasse både modell for databasen og APIen, men vi lager de ekstra skjema modellene for API-et med kun *@Data* annotasjon da de kun skal brukes til forespørsler i APIen. Siden vi lagrer felt i databasen som ikke skal vises i APIen, slik som *hashedPassword*, bruker vi *@JsonIgnore* annotering for de feltene som skal ignoreres. Vi bruker også *@JsonProperty* annotering på noen felt der navnet i klassen bruker camelcase mens feltet i API-et skal være snake case.

```java
@Data
@Table(name = "users")
@Entity
@NoArgsConstructor
public class User {
    @Id
    @GeneratedValue
    @JsonIgnore
    private int id;
    private String username;
    private String bio;

    @JsonProperty(value = "avatar_url")
    private String avatarUrl;

    @JsonProperty(access = JsonProperty.Access.READ_ONLY)
    private LocalDateTime created;

    @JsonProperty(access = JsonProperty.Access.READ_ONLY)
    private LocalDateTime edited;

    @JsonIgnore
    private String hashedPassword;
```

Figur 97. Datamodell og skjema for en bruker i Spring Boot

```java
@Data
@NoArgsConstructor
public class CreateUser {
    private String username;
    private String password;
}
```

Figur 98. Skjema for å opprette en bruker i Spring Boot

4.7.2.3 openapi - OpenAPI konfigurasjoner

I openapi pakken ligger det beskrivelse om hvordan autentiseringen skal fungere. Her beskrives navn, type, format, flow, url og scope for OAuth2.

4.7.2.4 repository - Databaseoperasjoner

Alle modellene får en Java Persistence repository som gir oss tilgang til mange forskjellige kall til databasen. Disse kallene er er typiske *findBy*, *findAll*, *save*, *delete* og så videre. I tillegg kan vi legge til egne metoder, som å finne alle poster basert på hvilken bruker som laget kommentaren.

Disse metodene bruker vi i service klassene. Java Persistence vil da automatisk implementere disse metodene med SQL queries.

```
@Repository
public interface CommentRepository extends JpaRepository<Comment, Integer> {
    List<Comment> findAllByAuthor(User author);

    List<Comment> findAllByParentResub(Resub resub);

    List<Comment> findAllByParentPost(Post post);
}
```

Figur 99. Java Persistence repository for kommentarer

4.7.2.5 security - Autentisering, autorisering og passord

Denne pakken inneholder konfigurasjon og oppsett for alt av sikkerhet. Her setter vi opp alt fra BCrypt for passord til web security for autentisering. For at en bruker skal kunne logge inn må vi motta brukernavn og passord, og sjekke dette passordet opp mot det hashede passordet som er lagret i databasen. Når en bruker gjør kall til ressurser som trenger autentisering må det sendes med en *Authorization header* med en token fra innloggingen. I security pakken ligger annoteringene *@AuthorizeUser* og *@CurrentUser*. *@AuthorizeUser* bestemmer at et endepunkt krever autentisering og *@CurrentUser* henter brukerobjektet til brukeren som eier token brukt i forespørselen.

4.7.2.6 service - Logikk

Vi lager en service for hver modell og der bruker vi repository til å lagre, hente, endre og slette objekter. Dette blir lignende som *crud* pakken vi laget i FastAPI. Det er servicen som tar seg av sjekk av diverse ting og kaster feil om ting ikke er som det skal være. For eksempel lager vi en *404 Not Found* feil om noe ikke blir funnet, en *401 Forbidden* feil om noen prøver å endre på en post de ikke eier og så videre. Her skriver vi også en kort respons som skal beskrive problemet slik at klienten kan vite hva som gikk galt.

```java
/**
 * Get a user by username.
 *
 * @param username the username of the user.
 * @return the user with the given username.
 */
public User getUser(String username) {
    User user = repository.findByUsername(username);
    if (user == null) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND,
                String.format("User %s was not found", username));
    }

    return user;
}
```

Figur 100. Metode i brukerservice for å hente en spesifikk bruker

Å samle logikken på ett sted, altså i service klassene, sparer oss for arbeid. Om vi ikke gjenbruker kode må vi skrive samme funksjonalitet flere plasser og lager dermed duplikat kode. Dette er ikke effektivt, og det er enda mer arbeid om den funksjonaliteten skal endres senere. Da må man gå gjennom alle steder det er implementert og endre koden. Siden vi bruker metoden fra service klassen trenger vi bare å endre funksjonaliteten der om nødvendig.

4.7.2.7 resources - Ressurser og konfigurasjon

I *resources* ligger det kun én fil. Dette er *application.properties*. Properties filen består av variabler som brukes av Spring eller andre avhengigheter. Dette er altså måten vi håndterer konfigurasjon av serveren. På bildet er det satt opp konfigurasjon for å bruke en H2 database fil og ikke en database server. Vi setter også innstillinger for springdoc i denne filen.

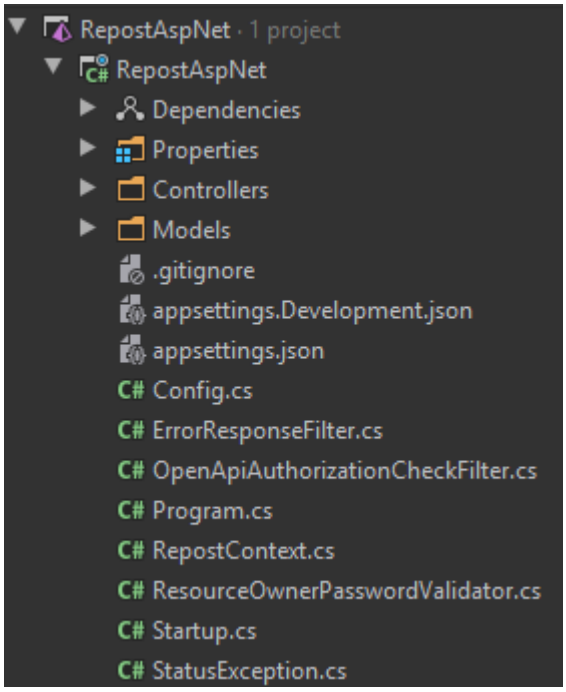
```
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:h2:file:./repostdb
spring.datasource.username=
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
springdoc.api-docs.path=/openapi.json
springdoc.swagger-ui.path=/api/swagger
```

Figur 101. Konfigurasjoner for Spring Boot i application.properties

4.7.3 ASP.NET implementasjon

Prosjektet for Repost i ASP.NET er satt opp etter konvensjon av Microsoft, hvor vi brukte template kode som de laget (Addie and Dykstra, 2020). Nedenfor er oversikt over de viktigste komponentene. Diverse filer her er lagt i hovedmappen til prosjektet ettersom vi ikke følte at det fantes noen god måte å kategorisere disse på. Det er disse filene som ikke er beskrevet i oversikten nedenfor.

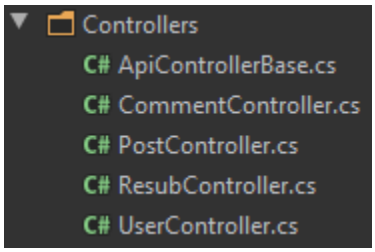
- Properties - innstillinger for debugging
- Controllers - klasser som definerer alle endepunkt i API-et og deres funksjonalitet
- Models - datamodeller for både databasen og JSON skjema
- Config.cs - konfigurasjoner for OAuth2, CORS og database
- Program.cs - hovedfila som definerer *Main* funksjonen
- RepostContext.cs - klasse for kommunikasjon med databasen
- Startup.cs - oppsett av deler av ASP.NET tjenesten som skal brukes



Figur 102. Fil- og mappestruktur i ASP.NET prosjektet i JetBrains Rider

4.7.3.1 Kontrollere

I ASP.NET implementasjonen ligger både API og grunnleggende logikk i kontroller klassene. Disse ligger i *Controllers* mappen, og definerer ulik funksjonalitet i API-et. Vi valgte å ikke separere logikk og API her ettersom ASP.NET har god støtte for å gjenbruke API endepunkt som vanlige metoder. Denne funksjonaliteten må bli skrudd på i *Startup*.cs:



```
services.AddControllers().AddControllersAsServices()
```

Da er det mulig å referere i kryss blant kontrollerne for å bruke metoder definert andre steder. Dette måtte vi ta nytte av i f.eks *ResubController* hvor vi har behov for å hente en bruker basert på brukernavnet sendt i forespørselen. Da krever denne kontrolleren tilgang til *UserController*, som kan oppnås ved å legge den til i konstruktøren.

```
private readonly UserController _userController;

public ResubController(RepostContext context, UserController userController) : base(context)
{
    _userController = userController;
}
```

Figur 103. Bruk av UserController i ResubController klassen

ApiControllerBase er vår egen derivasjon av *ControllerBase*. Denne er brukt til å definere database konteksten, *RepostContext*, slik at den er tilgjengelig i alle kontrollere. Her ligger også en metode for å hente brukeren som sendte en forespørsel til API-et dersom det ble sendt med en autentiserings token. Andre klasser tar *ApiControllerBase* i bruk ved å arve fra den. Prefix for stien til alle endepunktene i kontrolleren defineres med *[Route]* attributten.

```
[Route( template: "/api/resubs")]
public class ResubController : ApiControllerBase
{
```

Figur 104. Bruk av ApiControllerBase for å definere en kontroller

I kontrollerne lager vi alle endepunktene til API-et ved å spesifisere HTTP metode og sti til endepunktet. Endepunktet i Figur 105 bruker *GetAuthorizedUser* metoden som er definert i *ApiControllerBase* for å returnere den innloggede brukeren. Her kan vi også bruke *[Authorize]* for å definere dette som et endepunkt hvor man må tilføye en autentiseringstoken i API forespørselen.

```
/// <summary>Get Current User</summary>
/// <remarks>Get the currently authorized user.</remarks>
[HttpGet]
[Route( template: "me")]
[Authorize]
public User GetCurrentUser()
{
    return GetAuthorizedUser();
}
```

Figur 105. Bruk av GetAuthorizedUser metoden definert i ApiControllerBase

For alle endepunktene skriver vi XML for å dokumentere dem. Dersom et endepunkt har flere mulige responser, kan disse defineres med *[ProducesResponseType]* attributten. Standard responsverdi er *200 OK*, men når vi definerer flere responser må vi også ha med en respons for *200 OK* manuelt.

```
/// <summary>Get Resub</summary>
/// <remarks>Get a specific resub.</remarks>
[HttpGet]
[Route( template: "{resub}")]
[ProducesResponseType( statusCode: StatusCodes.Status200OK)]
[ProducesResponseType(typeof(ErrorResponse),  statusCode: StatusCodes.Status404NotFound)]
public Resub GetResub([FromRoute(Name = "resub")] string name)
{
```

Figur 106. Endepunkt for å hente en resub i ResubController

Eksempelet ovenfor viser også hvordan vi kan deklarere parameter i stien til et endepunkt. Her bruker vi attributten *[FromRoute]* for å hente variabelen fra stien, slik at den kan bli brukt videre i logikken til endepunktet.

4.7.3.2 Datamodeller

Datamodellene ligger i *Models* mappen. Disse definerer både database modeller og JSON skjema samtidig, som i Spring Boot implementasjonen. Vi bruker *[Table]* attributten for å definere navn på tabellene til de modellene som også skal være med i databasen. ASP.NET vil lete etter et felt med navn *Id* og bruke det som primærnøkkel i databasen.

For felt som skal finnes lokalt og brukes i databasen, men som også ikke skal sendes med JSON skjema bruker vi *[JsonIgnore]* attributten for å ekskludere dem fra skjemaet. Vi kan også bruke *[JsonPropertyName]* attributten for å endre navnet til felt slik at de bruker snake case navnestil framfor camel case.

```
public class CreateUser
{
    public string Username { get; set; }
    public string Password { get; set; }
}
```

Figur 107. Datamodell som skjema for å lage en ny bruker

```
[Table( name: "users")]
public class User
{
    [JsonIgnore] public int Id { get; set; }

    public string Username { get; set; }
    public string Bio { get; set; }

    [JsonPropertyName("avatar_url")] public string AvatarUrl { get; set; }

    public DateTime Created { get; set; }
    public DateTime? Edited { get; set; }

    [JsonIgnore] public string HashedPassword { get; set; }

    [JsonIgnore] public List<Resub> Resubs { get; set; }
    [JsonIgnore] public List<Post> Posts { get; set; }
    [JsonIgnore] public List<Comment> Comments { get; set; }
}
```

Figur 108. Datamodell og skjema for en bruker

I modellene kan vi lage relasjoner for databasen ved å henvise til de andre datamodellene. ASP.NET vil da automatisk lage fremmednøkler for disse objektene. I tilfellet i *Resub* refererer vi til lister av våre andre modeller. I de andre modellene er en referanse definert andre veien, og det er da disse objektene som får fremmednøkkel for brukerobjektet. Altså har *Resub* en definisjon som bildet under. Denne funksjonaliteten er unik i forhold til FastAPI og Spring, hvor relasjoner eksplisitt må defineres som en relasjon.

```
[JsonIgnore] public User Owner { get; set; }
```

Figur 109. Eieren til en ressurs som en relasjon i databasen

4.7.3.3 Database

Databasen må legges til som en *Context* i oppstartskonfigurasjonen i *Startup.cs*. Denne konfigureres som en service til applikasjonen. Her har vi koblet opp en konfigurasjon for kobling mot en PostgreSQL database. Hvis denne konfigurasjonen ikke er satt kjører vi en *InMemoryDatabase*. Dette er en spesiell implementasjon av databasen hvor ASP.NET bruker datatyper i C#, slik at det ikke kreves noen database . Det vil altså ikke lagres til disk og er dermed kun nyttig i utvikling.

```
var connectionString = _config.DatabaseConnectionString;
if (string.IsNullOrWhiteSpace(connectionString))
{
    _log.Add((LogLevel.Warning,
        "No database connection string provided. Defaulting to in-memory database"));
    services.AddDbContext<RepostContext>( optionsAction: options =>
        options.UseInMemoryDatabase( databaseName: "RepostContext"));
}
else
{
    services.AddDbContext<RepostContext>( optionsAction: options => options.UseNpgsql(connectionString));
}
```

Figur 110. Oppsett av database med PostgreSQL eller database i minne

For å forsikre om at databasen blir opprettet når en PostgreSQL database er brukt må vi legge til en ekstra konfigurasjon i *Program.cs* fila. Denne kjører *Database.EnsureCreated()* på konteksten slik at alle tabellene blir opprettet når programmet starter dersom de ikke allerede finnes.

Ved oppsett av databasen må vi også henvise til *RepostContext*, slik at ASP.NET vet hvilke modeller som skal være med i databasen. Det er i *RepostContext* vi definerer alle koblinger mellom datamodeller og database. Merk at dette steget er forskjellig fra Spring Boot, hvor vi lager *Repository* klasser for å kommunisere med databasen. Her lager vi heller datasett av typen *DbSet*, som kan brukes til å gjøre alle operasjonene vi trenger av databasen. ORM vil bli basert på modellene definert og brukt i denne klassen.

```
public class RepostContext : DbContext
{
    public RepostContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<User> Users { get; set; }
    public DbSet<Resub> Resubs { get; set; }
    public DbSet<Post> Posts { get; set; }
    public DbSet<PostVote> PostsVotes { get; set; }
    public DbSet<Comment> Comments { get; set; }
    public DbSet<CommentVote> CommentsVotes { get; set; }
```

Figur 111. ORM sett med alle tabellene i databasen

I *RepostContext* kan vi også sette opp primærnøkler om de ikke kan defineres i datamodellene. Dette har vi gjort for *PostsVotes* og *CommentsVotes* som krever komposittnøkler. Den eneste måten å definere komposittnøkler på i ASP.NET er i database kontekst definisjonen slik som vist i Figur 112.

```
// Assign composite keys to vote tables
builder.Entity<PostVote>().HasKey(p :PostVote => new {p.UserId, p.PostId});
builder.Entity<CommentVote>().HasKey(c :CommentVote => new {c.UserId, c.CommentId});
```

Figur 112. Konfigurasjon av komposittnøkler til relasjonene for stemming

SQL queries gjøres gjennom metoder definert av ASP.NET for *DbSet* objektene. I Figur 113 er bilde av en query for å få en side av resuber. Merk at her må relasjoner inkluderes i query ettersom databasen ikke bruker lazy loading. Dette gjøres ved bruk av *Include* metoden.

```
return Db.Resubs // DbSet<Resub>
    .Skip(page * pageSize)
    .Take(pageSize) // IQueryable<Resub>
    .Include( navigationPropertyPath: r :Resub => r.Owner) // IIncludableQueryable<Resub,User>
    .ToList(); // List<Resub>
```

Figur 113. Forespørsel til databasen av en liste med resuber med sideinndeling og henting av relasjoner

4.7.3.4 Konfigurasjon

Der det skal være nødvendig å kunne konfigurere applikasjonen har vi brukt *Configuration* klassen i ASP.NET. Denne har vi koblet opp mot vår egen *Config* klasse slik at vi kan få et grensesnitt for alle mulige konfigurasjoner. *Configuration* klassen som er inkludert har et veldig grundig system for konfigurasjon, og vil lete etter nøkler både i konfigurasjonsfilene i *Properties* mappen og miljøvariabler i operativsystemet.

```
public interface IConfig
{
    IEnumerable<ApiResource> Apis { get; }
    IEnumerable<Client> Clients { get; }
    IEnumerable<string> Origins { get; }
    X509Certificate2 SigningCredential { get; }
    string DatabaseConnectionString { get; }
}
```

Figur 114. Grensesnitt for alle konfigurasjonene i ASP.NET applikasjonen

```
public IEnumerable<string> Origins => (Configuration["Origins"] ?? Configuration["ORIGINS"] ??
    "http://localhost;http://localhost:8080").Split( separator: ';');
```

Figur 115. Konfigurasjon for CORS origins

Figur 114 demonstrerer implementasjon av Config klassen, hvor vi har en liste over tillatte CORS origins. Merk at vi bruker to varianter av navnet til nøkkelen, ettersom den kan både være

definert i en properties fil eller definert som en miljøvariabler. Her følger vi da altså navnestandarden til de ulike måtene av konfigurasjon.

For å støtte miljøvariabler må vi også inkludere en konfigurasjon i *Program.cs* fila. Her setter vi prefix *REPOST_* foran alle miljøvariablene, slik at de ikke vil kollidere med miljøvariabler fra andre program. Navnet på *Origins* miljøvariabelen blir da effektivt *REPOST_ORIGINS*.

```
return Host.CreateDefaultBuilder(args)
    .ConfigureAppConfiguration((context, config) => config.AddEnvironmentVariables("REPOST_"))
```

Figur 116. Konfigurasjon av miljøvariabler

4.7.3.5 Passord

For validering av passord bruker vi *BCrypt.Net.BCrypt*. Den samme NuGet pakken er brukt i endepunktet for å lage en ny bruker for å hashe passordet. Altså trengte vi ikke å definere denne funksjonaliteten noen steder for å bruke dem, ettersom både hashing og verifisering er statiske metoder i denne pakken.

Klassen som definerer hvordan passord skal verifiseres med *ResourceOwnerPassword* som grant type i klienten implementerer *IResourceOwnerPasswordValidator*. Her må vi endre result basert på om passordet er riktig eller ikke.

```
public async Task ValidateAsync(ResourceOwnerPasswordValidationContext context)
{
    context.Result = ValidateUser(context.UserName, context.Password);
}

private GrantValidationResult ValidateUser(string username, string password)
{
    var user = _repostContext.Users.SingleOrDefault(u :User => u.Username == username);
    if (user == null || !BCrypt.Net.BCrypt.Verify( text: password, hash: user.HashedPassword))
    {
        return new GrantValidationResult( error: TokenRequestErrors.InvalidGrant);
    }

    return new GrantValidationResult( subject: user.Username, authenticationMethod: "password");
}
```

Figur 117. ResourceOwnerPasswordValidator som verifiserer passordet til en bruker

4.7.3.6 Eksterne biblioteker

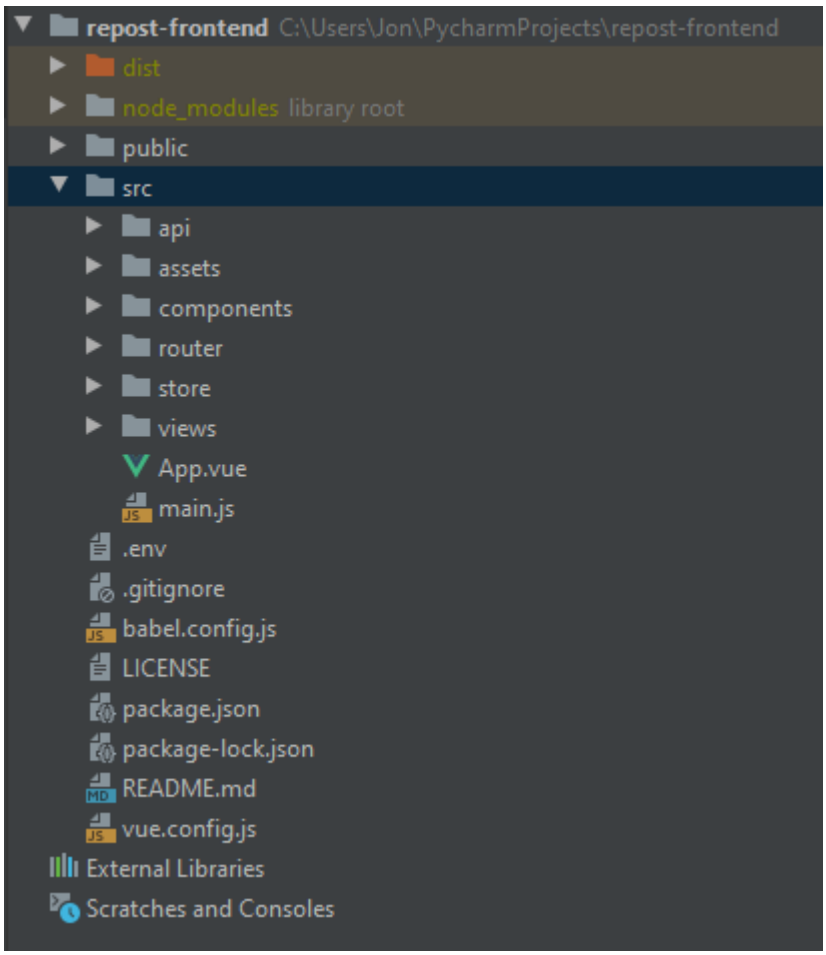
ASP.NET gjør det veldig enkelt å definere biblioteker, kalt NuGet pakker, som kreves for bygging av applikasjonen. Disse kan listes i selve *RepostAspNet.csproj* fila, som vil være prosjektfila til applikasjonen. Her kan vi henvise til pakkenavn og versjonsnummer, og slik forsikrer vi oss om at vårt eget testet miljø skal være likt hos alle som bygger applikasjonen.

```
<ItemGroup>
    <PackageReference Include="BCrypt.Net-Next" Version="3.3.3"/>
    <PackageReference Include="IdentityServer4" Version="3.1.2"/>
    <PackageReference Include="IdentityServer4.AccessTokenValidation" Version="3.0.1"/>
    <PackageReference Include="Microsoft.EntityFrameworkCore.InMemory" Version="3.1.3"/>
```

Figur 118. Avhengigheter som definert i RepostAspNet.csproj

4.7.4 Vue.js Implementasjon

Oppsettet til Vue prosjektet ble automatisk oppsatt med PyCharm. PyCharm følger automatisk vue-cli mappehierarki. Alle JavaScript-, Vue-, CSS- og bildefiler tilhører *src* mappen. I *public* skal det ligge en primær HTML fil som en bruker blir sendt til når de skal laste opp nettsiden. Vue kobles automatisk til HTML filen. Prosjektfiler ligger direkte i prosjektmappen. Prosjektet blir kjørt med kommandoen *npm run serve* som tilhører Node.js.



