Høgskulen på Vestlandet

BACHELOROPPGAVE

Automatisert klienttesting
Automated client testing

Henrik Engdal
Jan William Jensen

Fakultet for ingeniør- og naturvitenskap
Institutt for datateknologi, elektroteknologi og realfag
Dataingeniør

Veileder: Violet Ka I Pun
22.05.23

Jeg bekrefter at arbeidet er selvstendig utarbeidet, og at referanser/kildehenvisninger til alle kilder som er brukt i arbeidet er oppgitt, jf. Forskrift om studium og eksamen ved Høgskulen på Vestlandet, § 12-1.

Høgskulen på Vestlandet

Fakultetet for ingeniør- og naturvitskap
Institutt for datateknologi, elektroteknologi og realfag

TITTELSIDE FOR HOVEDOPPGAVE

Rapportens tittel: Automatisert klienttesting Automated client testing	**Dato:** 22.05.23
Forfatter(e): Henrik Engdal, Jan William Jensen	**Antall sider u/vedlegg:** 55 **Antall sider vedlegg:** 71
Studieretning: Dataingeniør	**Antall disketter/CD-er:** 0
Kontaktperson ved studieretning: Violet Ka I Pun	**Gradering:** Ingen

Oppdragsgiver: Uni Micro AS	**Oppdragsgivers referanse:** Ingen
Oppdragsgivers kontaktperson: Geir Bjellebø Kvam	**Telefon:** +47 453 93 348

Sammendrag:

Prosjektet omhandler utvikling for Uni Micro AS, et selskap som utvikler teknologi og programvare for økonomi- og regnskapstjenester. Løsningen utviklet i prosjektet erstatter manuelle tester knyttet til regresjonstesting av deres klienter. Løsningen benytter seg av TestCafe for automatisering av brukertesting via nettleser og er delt inn i en frontend og en backend.

This project covers software development for Uni Micro As, a company that develops technology and software for financial and accounting services. The solution developed in this project replaces manual testing that is done in regression tests for their clients. The solution utilizes TestCafe for automated, browser-based user testing and is divided into a frontend and a backend.

Stikkord:

Automatisert testing	Angular - TypeScript	TestCafe

Høgskulen på Vestlandet, Fakultetet for ingeniør- og naturvitenskap
Postadresse: Postboks 7030, 5020 BERGEN   Besøksadresse: Inndalsveien 28, Bergen
Tlf: 55 58 75 00   Fax 55 58 77 90   E-