Figur 119. Fil- og mappestruktur i Vue.js prosjektet i JetBrains PyCharm

4.7.4.1 Hovedapplikasjonsfilene

Hovedapplikasjonsfilen i Vue er bundet til alle skripter, komponenter og andre filer som blir brukt i front-end applikasjonen. I *App.vue* filen ligger det en *<router-view>* tag. I denne HTML taggen blir andre komponenter vist etter brukerens forespørsel (url) i nettsiden. Komponenter blir heller lastet ned direkte på siden uten at brukeren må laste ned siden på nytt. Dette er det største skillet mellom Single-page application og multi-page application.

```
<template>
    <div id="app">
        <router-view></router-view>
    </div>
</template>

<script>
    require('@/assets/css/reset.css')

    export default {
        name: "App.vue"
    }
</script>
```

Figur 120. App.vue fila som definerer hovedappen

Det eneste som må gjøres i HTML filen som ligger i *public* mappen er å definere en * * tag med id *app*. Denne id taggen bruker vi til å definere stedet hvor hovedkomponenten, altså *App.vue*, skal vises.

```
<body>
    <noscript>
        <strong>We're sorry but website doesn't work properly without JavaScript enabled. Please enable it to
            continue.</strong>
    </noscript>
    <div id="app"></div>
</body>
```

Figur 121. index.html fila hvor app visningen blir plassert

I JavaScript filen som ligger i roten *av* src mappen blir verktøy og andre programmeringsgrensesnitt (API) importert og satt opp. API-ene og verktøyene blir satt opp med Vue rammeverket, deretter blir Vue fortalt at den skal monteres til elementet med id *app* med VueRouter.

```
import Vue from 'vue'
import App from '@/App.vue'
import router from '@/router'
import api from '@/api'
import store from '@/store'
import NProgress from 'nprogress'

Vue.config.productionTip = false
Vue.prototype.$http = api.instance
Vue.prototype.$store = store(api.instance)

Vue.prototype.$load = NProgress.start
Vue.prototype.$loaded = () => {
    NProgress.done()
    return true
}

new Vue({
    router,
    render: h => h(App)
}).$mount( elementOrSelector: '#app')
```

Figur 122. Hovedfilen til JavaScript hvor Vue instansen av appen blir opprettet

4.7.4.2 Monterte programmeringsgrensesnitt

Det er en egen fil for montering av programmeringsgrensesnitt til Vue. I denne JavaScript filen blir det laget en instans av axios, som blir brukt for å innhente informasjon fra API-et. Dette blir satt opp ved å eksportere instansen slik at vi kan bruke den utenfra. Denne importerer vi i hovedfila som vist i Figur 122, hvor den settes som en *Vue.prototype*. Den blir da tilgjengelig i alle komponenter gjennom JavaScript nøkkelordet *this*.

I denne axios instansen blir det satt opp et par interceptors. Disse fungerer som funksjoner som skal kjøres før eller etter hver request i axios. Den første interceptoren konfigurer basis URLen til der back-end serveren ligger, i tillegg til å sette *Authorization* header til *userToken* som vil ligge i localStorage dersom brukeren er logget inn. Slik kan vi enkelt alltid sende med autentiseringstoken når brukeren er innlogget i hver request. Denne interceptoren blir brukt for requests til back-end.

```
instance.interceptors.request.use( onFulfilled: config => {
    config.baseURL = apis[parseInt(localStorage.selectedApi)].url

    // Always add authorization header if token is stored
    if (localStorage.userToken) {
        config.headers['Authorization'] = `Bearer ${localStorage.userToken}`
    }

    return config;
});
```

Figur 123. Interceptor til axios som legger til API url og autentiserings token

Den andre interceptoren fjerner *userToken* dersom serveren gir feilmeldingen *401 Unauthorized*. Altså sletter vi påloggingsinformasjonen dersom vi får beskjed fra serveren om at vi ikke er logget inn.

```
instance.interceptors.response.use( onFulfilled: response => {
    return response
}, onRejected: error => {
    // Remove authentication token when invalid
    if (localStorage.userToken && error.response.status === 401) {
        delete localStorage.userToken
    }

    return Promise.reject(error)
})
```

Figur 124. Interceptor til axios som fjerner token når respons fra server er 401 Unauthorized

I *api* pakken setter vi også opp menyvalget for valg av API. Denne informasjonen lagrer vi i en localStorage nøkkel *selectedApi* slik valgt API blir beholdt til neste gang brukeren besøker nettsiden. Konfigurasjonen for hvilke API-er som er satt opp defineres gjennom miljøvariabler, eller *.env* fila som ligger i hovedmappen til prosjektet.

4.7.4.3 Ressurser

assets mappen inneholder statiske ressursfiler for nettsiden. Der ligger logoen til nettsiden og en CSS fil som tilbakestiller standard stilen som nettlesere har. CSS filen er hentet fra DavidWells på GitHub (DavidWells, 2013). Filen har ikke noen lisens, som vil si at den er offentlig eiendom, og har blitt litt endret for å passe bedre til nettsiden.

4.7.4.4 Komponenter

Vue komponenter er deler av en nettside som blir brukt flere ganger i forskjellige sammenhenger. Et eksempel på en Vue komponent i produktet er navigasjonsbaren som ligger i toppsiden av nettsiden. Den blir brukt i flere sider, og er satt opp ved å skrive inn navnet på komponenten i HTML delen, og så importere komponenten i JavaScript delen. På denne måten er komponenter i Vue tett knyttet mot HTML dokumentet, ettersom du kan bruke de som en vanlig HTML tag.

```
<template>
    <div id="post-data" class="post-data">
        <Navbar/>
```

Figur 125. Bruk av Navbar komponenten i en annen komponent

```
<script>
    import Navbar from '@/components/Navbar';
    import PostList from '@/components/PostList';
    import Notice from "@/components/Notice";

    export default {
        name: "Home",
        components: {Notice, Navbar, PostList},
```

Figur 126. Importere Navbar komponenten for videre bruk i HTML

4.7.4.5 VueRouter

Med VueRouter kan vi definere hvilke filer som skal vises ettersom hva URLen er. Det blir lastet tilsvarende Vue fil som tilhører siden. Dette er definert med *component* egenskapen. Andre egenskaper som *redirect*, *props* og *navn* kan også bli definert. *props* brukes til å sende data mellom komponenter når man bruker *<router-link>* tag, eller som parameter i endepunktet. *redirect* egenskapen forteller hvor brukeren skal bli omdirigert om de taster inn URLen, om nødvendig.

Til slutt lages VueRouter med alle routes som ble definert. Her velger vi at VueRouter skal bruke *history* som modus til navigering. Dette vil si at Vue manipulerer historikken i nettleseren for å endre på URLen på en oversiktlig måte.

```
Vue.use(VueRouter);

const routes = [
    {path: '/', component: Home, redirect: ''},
    {path: '/users', component: Home},
    {path: '/login', component: Login, props: true, name: 'login'},
    {path: '/signup', component: SignUp},
    {path: '/users/:username', component: User, props: true},
    {path: '/me/edit', component: UserEdit},
    {path: '/resubs', component: Resubs},
    {path: '/resubs/create', component: ResubForm},
    {path: '/resubs/:resubname/posts', component: Resub, props: true},
    {path: '/resubs/:resubname', redirect: '/resubs/:resubname/posts', component: Resub, props: true},
    {path: '/resubs/:resubname/edit', component: ResubForm, props: true, name: 'editResub'},
    {path: '/resubs/:resubname/posts/create', component: PostForm, props: true},
    {path: '/resubs/:resubname/posts/:post_id', component: Post, props: true},
    {path: '/resubs/:resubname/posts/:post_id/edit', component: PostForm, props: true, name: 'editPost'},
    {path: '*', component: NotFound}
];

const router = new VueRouter( options: {
    mode: 'history',
    routes
})
```

Figur 127. Alle endepunkt i nettsiden satt opp med VueRouter

4.7.4.6 Lagring

Mappen *store* (lagring) definerer et lagringspunkt som komponentene burde bruke. Her settes abstraksjoner og ekstra funksjonalitet på autentiseringstoken og valgt API. På denne måten unngår vi å måtte bruke localStorage manuelt, og vi får et ekstra lag som vi kan lett endre senere om vi har behov for det. Her har vi også en egen metode for å hente brukeren som er logget inn ved bruk av autentiseringstoken. Denne lagrer brukeren til minne etter første forespørsel, slik at vi slipper å spørre om den samme dataen om og om igjen.

4.7.4.7 Sider

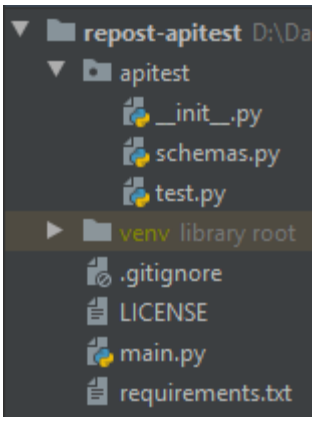
views mappen inneholder Vue komponenter som blir brukt som selvstendige sider. I motsetning til *components* mappen blir ikke disse filene brukt noen andre steder enn i VueRouter. Mange av disse sidene bruke axios for å innhente informasjon fra back-end serveren.

Axios kan bli brukt for å hente, lage, redigere og slette data. For å kunne lage og redigere data må man definere på forhånd hva dataen er før man sender den. Axios er forhåndskonfigurert til å inkludere den innloggede brukeren og hvilken back-end server dataen blir hentet fra, som beskrevet i Monterte programmeringsgrensesnitt (4.7.4.2).

4.7.5 Implementasjon av testverktøyet

Testverktøyet ble implementert som planlagt og har fungert etter forventning. Mappestrukturen er satt opp som i FastAPI prosjektet, hvor en liten forskjell er at *main.py* er plassert utenfor *apitest* pakken. Dette må gjøres slik at pakken kan brukes i *main.py* fra et vanlig miljø. I FastAPI prosjektet er ikke dette nødvendig ettersom main fila skal kjøres med en ASGI server, i vårt tilfelle uvicorn, som vil alltid sette opp miljøet slik at pakkene blir funnet.

main.py definerer den kjørende komponenten av testeverktøyet. Denne tar seg av kommandolinje argumenter for å kjøre alle testene, hvis funksjonalitet videre er definert i *apitest* pakken. I *requirements.txt* er alle eksterne Python bibliotek som kreves for å kjøre testverktøyet. Det eneste biblioteket som kreves er *requests* (3.3.6.1), som gjør det enkelt å sende forespørsler til API-ene.



Figur 128. Fil- og mappestrukturen til testverktøyet i JetBrains PyCharm

4.7.5.1 API skjema

For å teste på de ulike objekttypene vi har definert i API modellen vår laget vi forskjellige klasser for å definere API skjemaene som brukes for å sende JSON forespørslene fram og tilbake. Disse er definert i *schemas.py* og er internt laget som en Python dataclass. Disse klassene definerer alle felt for de forskjellige skjemaene, og noen ekstra properties for de diverse handlingene som kan gjøres, eller de andre skjemaene som f.eks *Create* eller *Edit* skjema.

Klassen skal kun instansieres før tester, og ikke av responsen fra en server. Altså lager vi dataene som skal gjennom API-et lokalt i testverktøyet, og så kan vi sammenligne dataen med responsen fra server. Dette er viktig for å forsikre at testverktøyet fullstendig definerer hva som er forventet fra APIen, og dermed kan feil fanges opp på riktig måte.

Nedenfor er definisjonen av skjemaet for en Resub. Den har felter for *owner_username*, *name* og *description*. Dette er de feltene som er viktig å teste i en respons fra APIen. Felt som *created* og *edited* er ikke inkludert ettersom det ikke er noen særlig nytte å teste på de. *owner_username* er det eneste feltet uten en standardverdi, og dette er fordi når en *Resub* lages så må den ha en eier. Dette blir satt automatisk i serveren ved bruk av autentiserings token, men i testverktøyet må den settes manuelt for å forsikre at serveren setter riktig eier av en resub.

```python
@dataclass
class Resub:
    owner_username: str
    name: str = field(default_factory=random_string)
    description: str = field(default_factory=random_string)

    create = property_to_dict(name='name', description='description')
    edit = method_edit(description='description', owner_username='new_owner_username')
    compare = method_compare()
```

Figur 129. Skjema for modeller i testverktøyet

Siden testene skal kunne kjøre på et API uavhengig av tilstand må de viktigste feltene være tilfeldig laget. Det er oppnådd her ved å bruke Python dataclass funksjonen *field*, som kan kjøre en funksjon når objektet lages. Her er funksjonen satt til *random_string*, som er en mapper til en innebygd Python funksjon *secrets.token_hex*. Den lager altså en streng med 8 tilfeldige tegn.

```python
random_string = partial(secrets.token_hex, 8)
```

Figur 130. Definisjon av en funksjon som lager en streng med 8 tilfeldige tegn

De tre nederste definisjonene i *Resub* klassen er ikke felt, men heller properties og metoder som kan brukes i testing.

create er en property som vil returnere en Python dictionary hvor kun navn og description er inkludert. Da får vi altså en dictionary som følger *CreateResub* modellen i API-et vårt. For å definere *create* bruker vi vår egen *property_to_dict* funksjon, som vil gjøre om *create* til en Python property som returnerer en dictionary med de feltene som er inkludert. Strukturen vil da se slik ut:

```
{'username': '934748d88a561d65', 'password': '26fcee9256bad967'}
```

edit fungerer noenlunde likt som *create*. Her vil den returnere en dictionary som følger *EditResub* modellen i API-et. Men til forskjell fra *create* vil den også ta inn nye felter som argument, og endre disse lokalt. På den måten kan vi både endre felt i klassen og lage en *EditModel* samtidig, slik at vi kan teste om de lokale endringene også ble utført i API-et. *edit* er definert med vår egen *method_edit* funksjon, som vil gjøre om *edit* til en metode som tar inn de feltene som skal endres og returnere en dictionary av feltene og de nye verdiene.

Både *create* og *edit* defineres med Python keyword arguments. Dette kunne ha vært en liste over alle felt som skal inkluderes, men denne syntaksen gir mer fleksibilitet på navngiving av feltene. Navnet før = tegnet vil være navnet på feltet lokalt, mens navnet etter = tegnet blir navnet satt i det resulterende skjemaet. Dette kan vi se i bruk i *edit* feltet til *Resub* klassen i Figur 129, hvor *Resub* feltet *owner_username* blir skrevet som *new_owner_username* i det nye skjemaet.

Til slutt er *compare* en metode som tar inn JSON objektet fra serveren og sammenligner alle feltene i det objektet med det lokale objektet. Den returnerer *True* hvis alle feltene i testverktøyet stemmer overens med feltene i responsen fra API-et.Siden vi ikke kan vite hvilken id et nytt objekt blir tildelt av serveren vil ikke det lokale objektet og responsen stemme overens. Derfor kan compare metoden oppdatere det lokale objektet med felt som blir generert av serveren slik at de blir like.

```
compare = method_compare(id='id')
```

Figur 131. Definere compare slik at den oppdaterer ID nøkkelen basert på respons fra API

4.7.5.2 API tester

Alle testene i testverktøyet er definert i *test.py*. Her er hovedpunktet for testing funksjonen *test_everything*, som tar inn en URL for API-et som skal testes, kjører alle testene på den og til slutt returnerer et *TestStart* objekt med informasjon om hvordan testen gikk.

```
def test_everything(url: str, logging: bool = True) -> TestStats:
    """Perform all tests and return statistics when passed."""
```

Figur 132. Definisjon av test_everything som returnerer en statistikk av testene

```
@dataclass
class TestStats:
    count: int = 0
    elapsed_seconds: int = 0
```

Figur 133. Dataklasse for å lagre statistikk i testene

Videre i denne funksjonen er en *test* funksjon definert. Denne laget vi for å redusere kode som kreves for testing, slik at vi kunne fokusere på operasjonene som må utføres framfor koden bak. Her definerer vi også en *log* funksjon, som ganske enkelt printer informasjon om testene som blir utført til terminalen.

Testene er definert kronologisk og vil stoppe dersom API-et gir en respons som ikke var forventet. I Figur 134 ser vi hvordan en ny bruker er laget lokalt med et API skjema. Denne brukeren får da et tilfeldig brukernavn og passord. Det er viktig at objekter er definert her, slik at de kan gjenbrukes og helt til slutt i løpet av testen også slettes. Dermed kan vi bruke minimalt med forespørsler til API-et ettersom vi ikke trenger å lage nye objekter hele tiden og kjøre mange API forespørsler som ikke behøver testing. Testene fortsetter slik nedover gjennom funksjonen.

```
user1 = User()

log('Test get user1 before creation')
test(get, f'/users/{user1.username}', status=404)

log('Test create user1')
test(post, '/users/', status=201, compare=user1, json=user1.create)
```

Figur 134. Bruk av testene som er definert kronologisk og med et lett leselig format

Deretter kjøres en test som sørger for at brukeren ikke finnes i systemet. Dette er en enkel test på om serveren gir *404 Not Found* når en ressurs ikke finnes. Negative tester som denne er de enkleste å lage, ettersom de kun trenger å sjekke statusen på en forespørsel. Fra Figur 134 ser vi at testen defineres med HTTP metode som første parameter, så endepunktet som skal testes og til slutt forventet statuskode.

Dersom statuskoden fra serveren ikke er *404* vil hele testen stoppe der og vise detaljert informasjon på hvorfor testen stoppet.

```
AssertionError: GET /users/test
Got: 200
Expected: 404
Response: {"username":"test","bio":null,"avatar_url":null,"created":"2020-05-03T14:29:48.169675+00:00","edited":null}
```

Figur 135. Feilmelding med detaljert informasjon når en test ikke går gjennom

4.7.5.3 Testing av opprettelse

Videre lages objektet på serveren ved bruk av *POST* til endepunktet for å lage ny bruker. Her testes også status. JSON skjemaet for å lage en ny bruker er definert ved hjelp av *create* property til user objektet. *test* funksjonen tar også inn objektet i parameteret *compare*, hvor objektets *compare* funksjon vil bli kjørt etter testen for sammenligning av feltene. På denne måten kan vi utføre flere tester uten å skrive flere linjer med kode, som gjør det enklere å holde oversikt på selve testene, deres data og forventet respons.

```
log('Test create user1')
test(post, '/users/', status=201, compare=user1, json=user1.create)
```

Figur 136. Teste opprettelse av en bruker og sammenligne responsen fra API

4.7.5.4 Testing med autentisering

For å autentisere brukerne som blir opprettet i testen brukes *data* feltet i requests bibliotekets funksjoner. Dette kan vi sende med vår *test* funksjon, sammen med forventet status. Her må vi også lagre responsen fra serveren slik at vi kan bruke token senere i endepunkt som krever autentisering. Vi har også en ekstra test for å sjekke at access token er inkludert i responsen.

```
log('Test login user1')
user1_token = test(post, '/auth/token', status=200, data=user1.login)
assert 'access_token' in user1_token
```

Figur 137. Spørre om og lagre autentiserings token

login en er property i vårt lokale *User* skjema som følger vår OAuth2 modell. Her inkluderer vi login informasjonen og andre OAuth2 nøkler som kreves.

```
@property
def login(self):
    return {
        'grant_type': 'password',
        'username': self.username,
        'password': self.password,
        'client_id': 'repost',
        'scope': 'user'
    }
```

Figur 138. OAuth2 skjema for login

For å bruke token i en annen forespørsel kan vi bruke *token* parameteret i vår *test* funksjon. Dette er demonstrert i Figur 139, hvor vi sammenligner brukerobjektet vårt med brukeren som serveren gir tilbake ved bruk av autentiserings token.

```
log('Test get current user with user1 token')
test(get, '/users/me', token=user1_token, status=200, compare=user1)
```

Figur 139. Bruk av autentiserings token i en test med token keyword argument

For å forsikre om at autentisering fungerer på alle endepunkt gjør *test* metoden vår et par ekstra tester på alle endepunkt som krever autentiseringstoken. Testene sørger for at man får *401 Unauthorized* dersom token mangler, eller om den er uleselig. Her kan vi også legge til flere tester dersom det er nødvendig, som da effektivt blir kjørt på alle tester som har token.

```
# Do extra authorization tests when token is provided
if token and not skip_token_test:
    log('\tWithout authorization')
    test(method, endpoint, status=401, skip_token_test=True, **kwargs)

    log('\tWith invalid token')
    test(method, endpoint, token={'access_token': 'not.a.token'}, status=401, skip_token_test=True, **kwargs)
```

Figur 140. Ekstra tester i endepunkt som krever autentiserings token

4.7.5.5 Testing av lister

For å teste at en liste inneholder et objekt vi har laget må vi skrive en ekstra *assert* etter testen på responsen fra serveren. Vi valgte å gjenbruke *compare* funksjonen her, slik at alle felt i responsen kan testes. Testen blir bestått dersom ett av alle objektene blir sammenlignet riktig.

```
log('Test resub1 in get resubs')
resubs = test(get, '/resubs/', status=200)
assert any(resub1.compare(r) for r in resubs)
```

Figur 141. Testing av lister ved å bruke any funksjonen i Python

4.7.5.6 Testing av endring

For å teste endring av objekt, altså *PATCH* forespørsler, brukes *edit* metoden til skjemaet. Her brukes også *compare* metoden for å sjekke at det stemmer overens. En nyttig funksjon i *edit* metoden er nøkkelordet *apply*, som kan settes til *False* dersom vi forventer at forespørselen ikke skal funke. I slike tilfeller ønsker vi heller ikke å endre objektet lokalt, ettersom det ikke blir endret på serveren.

```
log('Test edit resub1 description as user1')
test(patch, f'/resubs/{resub1.name}', status=200, token=user1_token, compare=resub1,
    json=resub1.edit(description='User1 description'))

log('Test transfer resub1 ownership to nonexistent user')
test(patch, f'/resubs/{resub1.name}', status=404, token=user1_token,
    json=resub1.edit(new_owner_username=User().username, apply=False))
```

Figur 142. Test av gyldig og ugyldig endring av en bruker

Siden *PATCH* endepunktene våre bruker Content-Type *application/patch+json* så settes dette også automatisk av *test* metoden når en *PATCH* forespørsel blir sendt:

```
if method == patch:
    headers['content-type'] = 'application/patch+json'
```

Figur 143. Automatisk endring til application/patch+json for PATCH metoder

4.7.5.7 Gjentagende tester

Noen tester kan gjennomføres likt på forskjellige objekter, som voting av poster og kommentarer. Her gjør Python det lett å bare lage en ny funksjon i samme flyt som testene som vi kan gjenbruke til de objektene vi ønsker å teste.

```
def test_vote_entity(path: str, entity_name: str, entity: Union[Post, Comment]):
    log(f'Test user1 vote -2 {entity_name}')
    test(patch, f'{path}/vote/-2', status=422, token=user1_token)

    log(f'Test user1 vote 2 {entity_name}')
    test(patch, f'{path}/vote/2', status=422, token=user1_token)
```

Figur 144. Lage en intern funksjon for å kjøre samme testene på ulike ressurser

```
test_vote_entity(f'/posts/{post1.id}', 'post1', post1)
```

Figur 145. Bruk av en intern funksjon definert i test_everything funksjonen

4.7.5.8 Ytelsestest med testverktøyet

Testverktøyet var laget til å kjøre tester på systemet som en ekstra sjekk på at alle API-ene er implementert på samme måte og fungerer som de skal. Den skal teste alle endepunkt i API-et, med alle mulige responser fra serveren. Derfor kan vi enkelt utvide testen slik at den også støtter kjøring av flere komplette tester av API-et, og måle ytelse fra dette. Merk at dette ikke er i nærheten så nyttig som våre JMeter ytelsestester, ettersom den kun kjører én forespørsel til API-et av gangen. En slik ytelsestest kan utføres gjennom kommandolinje parameteret *--runs*:

```
python main.py http://localhost:8000 --runs 100
```

I eksempelet over vil testen kjøres 100 ganger, hvor tiden det tar alle 100 testene å kjøre blir målt og lagret gjennom *TestStats* objektet som er returnert av *test_everything* metoden. Her inkluderer vi noen forskjellige mål på dataen.

Bildet under viser mål av alle API-ene med 100 runder av testen. Vi laget en egen Discord kanal for denne dataen, ettersom den muligens kunne være nyttig. Realistisk burde testene kjøres mer enn 100 ganger for å gi et bra resultat.

Denne delen av testverktøyet fungerer mer som en demonstrasjon av hva som også er mulig. Det er ikke en viktig del av prosjektet i sin helhet, men viser allikevel fleksibiliteten av testverktøyet.

PC 04/23/2020

FastAPI 100 runs

https://hastebin.com/unokasojaf

```
Executed 100 runs in 290.58740400000005 seconds
Performed a total of 24300 tests

Stats:
        mean: 2.90587404
        median: 2.901172
        stdev: 0.024800489828137574
        variance: 0.0006150642957155553
        population stdev: 0.0246761758130874
        population variance: 0.0006089136527583997
```

Spring 100 runs

https://hastebin.com/avomagahuv

```
Executed 100 runs in 227.1019799999999 seconds
Performed a total of 24300 tests

Stats:
        mean: 2.2710198
        median: 2.212126
        stdev: 0.17294034590896107
        variance: 0.029908363243111104
        population stdev: 0.17207347154828947
        population variance: 0.02960927961067999
```

ASP.NET 100 runs

https://hastebin.com/ininuvarec

```
Executed 100 runs in 202.58371300000005 seconds
Performed a total of 24300 tests

Stats:
        mean: 2.02583713
        median: 2.020157
        stdev: 0.027845997689186188
        variance: 0.0007753995873061624
        population stdev: 0.027706417874440226
        population variance: 0.0007676455914331008
```

Figur 146. Resultater av ytelsestester med testverktøyet

4.8 Implementasjonsdetaljer for diverse teknologier

I dette kapittelet vil vi demonstrere i dyp detalj hvordan noen av de viktigere teknologiene er implementert. Dette innebærer mange detaljerte arkitekturer og bilder av kode som gjør at det kan være nyttig for å få et godt innblikk i arbeidsmengde og kompetanse som kreves av rammeverkene.

4.8.1 Bruke OAuth2 med password flow

4.8.1.1 FastAPI implementasjon

Ettersom vår struktur i FastAPI prosjektet har en egen pakke for *api* valgte vi å definere all kode for autorisering her. Autentisering, altså verifisering av passord, er definert utenfor *api* pakken men brukes likevel også i API nivået.

Det var veldig enkelt å implementere en grunnleggende OAuth2 spesifikasjon i FastAPI, da det er en veiledning for dette på nettsiden til FastAPI (*Simple OAuth2 with Password and Bearer - FastAPI*, no date). Først definerte vi et OAuth2 skjema med *OAuth2PasswordBearer* fra FastAPI. Dette er et OAuth2 skjema med password flow slik vi ønsker, og den er definert i *security.py* fila i *api* pakken. Her setter vi også en sti for hvor endepunktet for å få en token skal ligge. Dette er altså endepunktet for autentisering. Vi kan også legge ved mulige scopes og deres beskrivelse her

```
oauth2_scopes = {'user': 'User access'}

# NOTE: this path is hardcoded and correlates to repost.api.routes.auth.login
oauth2_scheme = OAuth2PasswordBearer(tokenUrl='/api/auth/token', scopes=oauth2_scopes)
```

Figur 147. Definisjon av OAuth2 skjema og scopes

Endepunktet for autentisering er definert i *auth.py* i *routes* pakken. Dette endepunktet krever en modell for skjemaet vi skal bruke i OAuth2 autentisering. Skjemaet for tokenet måtte vi lage selv.

```
class OAuth2Token(BaseModel):
    """Schema for an OAuth2 token"""
    access_token: str
    token_type: str
    scope: str
```

Figur 148. Skjema for API respons av en OAuth2 token

I endepunktet definerer vi dette skjemaet som modell for respons fra serveren. Som inndata kan vi bruke en *OAuth2PasswordRequestForm* fra FastAPI. Dette er en modell definert etter OAuth2 standarden for form data, og er egnet til bruk med password flow.

```
@router.post('/token', response_model=OAuth2Token,
             responses={status.HTTP_401_UNAUTHORIZED: {'model': ErrorResponse}})
async def login(db: Session = Depends(get_db), form_data: OAuth2PasswordRequestForm = Depends(),
```

Figur 149. Bruk av OAuth2Token som respons og OAuth2PasswordRequestForm til inndata

Internt i denne funksjonen trenger vi bare å autentisere brukeren basert på brukernavn og passord som er inkludert i dataene sendt i forespørselen. Ved feil passord kan vi sende en *400 BAD REQUEST.* Dersom brukeren er verifisert kan vi sende ut en OAuth2 token som bruker en nylig opprettet JWT. Samtidig setter vi scope basert på hvilke scopes som ble sendt i forespørselen. Her settes også standardverdi med alle scopes, dersom scope ikke er spesifisert i forespørselen.

```
db_user = crud.get_user(db, username=form_data.username)
if not db_user or not verify_password(form_data.password, db_user.hashed_password):
    raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail='Invalid login information')

scopes = form_data.scopes or list(oauth2_scopes.keys())
return OAuth2Token(access_token=create_jwt_token(username=db_user.username, scopes=scopes),
                   token_type='bearer', scope=' '.join(scopes))
```

Figur 150. Verifisering av brukerens passord og opprettelse av JWT

I tillegg ønsket vi å forsikre om at *client_id* er testet i vår OAuth2 modell. FastAPI gjør ikke dette automatisk, så det måtte også implementeres i denne funksjonen. Siden OAuth2 både støtter client informasjon gjennom form data og HTTP Basic Auth må begge formene testes her. Form data *client_id* er allerede definert av *OAuth2PasswordRequestForm*, mens HTTP Basic Auth må testes mot *Authorization* HTTP header. Denne må dekodes fra Base64 før den kan sammenlignes. Verdien av *Authorization* header kan hentes ved å bruke *Header* i parameteret i funksjonen til endepunktet.

```
async def login(db: Session = Depends(get_db), form_data: OAuth2PasswordRequestForm = Depends(),
                authorization: str = Header(None)):
    """Authorize using username and password."""
    client_id = form_data.client_id
    if not client_id and authorization:
        http_basic_auth = b64decode(authorization.replace('Basic ', '')).decode('ascii')
        client_id = http_basic_auth.split(':')[0]

    if not client_id or client_id != config.client_id:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail='Invalid client')
```

Figur 151. Manuell sjekk av client_id gjennom form data og HTTP Basic Auth

For å videre håndtere autorisering av endpoints laget vi en FastAPI dependency *authorize_user* som forsøker å verifisere en token og returnere brukernavnet. Det er i denne funksjonen vi må

manuelt dekode JWT og lese brukernavnet den tilhører. Samtidig må vi håndtere feilmeldinger manuelt dersom token er utgått eller ugyldig.

```
async def authorize_user(security_scopes: SecurityScopes, jwt_token: str = Depends(oauth2_scheme)) -> str:
    """Validate and return the username in the JSON Web Token."""
    try:
        payload = decode_jwt_token(jwt_token)
    except jwt.exceptions.ExpiredSignatureError:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail='The JSON Web Token has expired')
    except jwt.exceptions.DecodeError as e:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
                            detail=f'Failed to parse JSON Web Token: {str(e)}')
```

Figur 152. Manuell håndtering av feilmeldinger når JWT er ugyldig i authorize_user dependency

```
    return payload.get('sub')
```

Figur 153. Returnere brukernavnet fra data i JWT

Her kan vi også håndtere scopes. FastAPI støtter definisjon av scopes i videre dependency, men disse blir ikke verifisert automatisk. Dermed må verifisering av scopes foregå i denne funksjonen. Da må vi først inkludere et *SecurityScopes* objekt som parameter, som vil inneholde en liste over alle scopes som kreves for endepunktet som bruker denne dependency funksjonen. Da kan vi sammenligne scopes som er i JWT payload med scopes som kreves av endepunktet.

```
# Verify that the user has the required scopes for this endpoint
scopes = payload.get('scopes', [])
for scope in security_scopes.scopes:
    if scope not in scopes:
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail=f'Missing scope \'{scope}\'')
```

Figur 154. Manuelt verifisere riktige OAuth2 scopes

Nå kan vi bruke denne som en dependency på metoder som krever autorisering. Ettersom vi ofte også vil ha nytte av mer informasjon enn brukernavnet laget vi en dependency i *resolvers* fila som også henter brukeren fra databasen. I signaturen til denne funksjonen bruker vi *authorize_user* som en dependency. Her bruker vi også *Security* i stedet for *Depends* slik at vi kan sende med hvilke scopes som er tillatt.

```
async def resolve_current_user(username: str = Security(authorize_user, scopes=['user']),
                               db: Session = Depends(get_db)) -> models.User:
    """Resolve the currently authorized User."""
    db_user = crud.get_user(db, username=username)
    if not db_user:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
                            detail=f'The owner of this JSON Web Token no longer exists')

    return db_user
```

Figur 155. Bruk av authorize_user hvor scope er spesifisert

For å sikre endepunkt bak autorisering bruker vi *resolve_current_user* videre. Ettersom den har *authorize_user* som dependency, vil da automatisk alle endepunkt som bruker *resolve_current_user* også kreve samme dependency. Dermed blir disse endepunktene testet for autorisering, og i tillegg vil brukerobjektet som hører til token bli lastet inn fra databasen og sendt med inn i funksjonen.

```
@router.get('/me', response_model=User,
            responses={status.HTTP_400_BAD_REQUEST: {'model': ErrorResponse},
                       status.HTTP_401_UNAUTHORIZED: {'model': ErrorResponse}})
async def get_current_user(current_user: models.User = Depends(resolve_current_user)):
    """Get the currently authorized user."""
    return current_user
```

Figur 156. Bruk av dependency resolve_current_user vil automatisk kreve autorisering

4.8.1.2 Spring Boot implementasjon

Til forskjell fra FastAPI som enkelt gir oss noen få skjema for å bygge et eget autentiseringssystem med passord, så må både autentisering og autorisering i Spring Boot gå gjennom et ferdiglaget system. Denne må konfigureres slik man ønsker, og man får da til slutt en løsning som følger disse standardene uavhengig av konfigurasjonen. For denne løsningen brukte vi et GitHub repo som referanse (Zelei, 2019).

Ulempen med en slik løsning er at man ender opp med mange grensesnitt som må kobles opp mot egen kode og data. Det kan bli forvirrende ved hvor mange koblinger som må lages blant de ulike komponentene. Vår egen løsning endte opp med 5 ulike klasser for autentisering og autorisering, og det er vanskelig å holde orden på hva som gjør hva.

Alle konfigurasjonene til påloggingsflyten er definert i *security* pakken. Her starter vi med *WebSecurityConfig*. Denne klassen sier til Spring at vi ønsker å ta i bruk Spring Security. Denne klassen trenger ikke å refereres noen andre steder siden vi bruker Bean annotations. Det vil si at Spring finner disse konfigurasjonene selv og tar de i bruk automatisk.

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
```

Figur 157. Definisjon av hovedkonfigurasjonen for Spring Security

For å sette opp autentisering må vi først koble opp vår egen *UserDetailsService*. Denne klassen vil fortelle Spring hvordan en bruker skal hentes fra databasen og bli verifisert. Merk at her krever Spring Security at brukerimplementasjonen har *authorities*. Dette er Spring sin måte å håndtere roller på. Ettersom vi ikke har noen nytte av disse lager vi bare en ny authority hver gang en bruker blir lastet inn av Spring Security.

```
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    me.kverna.spring.repost.data.User user = userRepository.findByUsername(username);
    if (user == null) {
        throw new UsernameNotFoundException("User " + username + " not found");
    }

    Set<GrantedAuthority> authorities = new HashSet<>();
    authorities.add(new SimpleGrantedAuthority( role: "USER"));

    return new User(user.getUsername(), user.getHashedPassword(), authorities);
}
```

Figur 158. Metode for å koble opp UserDetailsService mot vår database

I tillegg trenger vi en måte å verifisere at passordet til brukeren er riktig under innlogging. I Spring må dette implementeres i en *AbstractUserDetailsAuthenticationProvider*. Her kan vi bruke BCrypt algoritmen til å verifisere passordet. Dersom passordet er blankt eller ikke stemmer kastes en exception her, som sier til Spring at passordet er ugyldig.

```
@Override
protected void additionalAuthenticationChecks(UserDetails userDetails, UsernamePasswordAuthenticationToken token)
        throws AuthenticationException {
    if (token.getCredentials() == null) {
        throw new BadCredentialsException("Invalid login information");
    }

    String password = (String) token.getCredentials();
    if (!passwordEncoder.matches(password, userDetails.getPassword())) {
        throw new BadCredentialsException("Invalid login information");
    }
}
```

Figur 159. Metode for verifisering av passordet til brukeren

Denne klassen må også kobles opp mot *UserDetailsService* klassen vi laget tidligere for å kunne hente brukerobjektet fra databasen, selv om begge klassene er inkludert i konfigurasjonen i *WebSecurityConfig* klassen.

```
@Override
protected UserDetails retrieveUser(String username, UsernamePasswordAuthenticationToken token)
        throws AuthenticationException {
    return userDetailsService.loadUserByUsername(username);
}
```

Figur 160. Kobling til samme metode for å hente bruker i UserDetailsService

Disse klassene må brukes i *WebSecurityConfig* for å fortelle spring hvordan autentisering skal sjekkes.

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(userDetailsService);
    auth.authenticationProvider(userDetailsAuthenticationProvider);
}
```

Figur 161. Kobling mot UserDetailsService og UserDetailsAuthenticationProvider i hovedkonfigurasjonen

Til slutt, for å slå sammen denne typen autentisering med OAuth2 password flow må vi sette opp en ressursserver. Dette kan vi gjøre ved å lage en *ResourceServerConfigurerAdapter*.

```
@Configuration
@EnableResourceServer
public class ResourceServerConfiguration extends ResourceServerConfigurerAdapter {
```

Figur 162. Definisjon av ressursserveren

Her må vi også lage en *JwtTokenStore*. På den måten kan ressursserveren lage JWT som blir sendt som svar i token endpoint. I ressursserveren setter vi også opp hovedkonfigurasjonen for sikkerhet, hvor vi velger å tillate alle forespørsler. Dette gjør vi slik at krav om autorisering kan bli kontrollert videre i kontrollerene for de endepunktene som krever det. Da får vi også støtte for å sjekke OAuth2 scopes for hvert endepunkt.

```
@Override
public void configure(ResourceServerSecurityConfigurer resources) {
    resources.tokenStore(new JwtTokenStore(jwtAccessTokenConverter));
}

@Override
public void configure(HttpSecurity http) throws Exception {
    http
            .cors().and() HttpSecurity
            .csrf().disable() HttpSecurity
            .authorizeRequests() ExpressionInterceptUrlRegistry
            .antMatchers( ...antPatterns: "/**").permitAll() ExpressionUrlAuth
            .anyRequest() ExpressionUrlAuthorizationConfigurer<HttpSecurity>.Auth
            .authenticated();
}
```

Figur 163. Generelle innstillinger for JWT og krav om autorisering

For å koble opp autorisering må vi lage en *AuthorizationServerConfigurerAdapter*. Her konfigurerer vi en OAuth2 klient med password flow. Vi setter også opp client etter konfigurasjonsfila, og scopes.

```
@Override
public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
    clients
            .inMemory() InMemoryClientDetailsServiceBuilder
            .withClient(config.getClientId()) ClientDetailsServiceBuilder<InMemoryClientDe
            .secret(null) ClientDetailsServiceBuilder<InMemoryClientDetailsServiceBuilder>.ClientB
            .authorizedGrantTypes("password") ClientDetailsServiceBuilder<InMemoryClientDe
            .scopes("user");
}
```

Figur 164. Definisjon av klienten i OAuth2

Vi satt secret til *null* for å tillate autentisering uten *client_secret* i OAuth2 skjemaet. Merk at klientinfo ikke er nødvendig etter autentisering, som vil si at denne delen er egentlig oppsettet for autentisering. Denne delen er derfor ganske forvirrende i Spring. Samtidig må vi kjøre en metode *allowFormAuthenticationForClients()* dersom vi ønsker at klientinfo skal kunne skrives i form dataen slik som i FastAPI prosjektet. Spring sjekker også at *client_id* stemmer, så dette slipper vi å gjøre manuelt til forskjell fra FastAPI prosjektet.

Vi må også koble opp JWT her slik at autoriseringen vet hvordan den kan håndtere tokens sendt i en forespørsel. Etter oppsett tar Spring av seg resten av JWT dekoding og verifisering. Altså trenger vi ikke å gjøre noen manuelle sjekker om token er utløpt og om scopes er satt. I samme konfigurasjon sier vi også til Spring at vi ønsker å endre sti til OAuth2 token.

```
@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints) {
    endpoints
            .pathMapping( defaultPath: "/oauth/token",  customPath: "/api/auth/token")
            .tokenStore(new JwtTokenStore(jwtAccessTokenConverter))
            .authenticationManager(authenticationManager)
            .accessTokenConverter(jwtAccessTokenConverter);
}
```

Figur 165. Oppsett av JWT og endring av sti til OAuth2 token

For å legge til autorisering på endepunktene som krever det kan vi nå bruke *@PreAuthorize* annotasjonen i Spring. Da kjører Spring Security sjekkene som vi har definert for vår autorisering. Vi valgte å lage vår egen annotasjon *@AuthorizeUser*, som også inkluderer at *user* scope må være inkludert i JWT som blir sendt i forespørselen. Sjekking av scope er altså støttet per endepunkt i Spring, og vi slipper å implementere det selv.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})
@PreAuthorize("#oauth2.hasScope('user')")
public @interface AuthorizeUser {
}
```

Figur 166. Bruk av @PreAuthorize i en egen annotasjon for å begrense endepunkt til kun user scope

Endepunkt som krever autorisering er annotert med *@AuthorizeUser*.

```
@AuthorizeUser
@GetMapping(value = "/me", produces = {"application/json"})
public User getCurrentUser(@CurrentUser User currentUser) {
    return currentUser;
}
```

Figur 167. Bruk av @AuthorizeUser annotasjonen for å kreve autorisering på et endepunkt

@CurrentUser annotasjonen som er brukt før et parameter av typen *User* i endepunktet blir brukt for å hente brukerobjektet som tilhører brukeren som har laget forespørselen. Måten dette blir gjort er at vi henter ut hvilken bruker token hører til og henter denne brukeren fra databasen.

4.8.1.3 ASP.NET implementasjon

All konfigurasjonen for autentisering og autorisering er definert i *Startup.cs*. Her bruker vi IdentityServer4 som autorisering- og autentiseringsserver. ASP.NET er altså det eneste rammeverket vi har her hvor vi bruker et tredjepartsbibliotek for OAuth2 autentisering. Det er også verdt å bemerke at IdentityServer4 bruker OpenID Connect i stedet for vanlig OAuth2, men siden OpenID Connect bygger på OAuth2 får vi den spesifikasjonen vi trenger.

IdentityServer4 settes opp i *ConfigureService* metoden. Her legger vi også til API ressurser og OAuth2 klienter.

```
// Setup authorization and authentication server with a resource owner validator for
// authorizing database users
var builder = services.AddIdentityServer()
    .AddInMemoryApiResources(_config.Apis)
    .AddInMemoryClients(_config.Clients)
```

Figur 168. Oppsett av IdentityServer4 med API ressurser og klienter fra konfigurasjonsfilen

API ressurs i *IdentityServer4* er en enkel måte å definere sammenhenger mellom scopes og ressurser på serveren. Vi lager kun én ressurs i henhold til *user* scope som er i modellen vår. Denne definerer vi i *Config* fila.

```
public IEnumerable<ApiResource> Apis => new[]
{
    // The name of the resource will also be the name of the scope
    new ApiResource("user", "User")
};
```

Figur 169. Konfigurasjon for en API ressurs med user scope

OAuth2 klienten er også definert i *Config* fila slik at *client_id* kan defineres gjennom konfigurasjon. Her definerer vi klienten til å ikke bruke secret, og med OAuth2 password flow. I tillegg legger vi til *user* scope her.

```
public IEnumerable<Client> Clients => new[]
{
    new Client
    {
        ClientId = Configuration["ClientId"] ?? Configuration["CLIENT_ID"] ?? "repost",
        RequireClientSecret = false,
        AllowedGrantTypes = GrantTypes.ResourceOwnerPassword,
        AllowedScopes = {"user"}
    }
};
```

Figur 170. Konfigurasjon av klienten og scopes som kreves

Vi setter også opp konfigurasjon av en signing credential. Denne brukes for å hashe JWT signaturen og kan sammenliknes med secret i de andre rammeverkene. En forskjell her er at secret ikke er støttet, så man må bruke asymmetriske nøkler for signering av JWT med IdentityServer4.

```
// Load signing key from configuration or default to developer signing key if configuration is missing
var signingCredential :X509Certificate2 = _config.SigningCredential;
if (signingCredential != null)
{
    builder.AddSigningCredential(signingCredential);
}
else
{
    _log.Add((LogLevel.Warning, "Defaulting to temporary signing credential"));
    builder.AddDeveloperSigningCredential();
}
```

Figur 171. Konfigurasjon av JWT nøkler

For å verifisere brukernavn og passord gitt i OAuth2 skjemaet må vi legge til en *ResourceOwnerPasswordValidator* klassen som er beskrevet i Passord (4.7.3.5).

Vi ønsker å bruke */auth/token* som endepunkt for å logge inn i ASP.NET som i de andre rammeverkene. Siden IdentityServer4 bruker OpenID Connect er standarder for alle endepunktene satt fast. Siden vi ikke skal bruke OpenID Connect har vi ikke noen krav om standarder på disse endepunktene, så vi valgte å lete etter mulige måter å endre endepunktet på manuelt. Løsningen er hentet fra Stackoverflow bruker Rikki (Rikki, 2018) og er vist i Figur 172, hvor vi manuelt finner alle endepunkt som er definert i tjenestene og erstatter alle */connect* endepunkt med */api/auth*. Dette fungerer ettersom OpenID Connect token endepunktet ligger på */connect/token*.

```
// Hack to reroute the OpenID endpoint to /api/auth
// We only require the /token endpoint for our OAuth2 setup, so this hack adjusts the endpoint to
// match the other implementations
builder.Services // IServiceCollection
    .Where(descriptor => descriptor.ServiceType == typeof(Endpoint)) // IEnumerable<ServiceDescriptor>
    .Select(item :ServiceDescriptor => (Endpoint) item.ImplementationInstance) // IEnumerable<Endpoint>
    .ToList() // List<Endpoint>
    .ForEach(item :Endpoint => item.Path = item.Path.Value.Replace( oldValue: "/connect", newValue: "/api/auth"));
```

Figur 172. Endring av OpenID Connect sti slik at token stien blir /api/auth/token

For å kunne knytte opp *IdentityServer4* med ASP.NET applikasjonen må vi definere en *Authentication* i tillegg. Denne bestemmer hvordan påloggingsinformasjon som autentiseringstoken blir sendt i forespørslene til API-et. Her har vi definert en enkel *JwtBearer* for JWT autentisering. Innholdet av JWT er altså overlatt til ASP.NET, og vi trenger ikke å verifisere tokens manuelt.

```
// Add internal JSON Web Token validation (issuing tokens is done by IdentityServer4 defined above)
services.AddAuthentication( defaultScheme: "Bearer")
    .AddJwtBearer( authenticationScheme: "Bearer",  configureOptions: options =>
    {
        options.Authority = "http://localhost:8002";
        options.RequireHttpsMetadata = false;
        options.Audience = "user";
        options.TokenValidationParameters.ValidateIssuer = false;
    });
```

Figur 173. Oppsett av JWT bearer for autorisering

Til slutt må konfigurasjonene slås på i *Configure* metoden i *Startup.cs*. Da slår vi altså på *IdentityServer4* og *Authorization* delene.

```
app.UseIdentityServer();
app.UseAuthorization();
```

Figur 174. Konfigurasjonene må bli skrudd på i Configure metoden

Nå kan vi lett markere at et endepunkt krever autorisering ved å bruke *[Authorize]* attributten.

```
/// <summary>Get Current User</summary>
/// <remarks>Get the currently authorized user.</remarks>
[HttpGet]
[Route( template: "me")]
[Authorize]
public User GetCurrentUser()
{
    return GetAuthorizedUser();
}
```

Figur 175. Bruk av [Authorize] for å autorisere et endepunkt

4.8.2 Oppdatere objekter med PATCH endepunkt

Grunnen til at vi bruker PATCH endepunkt for å oppdatere objekter er fordi vi da kan velge hvilke felt vi vil oppdatere. Om vi tar utgangspunkt i en bruker så vil vi kunne oppdatere feltene *bio* og *avatar_url*, men vi vil også ha mulighet til å oppdatere bare ett av feltene om vi ønsker det. Dermed må vi kunne sende en forespørsel der vi gir en ny verdi til feltet *bio* uten å påvirke feltet *avatar_url*. Vi må også ha mulighet til å kunne sette et felt til *null*, om feltet tillater dette.

For å oppnå et endepunkt med slik funksjonalitet skal det altså være mulig å oppdatere et felts verdi uten å påvirke det andre. Dette kan gjøres ved å sette standardverdien til *null* eller *None*, slik at kun felt som har en gyldig verdi blir endret. Men i vårt tilfelle trenger vi at noen felt skal kunne settes til *null*. Derfor krever vi at rammeverkene har støtte for informasjon om hvorvidt et felt er inkludert eller ikke i JSON skjemaet.

4.8.2.1 FastAPI implementasjon

Først må vi ha en egen modell, eller API skjema, for å oppdatere en bruker. Denne modellen skal brukes i PATCH endepunktet og vi kaller den *EditUser*. Modellen inneholder kun de feltene som skal kunne endres, og vi ser at de er av typen *Optional* slik at Pydantic vet at verdien kan også være *None*. Vi må også sette standardverdi til *None*, som forteller Pydantic at feltet kan utelukkes. Det er i disse tilfellene at feltet ikke blir inkludert i et JSON skjema at vi ønsker å ikke gjøre noen endring på feltet i databasen.

```
class EditUser(BaseModel):
    """Schema for editing a user account"""
    bio: Optional[str] = None
    avatar_url: Optional[str] = None
```

Figur 176. Pydantic modell for oppdatering av en bruker der felt er valgfrie

Vi må ha et endepunkt i bruker kontrolleren som tar PATCH forespørsler. Dette endepunktet finner den innloggede brukeren ved hjelp av autoriserings token som brukeren sender med i forespørselen. I tillegg tar vi inn *EditUser* som data i forespørselen. Videre bruker vi en CRUD operasjon *update_user* som tar inn brukernavnet til den innloggede brukeren og de feltene som er satt i *EditUser* objektet.

Etter vi har fått *EditUser* skjemaet etter et PATCH kall må vi bruke parameteret *exclude_unset* i Pydantic for å få et objekt som består kun av de feltene som ble satt i forespørselen. Pydantic holder altså orden på hvilke felt som er inkludert i forespørselen, og kan ved bruk av dette parameteret kun inkludere de feltene som er med.

```
@router.patch('/me', response_model=User,
               responses={status.HTTP_400_BAD_REQUEST: {'model': ErrorResponse},
                          status.HTTP_401_UNAUTHORIZED: {'model': ErrorResponse}})
async def edit_current_user(*, current_user: models.User = Depends(get_current_user), edited_user: EditUser,
                            db: Session = Depends(get_db)):
    """Edit the currently authorized user."""
    return crud.update_user(db, username=current_user.username, **edited_user.dict(exclude_unset=True))
```

Figur 177. PATCH endepunkt for å oppdatere gjeldende bruker. Når vi bruker CRUD metoden blir kun felt med verdi tatt inn på grunn av exclude_unset=True parameteret i omgjøringen av EditUser objektet til dict

Videre i CRUD metoden oppdaterer vi de feltene som har fått ny verdi, og returner den oppdaterte brukeren.

```
def update_user(db: Session, *, username: str, **columns: Any) -> User:
    """Edit the user with the given username.

    Enter any `repost.models.User` column to update in `**columns`.
    """
    db.query(User).filter_by(username=username).update(columns)
    db.commit()

    return get_user(db, username=username)
```

Figur 178. CRUD metode for å oppdatere spesifiserte felt i en bruker

4.8.2.2 Spring Boot implementasjon

Her lager vi også en egen modell for *EditUser* som inneholder kun de feltene som skal kunne oppdateres. Til forskjell fra FastAPI har ikke Spring sin JSON serialisering støtte for å se hvilke felt som er inkludert i JSON skjemaet før de konverteres til *EditUser* objektet. Her blir de heller satt til *null* om de ikke er inkludert i skjemaet. Dette holder ikke til vårt bruk, ettersom de fleste av objektene våre skal kunne bli satt til verdien *null*. Dermed måtte vi finne en annen løsning.

For å oppnå dette brukte vi typen *Optional*<> på feltene i modellen. *Optional*<> i Java er egnet til å tillate et objekt å være *null* uten å skape problemer ved å gi metodene *isEmpty* og *isPresent*, som skal brukes til å teste om objektet er *null* eller ikke. For vårt bruksområde vil hele feltet bli satt til *null* om et felt ikke er inkludert. Men dersom feltet er inkludert og også satt til *null*, så blir feltet et gyldig *Optional*<> objekt som består av verdien *null*. På den måten har vi altså to forskjellige verdier basert på om feltet er satt eller ikke, og også om det blir satt til *null*.

```
@Data
@NoArgsConstructor
public class EditUser {
    private Optional<String> bio;

    @JsonProperty(value = "avatar_url")
    private Optional<String> avatarUrl;
}
```

Figur 179. Datamodell for å oppdatering av en bruker der felt er valgfrie

I kontrolleren har vi satt opp et endepunkt som tar imot PATCH forespørsler. Det som er viktig å merke seg er at her må vi spesifisere content type som *application/patch+json*. Grunnen til dette er at Spring Boot ikke godtar *application/json* slik som FastAPI gjør i PATCH forespørsler. Om vi skal bruke PATCH endepunkt må vi dermed bruke riktig *content type* i alle forespørsler til disse endepunktene.

Dette endepunktet finner brukeren som gjør forespørselen basert på autentiserings token som blir sendt med, slik vi gjør i FastAPI. Vi tar inn *EditUser* som data fra forespørselen. Vi bruker *UserService* metoden *editUser* der vi sender inn *editUser* objektet og *currentUser* objektet.

```
@Operation(
        summary = "Edit Current User", description = "Edit the currently authorized user.",
        responses = {
                @ApiResponse(responseCode = "200", description = "Successful Response"),
                @ApiResponse(responseCode = "400", description = "Bad Request"),
                @ApiResponse(responseCode = "401", description = "Unauthorized")
        },
        security = @SecurityRequirement(name = "OAuth2PasswordBearer", scopes = "user")
)
@AuthorizeUser
@PatchMapping(value = "/me", consumes = {"application/patch+json"}, produces = {"application/json"})
public User editCurrentUser(@RequestBody EditUser editUser, @CurrentUser User currentUser) {
    return service.editUser(editUser, currentUser);
}
```

Figur 180. PATCH endepunkt for å oppdatere gjeldende bruker. editUser og currentUser blir sendt som parametre til en servicemetode

I *UserService* sjekker vi først om feltene er *null*. Det er altså testen vi må gjøre for å se om feltet er inkludert i den sendte forespørselen. Deretter må vi også ha en sjekk på om feltets *isEmpty* metode er *true*. I dette tilfellet ble feltet inkludert i forespørselen, men også eksplisitt satt til *null*. Slik får vi da også i Spring oppnådd et PATCH endepunkt hvor felter som ikke skal endres ikke blir overskrevet med *null* verdien som vanligvis ville ha vært “utelukket verdi” i Jackson.

```java
/**
 * Edit the user.
 *
 * @param editUser the edited user fields.
 * @param user     the user to edit.
 * @return the edited user.
 */
public User editUser(EditUser editUser, User user) {
    if (editUser.getBio() != null) {
        user.setBio(editUser.getBio().isEmpty() ? null : editUser.getBio().get());
    }
    if (editUser.getAvatarUrl() != null) {
        user.setAvatarUrl(editUser.getAvatarUrl().isEmpty() ? null : editUser.getAvatarUrl().get());
    }

    user.setEdited(LocalDateTime.now());
    return repository.save(user);
}
```

Figur 181. Servicemetode for å oppdatere en bruker. Her sjekkes det om feltet har en verdi og setter brukerfeltet til den nye verdien

4.8.2.3 ASP.NET implementasjon

ASP.NET har heller ikke noen innebygd støtte for å vite hvilke felt som er inkludert i en forespørsel av et JSON skjema. Men i ASP.NET kan vi løse dette på en litt mer elegant måte enn i Spring på grunn av måten JSON konverteringen er håndtert. Når ASP.NET konverterer skjemaet til en modell, settes alle feltene som er inkludert med *set* metoden til feltet. Det vil si at felt som ikke er satt i JSON skjemaet ikke settes med *set* metodekallet. Dette kan vi bruke til fordel ved å holde styr på hvilke objekter som blir satt med *set* metoden til feltet.

Som en gjenbrukbar løsning laget vi en klasse som heter *EditModel*. Denne klassen kan arves fra når vi lager en ny modell til Edit. *EditModel* har oversikt over alle felt som hører til objektet, og som også skal kunne være *null*. Den har egne metoder for å sette eller hente verdien fra denne klassen, og en metode for å sjekke om et felt ble inkludert i JSON skjemaet.

```
4 usages   4 inheritors   Per Christian Kvalvik
public class EditModel
{
    private readonly Dictionary<string, object> _fields = new Dictionary<string, object>();

    6 usages   Per Christian Kvalvik
    public bool IsFieldSet(string field)
    {
        return _fields.ContainsKey(field);
    }

    6 usages   Per Christian Kvalvik
    public object GetField(string field)
    {
        return _fields[field];
    }

    6 usages   Per Christian Kvalvik
    protected void SetField(string field, object value)
    {
        _fields.Add(field, value);
    }
}
```

Figur 182. EditModel med metoder for å sjekke, hente og sette verdier på feltene til ressurser som skal oppdateres

Når vi skal arve fra *EditModel* må alle feltene bruke *get* og *set* metodene som er definert i *EditModel*, slik at modellen kan få informasjon om at *set* metoden til et felt er brukt. Som nevnt tidligere vil *set* metoden kun bli kjørt når feltet er inkludert i JSON skjemaet.

```
public class EditUser : EditModel
{
    4 usages   Per Christian Kvalvik
    public string Bio
    {
        get => (string) GetField(nameof(Bio));
        set => SetField(nameof(Bio), value);
    }

    [JsonPropertyName("avatar_url")]
    4 usages   Per Christian Kvalvik
    public string AvatarUrl
    {
        get => (string) GetField(nameof(AvatarUrl));
        set => SetField(nameof(AvatarUrl), value);
    }
}
```

Figur 183. EditUser som bygger på EditModel for oppdatering av brukere

I endepunktet definert i kontrolleren kan vi nå bruke *IsFieldSet* metoden for å først sjekke om feltet skal endres, og så kan vi få tilgang til verdien på vanlig vis. I koden under sjekker vi først om *Bio* feltet er satt, før vi endrer verdien til database modellen. Da kan også feltene være *null* som ønsket.

```
/// <summary>Edit Current User</summary>
/// <remarks>Edit the currently authorized user.</remarks>
[HttpPatch]
[Route( template: "me")]
[Authorize]
Per Christian Kvalvik
public User EditCurrentUser(EditUser editUser)
{
    var user = GetAuthorizedUser();

    if (editUser.IsFieldSet(nameof(editUser.Bio)))
        user.Bio = editUser.Bio;
    if (editUser.IsFieldSet(nameof(editUser.AvatarUrl)))
        user.AvatarUrl = editUser.AvatarUrl;
    user.Edited = DateTime.UtcNow;

    Db.SaveChanges();
    return user;
}
```

Figur 184. EditUser modellen brukes for å oppdatere brukeren i PATCH endepunktet for oppdatering av bruker

Det blir litt ekstra kode i forhold til FastAPI, og bruken ligner på det vi gjør i Spring Boot, men det virker som en ganske elegant løsning. Bruken av arv gjør det enkelt å gjenbruke koden, noe som er veldig positivt.

4.8.3 Integrere OpenAPI dokumentasjon

For at det skal være enkelt for andre å bruke vår API ønsker vi å ha en god og oversiktlig dokumentasjon som viser alle endepunkter med relevant informasjon. Vi vil dokumentere våre endepunkter slik at det blir generert en OpenAPI dokumentasjon. Ved å bruke OpenAPI standarden har vi mulighet til å bruke mange forskjellige verktøy som bygger på denne standarden. Vi har valgt å bruke Swagger UI for å visualisere dokumentasjonen i en interaktiv nettside. Med Swagger UI er det mulig å prøve de forskjellige endepunktene og se hva som er forventet svar, og hva som er et faktisk svar og så videre. Her kan man også autentisere seg for å bruke endepunkter som ikke er åpne.

Figur 185. Liste over resub endepunktet i Swagger UI

Her ser vi en samling av endepunkter for resuber. Hver ruter har en tag slik at endepunktene til den ruteren blir gruppert i dokumentasjonen. Den informasjonen vi får her er hvilken type forespørsel som må benyttes på hvilken sti, med en kort beskrivelse av hva operasjonen er. I tillegg ser vi et låsikon på noen endepunkter, dette indikerer at vi må være autentisert for å få tilgang til disse endepunktene.

For å kunne se mer informasjon om et enkelt endepunkt kan vi klippe på feltet for å utvide det. Den første ekstra informasjonen vi får her er en lengre beskrivelse av operasjonen. Videre har vi parametre, her har vi bare brukt sti parametere, og deretter har vi data som skal sendes med. Her er det API skjema modellen som bestemmer hvordan dataene skal se ut, altså hvilke felt som skal brukes. Her får vi også media typen som skal brukes i forespørselen, hvor alle endepunkt utenom autentiserings token bruker JSON som media type.

Neste del er alle mulige responser vi kan få fra dette endepunktet. Hver respons viser statuskodene, dataene og media typen som blir returnert.

Det er også mulig å bruke Swagger UI for å kjøre forespørsler direkte. Om vi trykker Try it out kan vi skrive i feltene for parametre og data, og ved å trykke *Execute* kjøres forespørselen og vi får et svar fra APIen. I tillegg så vises curl kommandoen som blir generert for å kjøre forespørselen slik at man kan bruke den fra en konsoll lokalt om man ønsker det.

users

POST /api/users/ Create User

Create a new user.

Parameters Try it out

No parameters

Request body required application/json

Example Value | Schema

```
{
  "username": "string",
  "password": "string"
}
```

Responses

Code	Description	Links
201	Successful Response Media type application/json Controls Accept header. Example Value | Schema	*No links*

```
{
  "username": "string",
```

```
  "bio": "string",
  "avatar_url": "string",
  "created": "2020-05-14T07:34:02.103Z",
  "edited": "2020-05-14T07:34:02.103Z"
}
```

400 Bad Request — No links

Media type

application/json

Example Value | Schema

```
{
  "detail": "string"
}
```

422 Validation Error — No links

Media type

application/json

Example Value | Schema

```
{
  "detail": [
    {
      "loc": [
        "string"
      ],
      "msg": "string",
      "type": "string"
    }
  ]
}
```

Figur 186. Full visning med parametre og responser av POST endepunkt i kontrolleren for brukere

4.8.3.1 FastAPI implementasjon

FastAPI har støtte for automatisk generasjon av OpenAPI dokumentasjon og den har både Swagger UI og ReDoc innebygget. Dette betyr at vi ikke trenger å gjøre noe konfigurasjon for at

dette skal fungere. Det eneste vi gjorde var å sette egendefinert sti slik at det ble liggende under */api* på serveren.

For å legge til ekstra informasjon på toppen av dokumentasjonen lager vi en docstring for main filen vår. I denne filen har vi også spesifisert hvilken sti vi ønsker at Swagger UI og ReDoc skal være tilgjengelig på.

```
"""Repost API written in FastAPI.

[View source code on GitHub](https://github.com/pckv/repost-fastapi)

Authors: pckv, EspenK, jonsondrem
"""
```

Figur 187. Docstring i main.py som blir lagt inn som tekst i Swagger UI

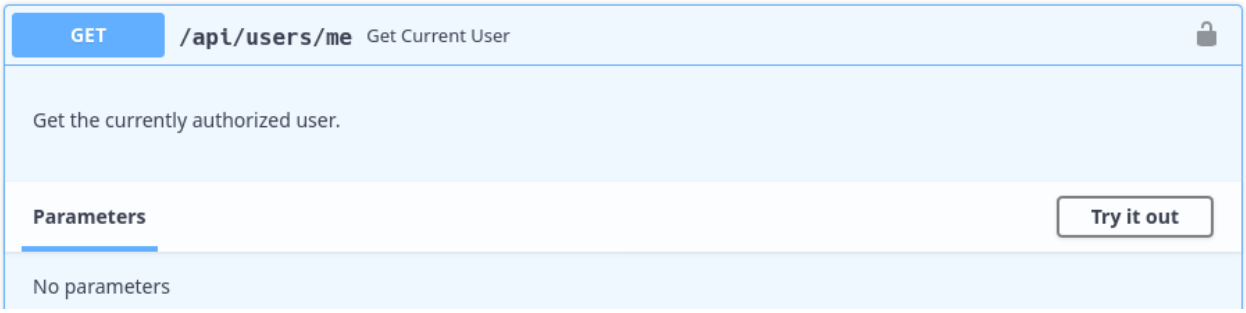
```
app = FastAPI(title='Repost', version=__version__, description=__doc__,
              docs_url='/api/swagger', redoc_url='/api/docs')
```

Figur 188. Konfigurasjon av Swagger UI i instansiering av FastAPI

Når vi lager et endepunkt i FastAPI blir funksjons-navnet gjort om til en kort beskrivelse, så funksjonen *get_current_user* får kort beskrivelse *Get Current User*. Ekstra beskrivelse blir hentet fra docstring i funksjonen. Her ser vi at stien er */me*, men i dokumentasjonen er den */api/users/me*, det er fordi alle endepunktene for brukere ligger under */api/users* og er bestemt av ruteren. Videre har vi med de mulige responsene som blir vist lenger nede i dokumentasjonen. Responsmodellen til de alternative responsene er et enkelt skjema med bare ett felt for melding som beskriver kort hva som er feil.

```
@router.get('/me', response_model=User,
            responses={status.HTTP_400_BAD_REQUEST: {'model': ErrorResponse},
                       status.HTTP_401_UNAUTHORIZED: {'model': ErrorResponse}})
async def get_current_user(current_user: models.User = Depends(resolve_current_user)):
    """Get the currently authorized user."""
    return current_user
```

Figur 189. Endepunkt med alternative responser og docstring for å beskrive endepunktet



Figur 190. Samme endepunkt i Swagger UI

Siden dette endepunktet ikke har noen parametre blir det ingen felt synlig i dokumentasjonen heller, men om vi hadde med enter en variabel i stien eller vi trenger forespørsel-data hadde dette blitt dokumentert. Når vi skal ha data i endepunktet bruker vi et skjema for å beskrive hvilken data vi vil ha i *response_model* parameteret.

4.8.3.2 Spring Boot implementasjon

For å kunne generere samme dokumentasjon for Spring Boot måtte vil legge til et tredjepart bibliotek som heter springdoc-ui. Denne inneholder verktøy for å generere dokumentasjon og for å vise fram dokumentasjonen med Swagger UI. Siden det ikke er bygget direkte inn i Spring Boot blir det noe mer skriving for å oppnå samme resultat.

Etter at vi la til springdoc-ui avhengigheten var det lite annet som måtte gjøres før vi kunne dokumentere koden. I application.properties satte vi sti for hvor spesifikasjonsfilen blir tilgjengelig og hvor Swagger UI blir tilgjengelig. For å kunne autentisere med Swagger UI måtte vi sette opp en OpenAPI konfigurasjon som bestemte skjema for autentisering. Etter det er gjort blir alle mappinger lagt til automatisk, og videre legger vi til ekstra informasjon.

```
@SecurityScheme(
        name = "OAuth2PasswordBearer",
        type = SecuritySchemeType.OAUTH2,
        in = SecuritySchemeIn.HEADER,
        bearerFormat = "jwt",
        flows = @OAuthFlows(
                password = @OAuthFlow(
                        tokenUrl = "/api/auth/token",
                        scopes = {
                                @OAuthScope(name = "user", description = "User access")
                        }
                )
        )
)
public class OpenApiConfig {

}
```

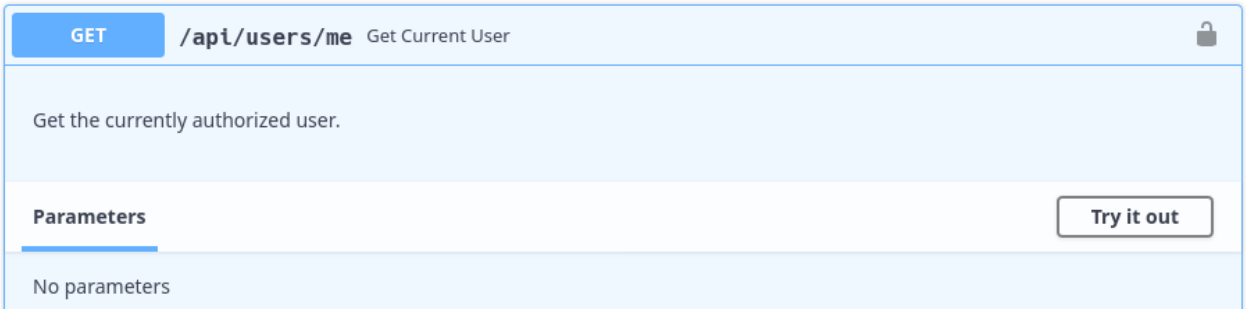
Figur 191. Konfigurasjon av sikkerhetsskjema for OpenAPI

Når vi annoterer en metode med en type *Mapping* blir den automatisk lagt til som et endepunkt. Informasjonen vi får med er stien til endepunktet, hvilken media type som skal brukes og hvilken modell eller skjema som returneres. For videre informasjon trenger vi *Operation* annotasjonen som gir oss en kort beskrivelse og en beskrivelse av endepunktet. Disse to feltene er hentet fra henholdsvis funksjons-navnet og docstringen i FastAPI, men det fungerer ikke slik i Spring Boot, så vi må manuelt skrive inn teksten. Derfor valgte vi å ikke skrive JavaDoc for kontrolleren da vi allerede dokumenterer funksjonaliteten i *Operation* annotasjonen. Videre legger vi med mulige

responser på nesten samme måte som i FastAPI. Til slutt legger vi med et felt for sikkerhet som forteller hvilken autorisering som trengs for å bruke endepunktet.

```
@Operation(
        summary = "Get Current User", description = "Get the authorized user.",
        responses = {
                @ApiResponse(responseCode = "200", description = "Successful Response"),
                @ApiResponse(responseCode = "400", description = "Bad Request"),
                @ApiResponse(responseCode = "401", description = "Unauthorized")
        },
        security = @SecurityRequirement(name = "OAuth2PasswordBearer", scopes = "user")
)
@AuthorizeUser
@GetMapping(value = "/me", produces = {"application/json"})
public User getCurrentUser(@CurrentUser User currentUser) {
    return currentUser;
}
```

Figur 192. @Operation annotasjon inneholder ekstra informasjon som skal vises i Swagger UI, for felt som krever innlogging må vi legge til security med type og scope på autentisering



Figur 193. Samme endepunkt i Swagger UI

I dette eksempelet er det ingen data som skal bli sendt til APIen, men om det er det blir også dette dokumentert. Om vi har en sti-variabel blir denne automatisk dokumentert, og om vi trenger data i forespørselen blir feltene automatisk dokumentert siden vi bruker et skjema for å beskrive hvilken data vi trenger.

En konflikt som oppsto når vi brukte springdoc var at vi ville bruke resolvere for å skrive mindre duplisert kode. Som et eksempel trenger vi å finne en post basert på ID-en til posten vi får som sti-parameter flere plasser. Vi tenkte det ville være fornuftig å bruke en resolver som finner posten basert på ID-en og automatisk putte objektet vi er ute etter inn i funksjonene som blir kalt.

Funksjonene som krever ID-en til en post kan da defineres slik:

```
@GetMapping(value = "/{postId}", produces = {"application/json"})
public Post getPost(Post post) {
    return post;
}
```

Figur 194. Alternativ måte å hente finne ressurser med resolver der service blir brukt i bakgrunnen

Problemet med dette ble at vi må legge inn et parameter i metoden som er av typen Post i stedet for å ha et parameter for sti-parameteret. Dette ble da med i OpenAPI dokumentasjonen som et parameter av typen vi skulle finne. For å fjerne dette kunne vi ha lagt til en annotering der vi velger å ignorere dette parameteret. Etter en diskusjon ble vi enige om at vi kom ikke til å spare noe særlig på å bruke en resolver når vi må bruke mer kode på å ignorere det i springdoc enn vi trenger kode for å manuelt hente objektet via en service. Dermed bruker vi heller bare service metoden for å få posten basert på ID:

```
@GetMapping(value = "/{postId}", produces = {"application/json"})
public Post getPost(@PathVariable int postId) {
    return service.getPost(postId);
}
```

Figur 195. Endepunkt slik vi endte opp med å hente ressurser uten bruk av resolver

4.8.3.3 ASP.NET implementasjon

For at det skal genereres OpenAPI dokumentasjon må vi legge til SwaggerGen som en service i Startup filen i prosjektet. Her setter vi metadata som tittel, versjon og beskrivelse. SwaggerGen tillater flere dokumenter, og dermed må vi også sette et navn på dokumentet, her kalt *openapi*.

```
services.AddSwaggerGen(options =>
{
    // Add main OpenAPI document
    // The name of the document is openapi, in order to name the endpoint /openapi.json
    options.SwaggerDoc( name: "openapi", new OpenApiInfo
    {
        Title = "Repost",
        Version = "0.0.1",
        Description = "Repost API written in ASP.NET Core 3.1 Web APIs\n\n" +
                      "[View source code on GitHub](https://github.com/pckv/repost-aspnet)\n\n" +
                      "Authors: pckv, EspenK, jonsondrem"
    });
```

Figur 196. Konfigurasjon for SwaggerGen

Videre må vi legge til skjema for autentisering, som bestemmer blant annet type, flow og sti. Som i Spring Boot vil ikke autentiserte metoder automatisk bli dokumentert med autentiseringsmetode eller mulige svar fra APIen. For å oppnå dette må vi lage vår egen

OperationFilter. Dette filteret kan se etter annotasjoner på metodene for å finne ut hvordan de skal dokumenteres i OpenAPI.

I vår API er alle endepunkt som krever innlogging annotert med [*Authorized]*. Dermed kan vi lage vår *OperationFilter* slik at den automatisk dokumenterer med autentiseringsmetode og mulige responser for alle metoder som er annotert med *[Authorized]*. Filterklassen må defineres som et filter til SwaggerGen for å brukes.

```
// Add OpenAPI security scheme for OAuth2
options.AddSecurityDefinition( name: "oauth2", new OpenApiSecurityScheme
{
    Type = SecuritySchemeType.OAuth2,
    Flows = new OpenApiOAuthFlows
    {
        Password = new OpenApiOAuthFlow
        {
            TokenUrl = new Uri( uriString: "/api/auth/token", UriKind.Relative),
            Scopes = new Dictionary<string, string>
            {
                {"user", "User access"}
            }
        }
    }
});

// Add OpenAPI operation filter to add security schemes to all authorized endpoints automatically
options.OperationFilter<OpenApiAuthorizationCheckFilter>();
```

Figur 197. OperationFilter som automatisk legger til autentiseringsmetode på endepunkter som krever innlogging

For å kunne legge ved tekst som navn og beskrivelse må vi konfigurere dokumentasjonsgeneratoren til å inkludere tekst som blir skrevet i XML format over hvert endepunkt. XML fila blir generert når koden bygges, slik at SwaggerGen kan lese den.

```
// Enable use of generated XML document to annotate endpoint name and description in OpenAPI
var assemblyName :string?  = Assembly.GetExecutingAssembly().GetName().Name;
var path :string  = Path.Combine(AppContext.BaseDirectory, $"{assemblyName}.xml");
options.IncludeXmlComments(path);
```

Figur 198. Konfigurasjon for å inkludere tekst fra XML i OpenAPI dokumentasjon

Swagger UI må også settes opp. Dette skjer i runtime i Configure metoden. Her velger vi dokumentet som inneholder OpenAPI spesifikasjonen, altså det dokumentet som blir automatisk generert. Videre må vi sette hvilken sti siden skal være tilgjengelig på. For å følge implementasjonen i FastAPI og Spring Boot ønsket vi å sette OpenAPI stien til */openapi.json*. Dette kan ikke gjøres statisk, men vi kan sette stien til alle kjente OpenAPI dokumenter som *{documentName}.json*. Ettersom vi kalte vårt dokument *openapi*, får vi da ønsket sti.

```
// Use OpenAPI with the route set to {documentName}.json (resolves to openapi.json)
app.UseSwagger(options => options.RouteTemplate = "{documentName}.json");

// Use Swagger UI on top of OpenAPI
app.UseSwaggerUI(options =>
{
    options.SwaggerEndpoint( url: "/openapi.json",  name: "Repost API 0.0.1");
    options.RoutePrefix = "api/swagger";
});
```

Figur 199. Konfigurasjon av Swagger UI som bruker den genererte OpenAPI filen

Når alt er satt opp kan vi dokumentere endepunktene. Vi bruker XML kommentarer som inneholder navnet og beskrivelsen til endepunktet for å dokumentere hva det gjør. I tillegg blir stien og modellen for svaret hentet fra metoden. Hvilken mediatype som skal brukes er satt automatisk, men kan også bli satt manuelt.

```
/// <summary>Get Current User</summary>
/// <remarks>Get the currently authorized user.</remarks>
[HttpGet]
[Route( template: "me")]
[Authorize]
Per Christian Kvalvik
public User GetCurrentUser()
{
    return GetAuthorizedUser();
}
```

Figur 200. Endepunkt som viser tekst i XML som skal vises i Swagger UI

I det første eksempelet ser vi at vi ikke har noen respons status. Et godkjent API kall får status *200 OK*, som forventet. De endepunktene som bruker autorisering får automatisk lagt til *400 Bad Request* og *401 Unauthorized* som mulige responser. I tillegg blir det lagt til en referanse til autoriserings skjemaet. Om vi trenger flere mulige responser kan vi legge til disse i endepunktet. Dette gjelder både om vi skal ha en annen suksess respons eller om det er en feilmelding. Dette skjer automatisk gjennom *OperationFilter* klassen som var beskrevet tidligere. I eksempelet bruker vi *204 No Content* som respons for en suksess.

```
/// <summary>Delete Current User</summary>
/// <remarks>Delete the currently authorized user.</remarks>
[HttpDelete]
[Route( template: "me")]
[ProducesResponseType( statusCode: StatusCodes.Status204NoContent)]
[Authorize]
Per Christian Kvalvik
public ActionResult DeleteCurrentUser()
{
```

Figur 201. Endepunkt med spesifisert statuskode ved suksess

5 Drøfting

5.1 Evaluering av resultatet

I dette kapittelet skal vi drøfte rundt resultatet av ytelsestestene og kriteriene sett fra en utviklers perspektiv for de forskjellige rammeverkene.

5.1.1 Ytelse

Fra det vi kan se og lese i Resultat av målinger med JMeter (4.2) er det et klart skille på ytelsen mellom de tre API implementasjonene.

Spring Boot API-et hadde den beste ytelsen med tanke på forespørsler per sekund og gjennomsnittlig responstid. Gruppering av responstider er veldig jevnt mellom Spring Boot og ASP.NET. ASP.NET ligger på andreplass på ytelse og FastAPI er tregest av de tre. At FastAPI er tregere enn de to andre er som forventet, siden Python ikke er et kompilert språk. Forskjellen mellom Spring Boot og ASP.NET var ikke helt ventet. Spring Boot er 1,4 ganger så raskt på GET og 2,4 ganger så raskt på POST som ASP.NET. Begge er skrevet i raske kompilerte språk, så vi forventet at disse skulle være jevnere.

Ytelsen er ganske lik mellom load plan A og load plan B med 25 tråder. Selv om det blir mange flere kall i plan B greier API-et å holde lik hastighet gjennom hele testen. Da kan vi regne med at ytelsen vi ser ikke vil falle over tid. Vi vil anta at flere forespørsler ikke vil ha noen umiddelbart negativ effekt.

Vi kjørte plan B testen med 10, 25 og 50 tråder (simulerte brukere). Med mindre tråder er responstiden lavere for alle API-er. Når vi øker antall simulerte brukere øker også responstiden. ASP.NET og Spring Boot har en ganske lav responstid på henholdsvis 15 og 10 ms selv med 50 tråder i testen, men FastAPI har over 37 ms responstid i samme test.

Ytelsen for 10 tråder er nesten lik ytelsen for 25 tråder for det enkelte API-et. Testen med 50 tråder har litt andre resultater. Relativt til resultatene med 25 tråder øker forespørsler per sekund for ASP.NET og FastAPI. Spring Boot har en liten nedgang i forespørsler på sekund når det blir flere tråder. Hvorfor FastAPI og ASP.NET øker i antall forespørsler per sekund mens Spring Boot går tregere var ikke forventet. Hvorfor det ble slik har vi ikke et klart svar på.

I alle resultatene for plan B ser vi at FastAPI er veldig treg til å slette kommentarer og poster. I alle tre testene ser vi at hastigheter øker gradvis. Hvorfor dette skjer med FastAPI og ikke de to andre har vi ikke fastslått. Som vi har skrevet i Mulige feilkilder (5.1.1.1) kan vi ha brukt lazy-loading feil i forhold til de andre implementasjonene, eller så kan det ha noe med databaseadapteren vi brukte.

5.1.1.1 Mulige feilkilder

Når vi har målt ytelse brukte vi JMeter til å kjøre forespørsler mot API-et i de forskjellige rammeverkene. Vi har altså testet implementasjonen som helhet. Alle tre bruker samme virtuelle server og den samme PostgreSQL serveren i testene, med hver sin database. Feilkilder som kan ha oppstått vil derfor antas å finne sted utenfor selve miljøet.

Selv om vi bruker samme database vil de ulike språkene ha sine egne måter å kommunisere med databasen på. Vi mener derfor at en viktig feilkilde ligger i implementasjonen av databaseadapterne vi har brukt. Ytelsestestene våre vil være påvirket av ytelsen til kommunisering med databasen, og intern implementasjon av databasemodeller i koden. Denne feilkilden kunne vi ha undersøkt videre ved å teste noen andre databaseadaptere og sammenlignet hastigheten mellom dem.

Som beskrevet i Avhengigheter (3.4.1.5) bruker FastAPI løsningen vår psycopg2 som databaseadapter. Her kunne vi ha testet med en raskere adapter som asyncpg, som implementerer en asynkron databaseadapter slik at den vil få bedre ytelse. Den er laget av de samme utviklerne som står bak uvloop (*asyncpg — asyncpg Documentation*, no date).

Samtidig er det mulig at de forskjellige driverne håndterer lazy loading forskjellig, slik at det blir forskjeller på hvilke SQL queries som faktisk utføres. Her er det mulig at noen implementasjoner utfører flere operasjoner enn nødvendig. Dette kan komme av dårlig planlegging på vår side: kanskje vi ikke burde ha brukt lazy loading slik at vi kunne ha lastet inn nøyaktig de samme objektene i hvert språk. Men her er det også mulig vi kunne ha forsikret oss om at alle endepunktene utfører de samme operasjonene mot databasen. Begge disse løsningene hadde eliminert denne feilkilden.

En annen feilkilde kan ligge i hastigheten på biblioteket som er brukt til JSON serialisering. Denne feilkilden er ikke i nærheten like relevant ettersom vi ikke sender veldig mye data imellom, men med flere forespørsler samtidig vil det til slutt ha en grad av påvirkning på gjennomsnittlig responstid. Her kunne vi også ha testet noen forskjellige biblioteker til JSON serialisering i de ulike rammeverkene for å få bedre innsikt om i hvor stor grad dette påvirker ytelse.

I samme stil kan også biblioteket som er brukt til JWT påvirke de endepunktene som krever autentisering. Hver gang en forespørsel krever autentisering må den sjekke JWT som ble sendt med. Dette krever også selvsagt ressurser, men det blir mest relevant i testene som bruker POST og DELETE ettersom GET endepunktene i testene vår ikke krever autentisering. Dette er også trolig en veldig minimal feilkilde, men som kan utgjøre en større effekt med flere samtidige forespørsler.

Til slutt er det verdt å bemerke at vår implementasjon av API-ene har variert mellom de tre rammeverkene. Her har vi forsøkt å forholde oss til hva som er vanlig i språkene, men det er også mulig at vi kunne ha fått bedre ytelse ved å bruke andre strukturer eller prinsipper.

5.1.2 Kriteriene for rammeverkene

5.1.2.1 OpenAPI dokumentasjon med støtte for Swagger UI

En komplett beskrivelse av OpenAPI implementasjonsdetaljer er skrevet i Integrere OpenAPI dokumentasjon (4.8.3).

Integrering av OpenAPI og Swagger UI var veldig enkelt i Spring Boot. Vi må legge til en avhengighet og gjøre en konfigurasjon for at OAuth2 skal kunne brukes i Swagger UI. Etter det er gjort er det enkelt å legge til dokumentasjon for hvert endepunkt. I resultatet har vi vurdert Spring Boot en grad lavere enn de andre rammeverkene. Grunnen til dette var at man måtte legge til annoteringer for hvilken sikkerhet som er brukt for hvert endepunkt der de er definert. Her føler vi at det burde ha vært støtte for å automatisk legge til skjema basert på om endepunktet er annotert med *@Authorized*.

ASP.NET trenger også en avhengighet for OpenAPI, men vi hadde ingen problemer med å bruke dette. Her er problemet beskrevet ovenfor løst ved at utvikleren selv kan implementere en metode som legger til OpenAPI dokumentasjon basert på hvordan endepunktene er annotert.

FastAPI har integrert støtte for OAuth2 med OpenAPI, med unntak av at man må manuelt annotere de ekstra responsene som kan bli gitt av sikkerheten.

5.1.2.2 OAuth2 autentisering med password flow

En komplett beskrivelse av OAuth2 implementasjonsdetaljer er skrevet i Bruke OAuth2 med password flow (4.8.1).

I dette resultatet har vi forklart at det var vanskelig å sette opp OAuth2 i Spring Boot. Dette var hovedsakelig grunnet forvirrende dokumentasjon hvor vi ikke klarte å finne en enkel forklaring på hvordan OAuth2 med password flow kan løses i dette rammeverket. Som beskrevet i Spring Boot implementasjon (4.7.2) endte vi opp med å basere løsningen på et GitHub prosjekt, som var det eneste eksempelet vi fant på hvordan password flow fungerer i Spring Boot. Etter sammenligning av dette prosjektet og Spring Boot sin egen dokumentasjon fikk vi mer innblikk i hvor denne løsningen var beskrevet, men det var fortsatt mange forskjeller mellom den offisielle dokumentasjonen og eksempelprosjektet vi fant.

I ASP.NET var det enklere å forstå hvordan OAuth2 settes opp ettersom offisiell dokumentasjon henviste til en rik og omfattende dokumentasjon av forskjellige måter å sette opp OAuth2 på. Fremdeles var det mer komplisert enn i FastAPI, og dermed tok det også her lengre tid å implementere dette slik vi ønsket. Samtidig støtter ikke symmetrisk signatur av JWT, som gjorde at vi ikke kunne bruke en secret slik som i de andre rammeverkene. Dette har derimot ingen effekt på resultatet av selve API-et.

5.1.2.3 ORM for å lage tabeller og relasjoner fra klasser

Alle rammeverkene har støtte for å lage klasser som blir gjort om til tabeller med ORM. I Spring Boot og ASP.NET kan vi bruke samme modell for både database og API skjema. Dette er ikke

mulig med FastAPI. ASP.NET skiller seg positivt ut siden vi ikke ikke trenger å spesifisere relasjoner og andre databasekonsept siden det er godt integrert med språket.

5.1.2.4 ORM for å lage spørringer til databasen

I FastAPI og ASP.NET lager vi spørringer, men i ASP.NET hadde vi et problem der vi måtte spesifisere hvilke relasjoner som skal lastes inn for hver spørring. Vanligvis kunne vi ha løst dette ved å bruke lazy loading i databasen, men her støtte vi på et problem i koblingen mellom JSON skjemaet og databasemodellen hvor *[JsonIgnore]* attributten ikke ble brukt i noen tilfeller. Dermed var det forskjeller i formatering av data basert på endepunktet som ble brukt. Utdrag fra commit *5c0ad12* i *pckv/repost-aspnet*:

> *"Add post endpoints and revert lazy loading.*
>
> *Lazy loading has an issue with JSON serializing when returning a single element. GetResubs returned a correctly serialized resubs, whereas GetResub included the entire owner object and a lazyLoading parameter."*

Spring Boot bruker et annet system der vi spesifiserer en metode i datarepository og spørringen blir automatisk generert. Dette gjør det veldig enkelt å lage spørringer uten å tenke på implementasjonen.

5.1.2.5 Enkelt å modellere endepunkter og parametere i endepunktene

Hovedsakelig var det enkelt å modellere API-et og endepunktene slik vi ønsket, med unntak av PATCH endepunkter. I Spring Boot og ASP.NET måtte vi bruke en egen løsning for å kunne skille mellom felt med verdien null og felt som ikke har en verdi. Å vite forskjell på dette er viktig når man skal gjøre en oppdatering på et objekt. Dette trekker litt ned på brukervennligheten. Hvordan vi løste det er beskrevet i Oppdatere objekter med PATCH endepunkt (4.8.2). En alternativ løsning hadde vært å bruke PUT, slik at brukeren av API-et kunne ha hentet en ressurs, endret verdiene og sendt hele ressursen tilbake i en PUT forespørsel. For vår front-end demo har vi allerede hentet objektene og kunne derfor ha brukt PUT.

5.1.2.6 Enkelt å bygge og kjøre applikasjonen i et testmiljø

I kapittelet Oppsett av testmiljø (3.7) er det beskrevet i detalj hvordan man setter opp alt i et testmiljø.

FastAPI lager ikke sitt eget miljø. Vi må derfor lage et virtuelt miljø manuelt for å holde det atskilt fra eventuelt andre Python prosjekter som kjører på samme server. Dette gjør det ikke mye vanskeligere å bruke, men det er et ekstra steg i prosessen som man slipper å tenke på med de to andre løsningene.

5.1.2.7 Støtte for alle populære operativsystem slik at utvikleren kan jobbe i sitt foretrukne miljø

Selv om gunicorn ikke fungere på Windows er ikke dette en veldig viktig del av FastAPI. Uvicorn som det er lagt opp til å bruke under utviklingen fungerer på alle plattformer, og for å kjøre flere

workers finnes det alternativer til gunicorn som også fungerer på Windows. Ellers fungerer alt vi har brukt på alle populære operativsystemer.

5.1.3 Tilgjengelig dokumentasjon

I dette kapittelet drøfter vi rundt hvordan dokumentasjonen i de ulike rammeverkene er satt opp og hvordan dette har fungert for oss.

5.1.3.1 FastAPI

På FastAPI sin hjemmeside (*FastAPI*, no date) får man en rask oversikt over hva FastAPI er og hvordan man installerer det. Videre er det er enkel gjennomgang av hvordan man lager et veldig lite eksempel prosjekt og kjører dette for å demonstrere den enkleste bruken og alt som allerede er inkludert. Videre blir det lagt til små oppgraderinger til eksempelet og mer av bruken blir forklart. Dokumentasjonen på hjemmesiden er den vi har brukt til det meste av dette prosjektet.

Features siden (*Features - FastAPI*, no date) går gjennom mye av den viktigste funksjonaliteten til rammeverket. For at det skal være enkelt å komme i gang første gang er det laget en brukerveiledning som følger utvikleren steg for steg (*Tutorial - User Guide - Intro - FastAPI*, no date). Denne veiledningen forklarer forskjellige konsepter og gjør det enkelt å forstå sammenhengen mellom konseptene og hvordan man implementerer det i rammeverket. For de som allerede har jobbet med FastAPI eller lignende rammeverk finnes det en avansert brukerveiledning (*Advanced User Guide - Intro - FastAPI*, no date). Denne trenger ikke nødvendigvis å være avansert, men den går nærmere i detalj i emnene som er beskrevet.

Det er også flere sider med informasjon, som for eksempel deployment (*Deployment - FastAPI*, no date) eller alternative rammeverk i Python (*Alternatives, Inspiration and Comparisons - FastAPI*, no date).

Om man bruker en søkemotor for å finne hjelp eller dokumentasjon til FastAPI er det gjerne hjemmesiden som dukker opp først, men det er også en god del aktivitet med GitHub Issues som også er langt oppe i resultatene. Noen artikler er det på forskjellige sider, men FastAPI er relativt nytt og lite utbredt.

5.1.3.2 Spring Boot

Spring Boot er et stort rammeverk og det er mange ressurser å bruke når man trenger hjelp. På hjemmesiden til Spring Boot får man en oversikt over noe av det rammeverket kan brukes til, med linker til veiledninger og andre generelle eller spesifikke emner. Man kan også velge en annen fane og vise lenker til referanse og API dokumentasjonen.

API dokumentasjonen (*Spring Boot Docs 2.2.6.RELEASE API*, no date) er Javadoc der man kan se alle klasser og metoder i hele rammeverket. Dette er rent teknisk dokumentasjon, og det er tungt å lese seg igjennom slik dokumentasjon uten å vite hva man egentlig leter etter, men det er bra å ha tilgang til denne om det oppstår en situasjon der man trenger å forstå noe på et mer detaljert nivå.

Referanse dokumentasjonen (Webb *et al.*, no date) er ikke bare teknisk, men den viser også hvordan man kan og skal bruke visse deler av rammeverket. Det er delt inn i seksjoner som igjen har underkategorier. Det som er av særlig interesse for noen som skal bruke rammeverket for første gang er seksjonene *Documentation Overview*, *Getting Started* og *"How-to" guides*. Her får man informasjon om hvordan man kan bruke Spring Boot og hva som er de første stegene for å lage sitt eget prosjekt med dette verktøyet. Det er også en egen guide for å sette opp en veldig enkel REST service (*Building a RESTful Web Service*, no date). Andre kategorier er mer avanserte og noen man kan komme tilbake til etterhvert.

Om man bruker en søkemotor for å finne ressurser for Spring Boot er det to sider som stadig dukker opp. Den ene er den offisielle siden til Spring, den andre siden er Baeldung (baeldung, 2020b).

5.1.3.3 ASP.NET

ASP.NET er Microsofts førsteparti rammeverk for utvikling av web applikasjoner og API. Selv om ASP.NET Core er open source er ikke ASP.NET rammeverket det. Som resultat er også ASP.NET Core hovedsakelig styrt av Microsoft, som leverer de fleste standarde løsningene man skulle trenge i en web applikasjon. Det er som regel Microsoft sin egen dokumentasjon man går til for å utvikle funksjonalitet (Pickett, no date).

Microsoft leverer grundig dokumentasjon med eksempler. Vi startet med template kode for en web API (Addie and Dykstra, 2020) og fikk forståelse derfra hvordan strukturen på prosjektet skal være, hvordan man lager kontrollere og endepunkter, og fikk se hvordan noe intern .NET metodikk skulle behandles. Videre fulgte vi resten av guidene for web API applikasjoner i dokumentasjonen som gikk i nærmere detalj på hvordan ulike parametere settes opp og andre typer endepunkt som POST, PATCH og DELETE. Disse guidene inkluderes også i videoformat (Anderson, Larkin and Wasson, 2020).

I tillegg til gode guider for oppsett av diverse miljøer en utvikler skal trenge har de også en full referanse på alle klassene og strukturene som ligger i ASP.NET Core. Her er referanser dokumentert for alle tilgjengelige versjonene av ASP.NET Core, både gamle og nye som ikke er utgitt ennå. Enhver struktur som skal brukes av utvikleren kan slåes opp her og studeres. For oss som utviklere har ikke referansen vært i nærheten like nyttig som guidene, ettersom strukturene på de forskjellige klassene i rammeverket ikke viser hvordan de skal brukes i en ønskelig kontekst (*.NET API browser*, no date).

Samtidig refererer selv Microsoft sin egen dokumentasjon til tredjepartsverktøy. For eksempel brukes Swashbuckle som OpenAPI generator, og dokumentasjonen til Microsoft inneholder guider for bruk av denne fremfor en selvlaget løsning. Det var også denne guiden vi brukte da vi skulle sette i gang med støtte for OpenAPI og Swagger UI (Nienaber and Suter, 2019). Et annet eksempel er autentisering, hvor Microsoft har egne løsninger men anbefaler å bruke IdentityServer4 dersom man skal lage en løsning med OAuth2 (Anderson, Larkin and Wasson, 2020).

Om man skal finne løsninger gjennom en søkemotor er det oftest Microsoft sin egen dokumentasjon som dukker opp. Man finner også mange artikler på stackoverflow.com hvor brukere stiller mer spesifikke spørsmål. Disse får ofte mange svar og dermed er det lett å finne løsninger som fungerer dersom Microsoft sin dokumentasjon ikke leverer. Men samtidig er det stor fare for at disse løsningene er utdaterte og egnet til eldre versjoner av .NET Core, eller til og med egnet for den Windows-eksklusive .NET Framework.

5.2 Evaluering av prosjektet

I dette kapittelet skal vi diskutere rundt hvorfor vi valgte API rammeverkene vi gjorde og hvordan det var å arbeide med de verktøyene vi hadde valgt, diverse utfordringer vi hadde underveis og hva som generelt har gått greit.

5.2.1 Utviklingsprosess

Å bruke GitHub Projects som Kanban tavle har fungert veldig bra. Siden det er veldig enkelt å se hva som må gjøres og hva som er underveis eller ferdig har alle på gruppen en god oversikt over hvordan vi ligger an. Vi kunne med fordel ha brukt flere milepæler for skriving av rapporten slik at tidsperspektivet hadde vært mer definert på forhånd. For kodeprosjektene hadde vi ingen problemer, her gikk det ganske rett fram slik vi hadde planlagt.

Bruken av pull request med review er vi veldig fornøyd med. Det gjør at flere må sette seg inn i det som er gjort. Da er det bedre forståelse for helheten av prosjektet og det blir enklere å diskutere som en gruppe når vi alle har en viss forståelse for hverandres kode.

På grunn av situasjonen med coronaviruset har vi måtte endre litt på måten vi jobber, men for det meste var vår arbeidsstruktur slik at vi kunne fortsette hver for oss. Den største endringen er at vi ikke kan møtes fysisk. Internt i gruppen skriver vi på Discord som før, men når vi skal planlegge eller snakke om framgang på prosjektet blir det møte over nett i stedet for å møtes på labben på skolen. Det samme gjelder møte med veileder. Vi sender fortsatt epost annenhver uke med en oppsummering av det som er gjort med relevante linker til GitHub Projects. Møtet har blitt tatt over Discord i stedet for på skolen her også.

All koding og annet arbeid vi gjør kan gjøres hjemme. Vi har ingen behov for å være på skolen og bruker ingen fysiske verktøy som kun er tilgjengelige på labb.

5.2.2 JetBrains IDE

Vi brukte JetBrains IDE-er til all koding i prosjektet. Vi opplevde ikke noen problemer med selve verktøyet eller integrasjonen med rammeverket. Når vi byttet mellom språk og rammeverk så var det veldig kjekt at alle IDEene så like ut slik at vi ikke måtte lete for å finne enkel funksjonalitet.

5.2.3 Git og GitHub

Arbeidsflyten er veldig enkel når man har alt samlet på ett sted. Vi bruker GitHub for å lage issues og legger labels på disse for kategorisering og eventuelt legger de til en milepæl. De blir

lagt inn i et GitHub Project som er kanban tavlen vår der alle kan se hva som må gjøres og hva det jobbes med. Alle i gruppen har dermed full oversikt over alt som skal gjøres og eventuelt når det må være ferdig.

Å lage ny branch, pull request og dermed ha review og merge for hver ny endring hjelper oss med å holde koden ryddig. Med en slik prosess vil vi alltid ha en stabil master branch og når ny funksjonalitet testes blir det gjort før det blir lagt i master for å unngå problemer. Om noe er feil og blir oppdaget i testen av pull requesten er det mulighet for å kommentere på problemet eller komme med helt konkrete forslag til endringer der den som laget pull requesten kan trykke godkjenn på forslag for å automatisk legge det til. Siden du vet noen andre skal lese og godkjenne koden før den blir brukt er det ekstra viktig at vi skriver lett leselig kode med god dokumentasjon og kommentarer for komplekse deler av koden.

5.2.4 Discord

Vi har brukt Discord for intern kommunikasjon både med tekst og samtaler. På grunn av at vi ikke kan møtes fysisk med over to måneder igjen av oppgaven var det veldig greit at vi allerede brukte Discord aktivt. Å ta møter over Discord oppringing ble dermed ingen utfordring for oss eller veileder.

De fleste funksjonene som vi har brukt i Discord er også tilgjengelige i andre verktøy, men siden alle i gruppen allerede var kjent med og hadde konto for Discord ble det et naturlig valg i stedet for å velge et nytt verktøy.

5.2.5 Google Docs og Microsoft Word

Det har fungert veldig fint å skrive rapporten i Google Docs. Da vi nærmet oss slutten av skrivingen oppdaget vi at kvaliteten på bilder blir veldig redusert når man laster ned en PDF direkte fra Google Docs. Ved å laste ned filen som *.docx* filformat og bruke Microsoft Word for å generere PDF blir resultatet bedre. Dette var et hinder, men heldigvis hadde vi tid til å finne en løsning.

5.2.6 Valg av autoriseringsmetode med OAuth2

Da vi valgte hvordan vi skulle håndtere autentisering av brukere gikk vi for å lage vårt eget autoriseringssystem. Alternativet er å bruke en tredjepart til autoriseringen, hvor vi kunne ha koblet opp rettigheter hos brukerne ved å la de logge inn med en eksisterende brukerkonto på en annen platform. Ved den løsningen hadde vår API kun håndtert autentisering, og overlatt brukerens egen sikkerhet hos en tredjepart. For prosjektet vårt så vi flere fordeler ved å gjøre dette lokalt ved bruk av en egen autoriseringstjeneste:

- Ved å styre autorisering selv er prosjektet vårt helt selvstendig og kan kjøre i ethvert miljø uten å kreve at det er mulig å kommunisere med en tredjepart.
- Autorisering gjennom en annen platform begrenser hva slags tester som kan utføres av testverktøyet. For eksempel vil vi trenge flere brukerkontoer på tredjeparten for å kunne teste forskjellige rettighetsnivå, som at en resub owner skal kunne slette andres poster.

- Autorisering kan trekkes med i ytelsesmålene og sammenlikning av dokumentasjon/tilgjengelighet til å implementere en slik komponent, ettersom løsningen vil være ønskelig for noen.

Vi valgte å bygge en slik autoriseringstjeneste ved å bruke OAuth2 password flow. Dette gjør at vi kan bruke andre OAuth2 flows senere dersom vi skulle utvide API-et for å inkludere tredjeparter i autorisering, eller gi tredjepart "apper" rettigheter til begrensede deler av API-et. Ved bruk av en standard som OAuth2 forsikrer vi oss for kompatibilitet med flere autentiseringsløsninger. OAuth2 Bearer Tokens er brukt til autentiseringen, og denne delen vil også være kompatibel med andre flows. I et tilfelle hvor flow skulle endres hadde denne funket på samme måte som før i front-end. Vi får også eksperimentert med støtten i OAuth2 standarden i de forskjellige rammeverkene.

Sammen med fordelene vi så for vårt eget brukstilfelle er det noen ulemper ved OAuth2 password flow som bør nevnes. OAuth 2.0 Security Best Current Practice (Lodderstedt *et al.*, 2019, sec. 2.4), som er del av IETF av OAuth2, sier følgende om OAuth2 password flow:

> *"The resource owner password credentials grant MUST NOT be used. This grant type insecurely exposes the credentials of the resource owner to the client."*

Altså er det oppklart at man skal aldri benytte seg av password flow i OAuth2, og at dette er en utdatert autoriseringsform. Dette gir oss utfordringer ettersom vi har laget en løsning som baserer seg på utdatert metodikk. Allikevel mener vi at fordelene vi har listet ovenfor er gyldige.

Okta (*What is the OAuth 2.0 Password Grant Type?*, 2018) beskriver password flow som en løsning rundt å autorisere i en førstepartstjeneste og samtidig få nytte av å bruke OAuth2 videre til integrering med andre innloggingsformer og applikasjoner. De forklarer altså hvordan password flow kan brukes dersom det er veldig nødvendig å integrere innlogging med en egen brukerdatabase.

Siden vårt prosjekt ikke er langvarig, men heller bare egnet til analyse, så kan det argumenteres at denne løsningen er på plass her. Men det betyr ikke at vi ikke burde teste funksjonaliteter som ikke er egnet til produksjon. Fra utdraget i avsnittet ovenfor kan vi se at det ikke er så uvanlig å ha behov for integrasjon med en egen brukerdatabase og samtidig bruke OAuth2. Her hadde ikke vi noen eksisterende brukerdatabase, men det er likevel nyttig å vise hvordan dette kan brukes dersom noen har behov for det selv. Samtidig er det en mulighet for at sikkerheten hos et selskap krever at de lagrer brukere lokalt framfor å bruke en tredjepart tjeneste.

5.3 Evaluering av produktene

Her vil vi drøfte rundt de produktene vi har laget, altså tre implementasjoner av et API, et testverktøy og en nettside.

5.3.1 Implementasjon av API-et

Som vi ser i Resultat av produktene - API-er (4.1) har vi klart å lage API-et vi beskrev i introduksjonen i alle tre rammeverkene vi valgte. I Brukeropplevelse for en utvikler som bruker de tre rammeverkene (4.3) har vi vurdert hvor enkelt det var å bruke rammeverkene til forskjellige deler av prosessen.

Vi er veldig fornøyde med resultatet av API-et i alle rammeverkene. Vi fikk lagt inn alle hovedaktivitetene som vi beskrev i kravspesifikasjonen. Alle API-er er funksjonelle, alle har Swagger UI, OAuth2, er satt opp med PostgreSQL og er koblet til samme front-end på demo nettsiden vår.

Noe vi ikke hadde med i den originale kravspesifikasjonen er pagination på endepunkter som returnerer en liste. Det betyr at det er en begrensning på hvor mange elementer vi kan hente på én gang, men vi kan hente ut alle elementene om vi gjør flere kall der vi endrer verdien for page for hvert kall. På den måten kan vi hente hundre elementer, så de neste hundre elementene og så videre. Standardinnstillingene på endepunktene er 100 objekter og side 0, og kan bestemmes av stivariabelene *page_size* og *page*. Dette var nødvendig å legge inn for at ytelsen skal holde seg lik selv med mange resuber og poster.

Det eneste som vi ikke har implementert, men som står i kravspesifikasjonen, er mulighet for å hente en spesifikk kommentar og sortering basert på tid og stemmer. Eneste muligheten for å hente en kommentar er å hente kommentarene som tilhører en bruker eller som tilhører en post. Sortering baserer seg ikke på tid og stemmer, men kun når et objekt er laget. Det siste opprettede objektet returneres først.

5.3.1.1 Tilstandsløst API og REST API

Vi har valgt å lage tilstandsløst API i stedet for å lage REST API. Grunnen til dette er at REST har flere krav som vi ikke ville ha oppfylt i våre implementasjoner. Et av punktene som definerer REST API er caching, noe vi ikke har implementert i noen av våre API-er. Vi mener at fordelene med et tilstandsløst API er sammenlignbart med et REST API, men selvfølgelig er REST en mer komplett løsning.

5.3.1.2 Databasetilkobling med ORM

Vi har valgt en stil hvor vi kan modellere databaseobjekter i koden slik at tabeller i databasen blir laget ved bruk av ORM. En ulempe med denne stilen er at de forskjellige rammeverkene håndterer ORM på ulike måter. Til slutt fikk vi tre forskjellige databaser av hver, og vi kunne ikke koble alle API-ene opp mot samme database. Alternativt kunne vi ha valgt en ORM hvor vi hadde laget databasen først, og så hatt et verktøy for konvertering av databasetabellene til

objekter i rammeverkene. På den måten hadde alle API-ene kunne ha kjørt kobling til samme database og delt ressurser.

Grunnen til at vi ikke valgte denne stilen for ORM er at vi ønsket å teste fra et perspektiv til nye utviklere. For dagens utvikler mener vi at det er viktigere å kunne modellere i samme miljø som du utvikler, uten å måtte manuelt sette opp strukturer i databasen. Samtidig har det ingen effekt på selve API-ene at de ikke kan kobles mot samme database, grunnet at databasen er en intern detalj og ikke del av API-et fra synspunktet til en bruker. Altså ligger dette lengre utenfor fokuset til oppgaven.

5.3.2 Nettside

I *Resultat: Nettside utviklet i Vue.js* viser vi nettsiden som vi laget. Nettsiden ble laget for å demonstrere at API-et er brukbart og fungerer som back-end i en virkelig nettside. For å gjøre det tydelig at det faktisk er en front-end som fungere med alle API implementasjonene har vi gjort det mulig å endre hvilken API man benytter seg av med et enkelt valg på nettsiden. Man kan se at forespørslene blir sendt på en annen port og at innholdet er forandret for hver API man velger.

I tillegg til kun funksjonalitet har nettsiden en del styling som gjør at det ser mer ut som et ferdig produkt. Design var ikke hovedfokus i dette prosjektet. Derfor er ikke nettsiden egnet for alle typer skjermer og enheter slik en moderne nettside bør være, men for vårt formål er det mer enn bra nok.

Det er en ting som er tilgjengelig i API-et som ikke er implementert i nettsiden, og det er mulighet for å legge inn stemmer på en post eller kommentar. Man kan se totalt antall stemmer på en post eller kommentar, men kan altså ikke påvirke dette tallet via nettsiden.

Vi er fornøyd med resultatet av nettsiden. Den fungerer slik vi hadde ønsket med unntak av å kunne stemme, noe som ikke er avgjørende for funksjonaliteten. Om nettsiden hadde støttet flere skjermstørrelser ville det vært positivt, men som sagt var ikke dette viktig i dette prosjektet.

5.3.2.1 Single-page application mot multi-page application

Single-page application (SPA) og multi-page application (MPA) har en rekke ulikheter, fordeler og ulemper. Disse ulikhetene har en påvirkning på front-end applikasjonen og dermed måtte vi reflektere over hva som kunne passe best til vårt bruk.

SPA er som regel kjappere enn MPA. SPA laster ned alle ressursene til nettsiden én gang. MPA derimot må laste ned ressursene på nytt hvis brukeren går til en ny side på nettsiden. En annen nytte med SPA som er relevant for prosjektet vårt er utviklingsprosessen (*Single Page Application (SPA) vs Multi Page Application (MPA) – Which and when to use?*, 2019).

Det er lettere å gjenbruke kode i SPA i forhold til MPA, ettersom JavaScripten er lastet inn i én og samme instans uansett om brukeren navigerer gjennom nettsiden. Da blir det også mindre data å laste inn, som for eksempel når vi fanger informasjonen til den påloggede brukeren. Denne dataen trenger vi ikke å laste inn hver gang brukeren navigerer til en ny side.

5.3.2.2 Local Storage mot Cookies for token lagring

Sikkerhetsmessig så er det ikke anbefalt å bruke local storage for å lagre tokens (Botto, 2019). Mauricio Cortazar (2018) derimot argumenterer at moderne front-end rammeverk gir deg beskyttelse mot Cross Site Scripting (XSS) og Cross Site Request Forgery (CSRF), men cookies kan ikke beskyttes mot CSRF hvis det skal være håndterbart av JavaScript. Ettersom våre API-er er tilstandsløse og ikke bruker cookies vil ikke dette være noe problem.

5.3.3 Testverktøy

Fra Implementasjon av testverktøyet (4.7.5) er det beskrevet hvordan testmiljøet er satt opp. Vi var snar med utviklingen av testverktøyet og fikk det på plass samtidig som FastAPI løsningen ble utviklet. Dette gjorde at vi kunne ekstensivt kunne kjøre tester under utvikling av Spring Boot og ASP.NET implementasjonen for å sørge for at implementasjonene ble like.

Ettersom vi startet med utviklingen i FastAPI ble testverktøyet basert på hva som var gjort i FastAPI. En ulempe med dette var at vi måtte gjøre endringer i etterkant da vi implementerte API-et i de andre rammeverkene. For eksempel hadde vi ikke enda en god modell for OAuth2 da vi begynte å implementere Spring Boot API-et, hvor OAuth2 komponenten hadde flere krav på hva som trengs for å definere en OAuth2 løsning. Da måtte vi gjøre endringer i testverktøyet slik at det kunne støtte den nye modellen. I slike tilfeller kunne vi teste mot Spring Boot før vi også gjorde endringer i FastAPI, hvor vi da kunne bruke de endrede testene for å sjekke at FastAPI løsningen også støtter API-et vårt. På den måten har testverktøyet vært svært nyttig for å forsikre om at API modellen forblir den samme i alle rammeverk.

Selve testverktøyet er veldig lett å bruke og det er også veldig lett å legge til nye tester. Ettersom vi kun trenger ett funksjonskall for å gjøre en test blir det veldig abstrahert, slik at fokuset kan ligge i selve testene som er satt opp. Å ha testene oppført kronologisk er også svært hjelpsomt til å kunne forstå struktur og holde orden over dataobjektene som lages til testene.

Samtidig var det også ulemper ved denne kronologiske formen for testing. Når tester skal kjøres må verktøyet gå gjennom hele sitt løp. Det vil si at dersom kun én av testene failer så vil den ikke få kjørt noen andre tester. Siden testverktøyet vil kommunisere med og lage ressurser på API-et fører dette ofte til at det henger igjen mange ressurser. For å løse dette uten store forskjeller kunne vi ha separert testene med enda et lag, hvor det første laget hadde opprettet alle ressurser vi har behov for i testen, og det siste laget hadde fjernet ressursene igjen.

En annen ulempe med kronologisk testing er at vi ikke kan teste spesifikk funksjonalitet. Altså kan vi ikke for eksempel kun teste om stemming av en post fungerer som den skal. Dette har vært en utfordring for oss under utvikling API-et i nye rammeverk, hvor vi ikke kan teste forbi et visst punkt i testverktøyet. For eksempel implementerte vi stemming av ressurser ved slutten av utviklingsløpet, som gjorde at vi ikke kunne kjøre tester på endepunkt som ligger etter testing av stemmeendepuntkene. Et system som forklart over kunne ha vært et godt startpunkt for et testverktøy med definerte tester for de forskjellige endepunktene.

6 Konklusjon

Problemstillingen som vi tar utgangspunkt i er at det er veldig mange teknologier for å lage nettsider med separat logikk og brukergrensesnitt. Vi satte fokus på open source rammeverk for å lage et tilstandsløst API. Vi modellerte et API som vi implementerte i de tre rammeverkene FastAPI, Spring Boot og ASP.NET Core Web APIs. Med erfaringen fra dette har vi sammenlignet brukeropplevelsen og ytelsen til de forskjellige rammeverkene med Apache JMeter.

For å forsikre oss om at API-ene har lik funksjonalitet og grensesnitt laget vi et testverktøy som gjør forespørsler til API-et over HTTP. Vi valgte å lage verktøyet selv i Python slik at vi kunne gjenbruke kunnskap fremfor å bruke enda en teknologi. Å ha et verktøy som kan finne forskjeller mellom de tre API-ene ga oss en bredere oversikt under utvikling av API-et med de tre rammeverkene.

Vi har også laget en komplett demo for bruk av API-et i form av en nettside hvor API-ene dynamisk kan byttes mellom. Nettsiden er en moderne single-page application laget i Vue.js. Med dette har vi demonstrert at det er mulig å lage et funksjonelt frakoblet nettgrensesnitt med API-et vi har modellert. Ettersom man i nettsiden kan velge API som skal kommuniseres med har vi også her fått demonstrert at de tre API-ene våre fungerer på samme måte.

Selv om rammeverkene er forskjellige greide vi altså å implementere det samme API-et i alle tre. Det viser at funksjonaliteten til produktet man ender opp med ikke er strengt avhengig av rammeverket som er brukt. Moderne rammeverk for API har mye av de samme verktøyene.

Det rammeverket som hadde den beste ytelsen i våre målinger med JMeter er Spring Boot. Resultatene for Spring Boot var veldig bra, 1,4 ganger så raskt som ASP.NET på GET forespørsler og hele 3,3 ganger så raskt som FastAPI. POST forespørsler har større forskjeller med henholdsvis 2,4 og 4 ganger raskere hastighet i Spring Boot. ASP.NET og Spring har lignende responstider, men Spring er litt bedre gjennomsnittlig.

Når det kommer til brukervennlighet har vi vurdert FastAPI og ASP.NET litt bedre enn Spring Boot. Det er noen punkter der Spring Boot trenger mer konfigurasjon og spesielle løsninger i forhold til de to andre. I tillegg er det veldig mange nettressurser med informasjon og råd som ikke lengre er gyldig i nyere versjoner. Vi tror at for en utvikler som ikke har kjennskap til noen av de tre vil FastAPI og ASP.NET være de enkleste å lære.

Om vi skal se på det rammeverket som kommer best ut sammenlagt er det ASP.NET som er vårt valg. Her får du et rammeverk med god dokumentasjon som er enkelt å lære seg med fortsatt bra ytelse.

Når man skal lage et API er det lurt å ha en viss formening om hvor stor trafikk det vil være. Vet man med sikkerhet at det vil være veldig stor last må man velge rammeverk ut ifra det. Hvis API-et skal brukes til noe som ikke vil ha ekstrem last vil brukervennlighet ovenfor utvikleren være

høyt prioritert. Våre resultater viser til at Spring Boot gir best ytelse, men det er fortsatt viktig å ta i betraktning hva slags trafikk man kan forvente å få i produktet.

Når man skal velge et rammeverk for bruk i en bedrift tror vi det viktigste er at utviklerne er komfortable med språket som rammeverket er skrevet i. Om utviklerteamet er vant til en teknologi eller programmeringsspråk vil et rammeverk innenfor samme teknologi eller språk være det logiske valget. Den største kostnaden for utvikling er ofte knyttet til hvor lang tid utviklingen tar, og derfor vil man at tiden skal bli brukt til utvikling fremfor opplæring for et nytt språk. Derfor vil vi konkludere med at ASP.NET er det beste rammeverket for en bedrift hvor ytelsen til produktet ikke er av høyeste prioritet, som vi tror gjelder de fleste små og mellomstore prosjekter.

7 Referanser

About - Git (no date). Available at: https://git-scm.com/about (Accessed: 25 March 2020).

Access Tokens - OAuth 2.0 Simplified (no date) *OAuth 2.0 Simplified*. Available at: https://www.oauth.com/oauth2-servers/access-tokens/ (Accessed: 11 May 2020).

Addie, S. and Dykstra, T. (2020) *Create web APIs with ASP.NET Core*, *Microsoft Docs*. Available at: https://docs.microsoft.com/en-us/aspnet/core/web-api/ (Accessed: 11 May 2020).

Advanced User Guide - Intro - FastAPI (no date). Available at: https://fastapi.tiangolo.com/advanced/ (Accessed: 13 May 2020).

Alternatives, Inspiration and Comparisons - FastAPI (no date). Available at: https://fastapi.tiangolo.com/alternatives/ (Accessed: 13 May 2020).

Anderson, R., Larkin, K. and Wasson, M. (2020) *Tutorial: Create a web API with ASP.NET Core*. Available at: https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api (Accessed: 6 May 2020).

An Introduction to JavaScript (2020). Available at: https://javascript.info/intro (Accessed: 13 May 2020).

Apache JMeter - Apache JMeter™ (no date). Available at: https://jmeter.apache.org/ (Accessed: 27 April 2020).

Apache JMeter - Download Apache JMeter (no date). Available at: https://jmeter.apache.org/download_jmeter.cgi (Accessed: 8 May 2020).

Apache JMeter - User's Manual: Component Reference (no date). Available at: https://jmeter.apache.org/usermanual/component_reference.html#JSON_Extractor (Accessed: 7 May 2020).

Apache JMeter - User's Manual: Functions and Variables (no date). Available at: https://jmeter.apache.org/usermanual/functions.html#__RandomString (Accessed: 7 May 2020).

Apache JMeter - User's Manual: Generating Dashboard Report (no date). Available at: https://jmeter.apache.org/usermanual/generating-dashboard.html (Accessed: 5 May 2020).

Apache License, Version 2.0 | Open Source Initiative (no date). Available at: https://opensource.org/licenses/Apache-2.0 (Accessed: 12 May 2020).

API Resources | Swagger (no date). Available at: https://swagger.io/resources/open-api/ (Accessed: 19 April 2020).

ArchWiki (2020) *systemd/User: Automatic start-up of systemd user instances - ArchWiki*. Available at: https://wiki.archlinux.org/index.php/Systemd/User#Automatic_start-up_of_systemd_user_instances (Accessed: 7 May 2020).

ASGI Documentation — ASGI 2.0 documentation (no date). Available at: https://asgi.readthedocs.io/en/latest/ (Accessed: 13 May 2020).

ASP.NET Web APIs | Rest API's with .NET and C# (no date) *Microsoft*. Available at: https://dotnet.microsoft.com/apps/aspnet/apis (Accessed: 6 May 2020).

asyncpg — asyncpg Documentation (no date). Available at: https://magicstack.github.io/asyncpg/current/ (Accessed: 14 May 2020).

Atlassian (2018) *Scrum - what it is, how it works, and why it's awesome*, *Atlassian*. Available at: https://www.atlassian.com/agile/scrum (Accessed: 26 February 2020).

Atlassian (2019) *Kanban - A brief introduction | Atlassian*, *Atlassian*. Available at: https://www.atlassian.com/agile/kanban (Accessed: 3 March 2020).

Atlassian (no date) *What is Agile? | Atlassian*, *Atlassian*. Available at: https://www.atlassian.com/agile (Accessed: 23 February 2020).

auth0.com (no date) *JWT.IO - JSON Web Tokens Introduction*. Available at: http://jwt.io/ (Accessed: 11 May 2020).

Authentication vs. Authorization (2018) *Okta*. Available at: https://www.okta.com/identity-101/authentication-vs-authorization/ (Accessed: 11 May 2020).

baeldung (2020a) *Introduction to Project Lombok | Baeldung*, *Baeldung*. Available at: https://www.baeldung.com/intro-to-project-lombok (Accessed: 4 May 2020).

baeldung (2020b) *REST with Spring Tutorial | Baeldung*, *Baeldung*. Available at: https://www.baeldung.com/rest-with-spring-series (Accessed: 4 May 2020).

BCrypt.Net-Next 4.0.0 (no date). Available at: https://nuget.org/packages/BCrypt.Net-Next/ (Accessed: 13 May 2020).

beanshell (no date) *beanshell/beanshell*, *GitHub*. Available at: https://github.com/beanshell/beanshell (Accessed: 10 May 2020).

Botto, B. (2019) 'Secure Access Token Storage with Single-Page Applications: Part 1'. Medium. Available at: https://medium.com/@benjamin.botto/secure-access-token-storage-with-single-page-applications-part-1-9536b0021321 (Accessed: 14 May 2020).

Building a RESTful Web Service (no date) *Spring*. Available at: https://spring.io/guides/gs/rest-service/ (Accessed: 4 May 2020).

Citations and bibliographies for Google Docs - Paperpile (no date). Available at: https://paperpile.com/features/google-docs-citations-bibliography (Accessed: 15 May 2020).

Comparison with Other Frameworks — Vue.js (no date). Available at: https://vuejs.org/v2/guide/comparison.html (Accessed: 13 May 2020).

Computer Hope (no date) *What is HTML (Hypertext Markup Language)?* Available at: https://www.computerhope.com/jargon/h/html.htm (Accessed: 13 May 2020).

Contributors to Wikimedia projects (2020) *Java (programming language) - Wikipedia*, *Wikimedia Foundation, Inc.* Available at: https://en.wikipedia.org/w/index.php?title=Java_(programming_language)&oldid=950596491 (Accessed: 19 April 2020).

Cortazar, M. (2018) *Is it safe to store a jwt in localStorage with reactjs?*, *Stack Overflow*. Available at: https://stackoverflow.com/questions/44133536/is-it-safe-to-store-a-jwt-in-localstorage-with-reactjs (Accessed: 14 May 2020).

DavidWells (2013) *reset.css*, *Gist*. Available at: https://gist.github.com/DavidWells/18e73022e723037a50d6 (Accessed: 11 May 2020).

Delfino, D. (2020) *'What is Discord?': Everything you need to know about the popular group-chatting platform*, *Business Insider*. Business Insider. Available at: https://www.businessinsider.com/what-is-discord (Accessed: 15 April 2020).

Dependencies - First Steps - FastAPI (no date). Available at: https://fastapi.tiangolo.com/tutorial/dependencies/ (Accessed: 28 March 2020).

Deployment - FastAPI (no date). Available at: https://fastapi.tiangolo.com/deployment/ (Accessed: 13 May 2020).

Di Gregorio, F. and Varrazzo, D. (no date) *Psycopg – PostgreSQL database adapter for Python — Psycopg 2.8.6.dev0 documentation*. Available at: https://www.psycopg.org/docs/ (Accessed: 13 May 2020).

Discord — How People are Using Discord to Keep in Touch (no date) *Discord*. Available at: https://discordapp.com/why-discord (Accessed: 15 April 2020).

Dvergsdal, H. (no date) *HTTP – Store norske leksikon*, *Store norske leksikon*. Available at: https://snl.no/HTTP (Accessed: 13 May 2020).

FastAPI (no date). Available at: https://fastapi.tiangolo.com/ (Accessed: 6 May 2020).

Features - FastAPI (no date). Available at: https://fastapi.tiangolo.com/features/ (Accessed: 13 May 2020).

Features - IntelliJ IDEA (no date). Available at: https://www.jetbrains.com/idea/features/ (Accessed: 8 May 2020).

Features - PyCharm (no date). Available at: https://www.jetbrains.com/pycharm/features/ (Accessed: 8 May 2020).

Features - Rider (no date). Available at: https://www.jetbrains.com/rider/features/ (Accessed: 8 May 2020).

Features - SQLAlchemy (no date). Available at: https://www.sqlalchemy.org/features.html (Accessed: 18 April 2020).

Frequently Answered Questions | Open Source Initiative (no date). Available at: https://opensource.org/faq (Accessed: 29 April 2020).

GNU General Public License version 2 | Open Source Initiative (no date). Available at: https://opensource.org/licenses/GPL-2.0 (Accessed: 12 May 2020).

Goel, A. (2020) *10 Best JavaScript Frameworks to Use in 2020*, *Hackr.io*. Available at: https://hackr.io/blog/best-javascript-frameworks (Accessed: 13 May 2020).

Google Trends (no date) *Google Trends*. Available at: https://trends.google.com/trends/explore?q=fastapi,ASP.NET%20api,spring%20boot%20api (Accessed: 13 May 2020).

Gunicorn - Python WSGI HTTP Server for UNIX (no date). Available at: https://gunicorn.org/ (Accessed: 10 May 2020).

H2 Database Engine (no date). Available at: http://h2database.com/html/main.html (Accessed: 13 May 2020).

Hardt, D. (2012) 'The OAuth 2.0 Authorization Framework', *RFC 6749*, (2070-1721), p. 76.

Hardt, D. and Jones, M. (2012) 'The OAuth 2.0 Authorization Framework: Bearer Token Usage', p. 18.

Homepage - Reddit (no date). Available at: https://www.redditinc.com/ (Accessed: 14 May 2020).

Ian (2016) *The 3 Types of Relationships in Database Design | Database.Guide*. Available at: https://database.guide/the-3-types-of-relationships-in-database-design/ (Accessed: 13 May 2020).

IBM Cloud Education (2019) *Relational Databases*, *IBM Cloud*. Available at: https://www.ibm.com/cloud/learn/relational-databases (Accessed: 11 May 2020).

Introduction — Vue.js (no date). Available at: https://vuejs.org/v2/guide/ (Accessed: 13 May 2020).

JetBrains: Developer Tools for Professionals and Teams (no date) *JetBrains*. Available at: https://www.jetbrains.com/ (Accessed: 8 May 2020).

Kashlach, D. (2012) *Knit One Pearl Two: How to Use Variables in Different Thread Groups | BlazeMeter*, *Knit One Pearl Two: How to Use Variables in Different Thread Groups | BlazeMeter*. Available at: https://www.blazemeter.com/blog/knit-one-pearl-two-how-use-variables-different-thread-groups (Accessed: 7 May 2020).

Kumar, S. (2020) *python-dotenv*, *PyPI*. Available at: https://pypi.org/project/python-dotenv/ (Accessed: 13 May 2020).

Licenses & Standards | Open Source Initiative (no date). Available at: https://opensource.org/licenses (Accessed: 29 April 2020).

Lodderstedt, T. *et al.* (2019) 'OAuth 2.0 Security Best Current Practice'. Available at: https://tools.ietf.org/html/draft-ietf-oauth-security-topics-14 (Accessed: 29 March 2020).

Manifestet for smidig programvareutvikling (no date). Available at: https://agilemanifesto.org/iso/no/manifesto.html (Accessed: 8 March 2020).

Maven – Introduction (2020). Available at: https://maven.apache.org/what-is-maven.html (Accessed: 10 May 2020).

Med din mobil - BankID (no date). Available at: https://www.bankid.no/privat/bankid-pa-mobil/ (Accessed: 11 May 2020).

Microsoft (2020a) *Choose between .NET Core and .NET Framework for server apps*. Available at: https://docs.microsoft.com/en-us/dotnet/standard/choosing-core-framework-server (Accessed: 10 May 2020).

Microsoft (2020b) *Install .NET Core on Ubuntu 18.04 package manager - .NET Core*. Available at: https://docs.microsoft.com/en-us/dotnet/core/install/linux-package-manager-ubuntu-1804 (Accessed: 7 May 2020).

Microsoft Word - Word Processing Software | Office (no date). Available at: https://www.microsoft.com/en-ww/microsoft-365/word (Accessed: 14 May 2020).

Murphy, N. (2019) *Reddit's 2019 Year in Review*, *Upvoted*. Available at: https://redditblog.com/2019/12/04/reddits-2019-year-in-review/ (Accessed: 29 April 2020).

.NET API browser (no date). Available at: https://docs.microsoft.com/en-us/dotnet/api/ (Accessed: 13 May 2020).

Nienaber, C. and Suter, R. (2019) *ASP.NET Core Web API help pages with Swagger / OpenAPI*. Available at: https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger (Accessed: 13 May 2020).

Npgsql - .NET Access to PostgreSQL | Npgsql Documentation (no date). Available at: https://www.npgsql.org/ (Accessed: 13 May 2020).

npm | build amazing things (no date). Available at: https://www.npmjs.com/ (Accessed: 13 May 2020).

npm: axios (no date) *npm*. Available at: https://www.npmjs.com/package/axios (Accessed: 13 May 2020).

npm: nprogress (no date) *npm*. Available at: https://www.npmjs.com/package/nprogress (Accessed: 13 May 2020).

npm: qs (no date) *npm*. Available at: https://www.npmjs.com/package/qs (Accessed: 13 May 2020).

NuGet Gallery | Home (no date). Available at: https://www.nuget.org/ (Accessed: 13 May 2020).

Passlib 1.7.2 documentation — Passlib v1.7.2 Documentation (no date). Available at: https://passlib.readthedocs.io/en/stable/ (Accessed: 13 May 2020).

pckv (no date) *pckv/kompis*, *GitHub*. Available at: https://github.com/pckv/kompis (Accessed: 16 May 2020).

Pickett, W. (no date) *ASP.NET documentation*. Available at: https://docs.microsoft.com/en-us/aspnet/core/ (Accessed: 13 May 2020).

PostgreSQL: About (no date). Available at: https://www.postgresql.org/about/ (Accessed: 18 April 2020).

PostgreSQL JDBC About (no date). Available at: https://jdbc.postgresql.org/about/about.html (Accessed: 13 May 2020).

Project Lombok (no date). Available at: https://projectlombok.org/ (Accessed: 4 May 2020).

pydantic (no date). Available at: https://pydantic-docs.helpmanual.io/ (Accessed: 7 May 2020).

Python License (Python-2.0) | *Open Source Initiative* (no date). Available at: https://opensource.org/licenses/Python-2.0 (Accessed: 12 May 2020).

Python-Multipart — python-multipart 0.0.1 documentation (no date). Available at: http://andrew-d.github.io/python-multipart/ (Accessed: 13 May 2020).

Python Packaging Authority (2020) *pip - The Python Package Installer — pip 20.1 documentation*. Available at: https://pip.pypa.io/en/stable/ (Accessed: 13 May 2020).

Redocly (no date) *Redocly/redoc*, *GitHub*. Available at: https://github.com/Redocly/redoc (Accessed: 19 April 2020).

Requests: HTTP for Humans™ — Requests 2.23.0 documentation (no date). Available at: https://requests.readthedocs.io/en/master/ (Accessed: 5 May 2020).

REST API Documentation Tool | *Swagger UI* | *Swagger* (no date). Available at: https://swagger.io/tools/swagger-ui/ (Accessed: 19 April 2020).

REST – Statelessness – REST API Tutorial (no date). Available at: https://restfulapi.net/statelessness/ (Accessed: 13 May 2020).

Rikki (2018) *Change default endpoint in IdentityServer 4*, *Stack Overflow*. Available at: https://stackoverflow.com/questions/39186533/change-default-endpoint-in-identityserver-4 (Accessed: 11 May 2020).

Selivanov, Y. (2016) *uvloop: Blazing fast Python networking — magicstack*. Available at: https://magic.io/blog/uvloop-blazing-fast-python-networking/ (Accessed: 10 May 2020).

Simple OAuth2 with Password and Bearer - FastAPI (no date). Available at: https://fastapi.tiangolo.com/tutorial/security/simple-oauth2/ (Accessed: 13 May 2020).

Single Page Application (SPA) vs Multi Page Application (MPA) – Which and when to use? (2019) *ASPER BROTHERS*. Available at: https://asperbrothers.com/blog/spa-vs-mpa/ (Accessed: 14 May 2020).

Spring Boot (no date) *Spring*. Available at: https://spring.io/projects/spring-boot (Accessed: 6 May 2020).

Spring Boot Docs 2.2.6.RELEASE API (no date). Available at: https://docs.spring.io/spring-boot/docs/2.2.6.RELEASE/api/ (Accessed: 4 May 2020).

springdoc-openapi | *Library for OpenAPI 3 with spring-boot* (no date) *springdoc*. Available at: http://springdoc.org/ (Accessed: 13 May 2020).

spring-io (2020) *spring-io/initializr*, *GitHub*. Available at: https://github.com/spring-io/initializr (Accessed: 8 May 2020).

SQLite Home Page (no date). Available at: https://www.sqlite.org/index.html (Accessed: 14 May 2020).

SQL (Relational) Databases - FastAPI (no date). Available at: https://fastapi.tiangolo.com/tutorial/sql-databases (Accessed: 11 May 2020).

Statistic - Plugins | JetBrains (no date) *JetBrains Plugin Repository*. Available at: https://plugins.jetbrains.com/plugin/4509-statistic (Accessed: 14 May 2020).

Swashbuckle.AspNetCore 5.4.1 (no date). Available at: https://nuget.org/packages/Swashbuckle.AspNetCore/ (Accessed: 13 May 2020).

The 3-Clause BSD License | Open Source Initiative (no date). Available at: https://opensource.org/licenses/BSD-3-Clause (Accessed: 12 May 2020).

The MIT License | Open Source Initiative (no date). Available at: https://opensource.org/licenses/MIT (Accessed: 1 May 2020).

The Open Source Definition | Open Source Initiative (2007). Available at: https://opensource.org/osd (Accessed: 29 April 2020).

The PostgreSQL Licence (PostgreSQL) | Open Source Initiative (no date). Available at: https://opensource.org/licenses/postgresql (Accessed: 12 May 2020).

Tutorial - User Guide - Intro - FastAPI (no date). Available at: https://fastapi.tiangolo.com/tutorial/ (Accessed: 13 May 2020).

ujson (no date) *PyPI*. Available at: https://pypi.org/project/ujson/ (Accessed: 13 May 2020).

Uvicorn (no date). Available at: https://www.uvicorn.org/ (Accessed: 10 May 2020).

Verifying Apache Software Foundation Releases (no date). Available at: http://www.apache.org/info/verification.html (Accessed: 8 May 2020).

Vue.js - Help | PyCharm (2020). Available at: https://www.jetbrains.com/help/pycharm/vue-js.html (Accessed: 13 May 2020).

Webb, P. *et al.* (no date) *Spring Boot Reference Documentation*, *Spring*. Available at: https://docs.spring.io/spring-boot/docs/2.2.6.RELEASE/reference/html/ (Accessed: 4 May 2020).

Welcome to IdentityServer4 (latest) — IdentityServer4 1.0.0 documentation (no date). Available at: https://identityserver4.readthedocs.io/en/latest/ (Accessed: 13 May 2020).

Welcome to NGINX Wiki! | NGINX (no date). Available at: https://www.nginx.com/resources/wiki/ (Accessed: 13 May 2020).

Welcome to PyJWT — PyJWT 1.7.1 documentation (no date). Available at: https://pyjwt.readthedocs.io/en/latest/ (Accessed: 13 May 2020).

Welcome to Python.org (no date) *Python.org*. Available at: https://www.python.org/about/ (Accessed: 19 April 2020).

What is ASP.NET Core? A cross-platform web-development framework (no date) *Microsoft*. Available at: https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core (Accessed: 8 May 2020).

What is CSS? - Tutorialspoint (no date). Available at: https://www.tutorialspoint.com/css/what_is_css.htm (Accessed: 13 May 2020).

What is Python? Executive Summary (no date) *Python.org*. Available at: https://www.python.org/doc/essays/blurb/ (Accessed: 19 April 2020).

What is REST – Learn to create timeless REST APIs (no date). Available at: https://restfulapi.net/ (Accessed: 13 May 2020).

What is the OAuth 2.0 Password Grant Type? (2018) *Okta Developer*. Available at: https://developer.okta.com/blog/2018/06/29/what-is-the-oauth2-password-grant (Accessed: 30 March 2020).

Wikipedia contributors (2020a) *C Sharp (programming language)*, *Wikipedia*. Wikimedia Foundation, Inc. Available at: https://en.wikipedia.org/wiki/C_Sharp_(programming_language)?oldid=956210825 (Accessed: 13 May 2020).

Wikipedia contributors (2020b) *GitHub*, *Wikipedia*. Available at: https://en.wikipedia.org/wiki/GitHub?oldid=946242057 (Accessed: 25 March 2020).

Wikipedia contributors (2020c) *Google Docs*, *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Google_Docs?oldid=954697737 (Accessed: 4 May 2020).

Wikipedia contributors (2020d) *Object-oriented programming*, *Wikipedia*. Wikimedia Foundation, Inc. Available at: https://en.wikipedia.org/wiki/Object-oriented_programming?oldid=955499701 (Accessed: 13 May 2020).

Wikipedia contributors (2020e) *Object-relational mapping*, *Wikipedia*. Wikimedia Foundation, Inc. Available at: https://en.wikipedia.org/wiki/Object-relational_mapping?oldid=952535425 (Accessed: 11 May 2020).

Wikipedia contributors (2020f) *Software testing*, *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Software_testing?oldid=941710517 (Accessed: 20 February 2020).

Zelei, D. (2019) *zeldan/spring-boot-oauth2-password-flow*, *GitHub*. Available at: https://github.com/zeldan/spring-boot-oauth2-password-flow (Accessed: 13 May 2020).

Figurer

Vedlegg

I dette kapittelet ligger en beskrivelse på alle vedleggene til rapporten.

Vedlegg 1 - systemd service filer for API

Dette vedlegget inneholder alle systemd service filer som må lages for testmiljøet. Merk at stien til API-ene må byttes ut dersom git prosjektene ikke ble lagt i */home/repost/apis/* mappen. Disse filene skal legges i mappen *~/.config/systemd/user*.

DATABASE_PASSORD må erstattes med passordet for *repost* brukeren i PostgreSQL databasen som administrerer databasen.

SERVER_IP må erstattes med den offentlige IP-en til serveren. Dette er nødvendig for at CORS skal være tilgjengelig fra nettleseren når man er inne på nettsiden til front-end.

repost-fastapi.service

sti: `~/.config/systemd/user/repost-fastapi.service`

```
[Unit]
Description=Repost API in FastAPI

[Service]
WorkingDirectory=/home/repost/apis/repost-fastapi
ExecStart=/home/repost/apis/repost-fastapi/venv/bin/gunicorn repost:app -b
0.0.0.0:8000 -w 17 -k uvicorn.workers.UvicornWorker
Environment=PYTHONUNBUFFERED=1
Environment=REPOST_DATABASE_URL=postgresql://repost:DATABASE_PASSORD@127.0.
0.1:5432/repost_fastapi
Environment=REPOST_ORIGINS=http://SERVER_IP
Restart=on-failure

[Install]
WantedBy=default.target
```

repost-spring.service

sti: `~/.config/systemd/user/repost-spring.service`

Merk at *ExecStart* linjen er veldig lang, så man må muligens scrolle til høyre i tekstredigeringsverktøy for å finne **DATABASE_PASSORD** og **SERVER_IP**.

```
[Unit]
Description=Repost API in Spring Boot

[Service]
WorkingDirectory=/home/repost/apis/repost-spring
ExecStart=/usr/bin/java -Dspring.datasource.driverClassName= -
Dspring.jpa.database-platform= -
Dspring.datasource.url=jdbc:postgresql://localhost:5432/repost_spring -
Dspring.datasource.username=repost -
Dspring.datasource.password=DATABASE_PASSORD -
Drepost.origins[0]=http://SERVER_IP -jar /home/repost/apis/repost-
spring/target/repost-0.0.1-SNAPSHOT.jar
Restart=on-failure

[Install]
WantedBy=default.target
```

repost-aspnet.service

sti: `~/.config/systemd/user/repost-aspnet.service`

```
[Unit]
Description=Repost API in ASP.NET

[Service]
WorkingDirectory=/home/repost/apis/repost-aspnet
ExecStart=/home/repost/apis/repost-
aspnet/bin/Release/netcoreapp3.1/publish/RepostAspNet
Environment="ASPNETCORE_URLS=http://0.0.0.0:8002"
Environment="REPOST_DATABASE_CONNECTION_STRING=Host=localhost;Database=repo
st_aspnet;Username=repost;Password=DATABASE_PASSORD"
Environment="REPOST_ORIGINS=http://SERVER_IP"
Restart=on-failure

[Install]
WantedBy=default.target
```

Vedlegg 2 - Nginx konfigurasjonsfil

Dette vedlegget inneholder Nginx konfigurasjonsfilen for nettsiden. Denne fila kobles opp mot de filene i front-end etter bygging og er egnet til bruk i testmiljøet. Merk at stien til API-ene må byttes ut dersom git prosjektet til front-end ikke ble lagt i */home/repost/web/* mappen.

Filen lagres som `/etc/nginx/sites-available/repost`

```
server {
    listen 80;
    listen [::]:80;

    root /home/repost/web/repost-frontend/dist;
    index index.html;

    server_name _;

    location / {
        try_files $uri $uri/ /index.html;
    }
}
```

Vedlegg 3 - README filer

Dette vedlegget inneholder alle README filer til prosjektene. Ettersom de er åpen kildekode er README filene skrevet på engelsk.

FastAPI

https://github.com/pckv/repost-fastapi/blob/master/README.md

README.md

Repost FastAPI

This is the FastAPI implementation of the Repost API.

Installation

Python 3 must be installed and accessible through the use of a terminal and the keyword `python` or `python3`. Below are the steps for a proper setup using VENV (Python Virtual Environment).

1. Clone the repository

```
git clone https://github.com/pckv/repost-fastapi.git
```

2. Navigate to the `repost-fastapi` directory and create a new VENV

```
cd repost-fastapi
python -m venv venv
```

3 (**Linux**). Activate the venv (alternatively: run all commands after this step prefixed with `venv/bin/`)

```
source venv/bin/activate
```

3 (**Windows**). Activate the venv (alternatively: run all commands after this step prefixed with `venv\Scripts\`)

```
venv\Scripts\activate
```

4. Install the required packages

```
pip install -r requirements.txt
```

Configurations

Configurations are set by environment variables. Follow the instructions below to run the server once and a file `config.env` will be created in the root directory. Otherwise, the following settings can also be set using exported environment variables.

- **REPOST_CLIENT_ID** - The OAuth2 client_id. Default is `repost`
- **REPOST_JWT_SECRET** - The secret key used for JSON Web Tokens
- **REPOST_JWT_ALGORIGHTM** - The algorithm used for the key above
- **REPOST_DATABASE_URL** - An SQLAlchemy database url. See Engine Configuration
- **REPOST_ORIGINS** - A list of CORS URLs separated by `;`

Running the API with uvicorn

Uvicorn is a single-threaded ASGI server designed around uvloop to run fast. It is included in the requirements and should be used to run the API.

```
uvicorn repost:app
```

The default host and port is `localhost` and `8000`. They can be changed with the `--host` and `--port` arguments. To run the server publically, set the host to `0.0.0.0` like so.

```
uvicorn repost:app --host 0.0.0.0
```

Running the API with gunicorn

Gunicorn is a WSGI server that can manage multiple workers. Uvicorn has a worker for Gunicorn that can be used to run multiple Uvicorn workers. Since Repost is a stateless API, this works perfectly and will allow utilizing more processing power.

```
gunicorn repost:app -w 17 -k uvicorn.workers.UvicornWorker
```

The example above uses 17 workers for a system with 8 CPUs (16 threads + 1 workers). This value can be tweaked to your setup. You can also set the host and port in gunicorn with the `-b` argument, which includes both host and port in the same argument.

```
gunicorn repost:app -b 0.0.0.0:8000 -w 17 -k uvicorn.workers.UvicornWorker
```

Documentation

Documentation for the API is available after deployment at the `/api/swagger` and `/api/docs` endpoints.

Spring

https://github.com/EspenK/repost-spring/blob/master/README.md

README.md

Repost Spring

This is the Spring Boot implementation of the Repost API.

Installation

A version of the Java JDK must be installed. OpenJDK 11 was used when creating the project and is therefore recommended. Below are the steps for building and deploying the API. These instructions are only for a Linux build.

1. Clone the repository

```
git clone https://github.com/EspenK/repost-spring.git
```

2. Navigate to the `repost-spring` directory. Make the `mvnw` file executable. This is an instanced version of the Apache Maven build tool

```
cd repost-spring
chmod +x mvnw
```

3. Package a JAR file using the Maven tool. This will also install the required dependencies

```
./mvnw package
```

4. Copy the default configuration file to the root folder. See Configurations for more information

```
cp src/main/resources/application.properties user.properties
```

Configurations

Configurations are set using a properties file. See step 4 above for copying this file to the root directory of the project for further use.

- **server.port** - Change this to use a different port for the API
- **repost.signing-key** - The secret key used for JSON Web Tokens
- **repost.client_id** - The OAuth2 client_id
- **repost.origins[0]** - A CORS URL. Use `repost.origins[1]` etc for multiple allowed origins

Using PostgreSQL as a database

First add a `#` symbol before the properties `spring.datasource.driverClassName` and `spring.jpa.database-platform` as they are not needed for PostgreSQL. Set the appropriate database username and password using the `spring.datasource.username` and `spring.datasource.password` properties. Finally set the database connection URL with the `spring.datasource.url` property. Example:

```
#spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:postgresql://localhost:5432/DATABASE_NAME
spring.datasource.username=USERNAME
spring.datasource.password=PASSWORD
#spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

Running the API

The API can now be executed using a java runtime. Make sure to replace **ROOT_PATH** below with an absolute path to your cloned project directory.

```
java -jar target/repost-0.0.1-SNAPSHOT.jar --spring.config.location=file:///ROOT_PATH/user.properties
```

Documentation

Documentation for the API is available after deployment at the `/api/swagger` and `/api/docs` endpoints.

ASP.NET

https://github.com/pckv/repost-aspnet/blob/master/README.md

README.md

Repost ASP.NET

This is the ASP.NET Core Web APIs implementation of the Repost API.

Installation

.NET Core 3.1 is required.

1. Clone the repository

```
git clone https://github.com/pckv/repost-aspnet.git
```

2. Navigate to the `repost-aspnet` directory and publish the configuration

```
cd repost-aspnet
dotnet publish --configuration Release
```

Configurations

Configurations are set by environment variables. The following environment variables can be used for configuring during runtime.

- **REPOST_CLIENT_ID** - The OAuth2 client_id. Default is `repost`
- **REPOST_SIGNING_CREDENTIAL_PATH** - Path to a X509Certificate file used for JSON Web Tokens
- **ASPNETCORE_URLS** - URL the server should be hosted on. For a public server use `http://0.0.0.0:8002` (change port if needed)
- **REPOST_DATABASE_CONNECTION_STRING** - An Npgsql Connection String for a PostgreSQL database
- **REPOST_ORIGINS** - A list of CORS URLs separated by `;`

Running the API

The API can now be executed using the compiled binary found in the `bin` folder.

```
bin/Release/netcoreapp3.1/publish/RepostAspNet
```

Documentation

Documentation for the API is available after deployment at the `/api/swagger` and `/api/docs` endpoints.

Front-end

https://github.com/jonsondrem/repost-frontend/blob/master/README.md

README.md

Repost Front-end

This is the front-end Vue.js web server for Repost API.

Installation

Node.js and npm are required for building.

1. Clone the repository

```
git clone https://github.com/jonsondrem/repost-frontend.git
```

2. Navigate to the `repost-frontend` directory and install npm packages

```
cd repost-fastapi
npm install
```

3. Make changes to the .env file as needed. See Configuration for details

4. Build the project

```
npm run build
```

Configuration

Configurations are made with environment variables before building. The easiest is using the provided `.env` file for further configuration. The following environment variables can be used.

- **VUE_APP_CLIENT_ID** - The OAuth2 client_id
- **VUE_APP_APIS** - Name of all APIs separated by `;`
- **VUE_APP_API_0** - URL to the first API in the list defined in `VUE_APP_APIS`. Add more APIs by using `VUE_APP_API_1` etc

Distribution

Use your favorite web server to host the files in the `/dist` directory.

Testverktøyet

https://github.com/pckv/repost-apitest/blob/master/README.md

README.md

Repost API Test

This is a complete test for the Repost API. It's designed to run through every possible API endpoint and test all possible responses.

The test is designed to communicate with the API using HTTP calls. As such, to perform the test a deployed version of the API must be running. Every endpoint will be tested in an order that makes sense for resource dependencies. As such, the complete test will eventually create and test operations with all resources, until it finally deletes them all. All of the operations mentioned here are tested both positively and negatively.

Installation

Python 3 must be installed and accessible through the use of a terminal and the keyword `python` or `python3`. Below are the steps for a proper setup using VENV (Python Virtual Environment).

1. Clone the repository:

```
git clone https://github.com/pckv/repost-apitest.git
```

2. Navigate to the `repost-apitest` directory and create a new VENV:

```
cd repost-apitest
python -m venv venv
```

3 (**Linux**). Activate the venv (alternatively: run all commands after this step prefixed with `venv/bin/`)

```
source venv/bin/activate
```

3 (**Windows**). Activate the venv (alternatively: run all commands after this step prefixed with `venv\Scripts\`)

```
venv\Scripts\activate
```

4. Install the required packages

```
pip install -r requirements.txt
```

Running the test

To run the full API test, run the following command with the VENV activated (or use the prefix from step 3):

```
python main.py http://localhost:8000
```

Replace *localhost* and *port* with the address and port of your deployed API.

Running multiple tests for benchmarking

The test can be run more than once using the `--runs` argument. The result will then show stastitics of the tests, such as average test run time, standdard deviation etc.

NOTE: This is not a very good benchmark as it only runs the above test multiple times, which is not concurrent. The benchmarking is only included as a proof of concept, and might work better if multiple runs could be executed concurrently.

```
python main.py http://localhost:8000 --runs 100
```

Vedlegg 4 - Kildekode

Dette vedlegget inneholder kildekoden for de fem prosjektene vi har jobbet med: repost-apitest, repost-fastapi, repost-spring, repost-aspnet, repost-frontend. Kildekoden inkluderer hele git historien som ligger i .git mappen og kan brukes med git verktøyet.

GitHub repositories

FastAPI	https://github.com/pckv/repost-fastapi
Spring Boot	https://github.com/EspenK/repost-spring
ASP.NET	https://github.com/pckv/repost-aspnet
Front-end	https://github.com/jonsondrem/repost-frontend
APITest	https://github.com/pckv/repost-apitest

Vedlegg 5 - JMeter reports

Dette vedlegget inneholder resultatene fra JMeter testene som CSV fil og som JMeter Report Dashboard.

Vedlegg 6 - Testplaner

Dette vedlegget inneholder testplaner som er brukt i JMeter.

Testplanene ligger også på GitHub: https://github.com/EspenK/repost-testplan

Vedlegg 7 - Mail til veileder

Dette vedlegget inneholder alle mailer til veileder før møte med status på hva som var gjort og hva som skulle gjøres.

Vedlegg 8 - Forprosjektrapport

Dette vedlegget inneholder forprosjektrapporten.

Vedlegg 9 - OpenAPI filer

Dette vedlegget inneholder OpenAPI filer og eksportert visning av de i Swagger UI.

openapi.json filene kan leses av en OpenAPI støttet motor. Det finnes nettbaserte motorer til Swagger UI og Redoc hvor man kan lime inn en URL til en opplastet openapi.json fil.

Eksporterte PDF filer av Swagger UI starter med swagger-ui og viser API-et i sin helhet.

Vedlegg 10 - Logg av issues i GitHub

Dette vedlegget inneholder en logg av av alle issues og pull requests i GitHub. Dette viser hva og når oppgaver i de ulike prosjektene har blitt gjort.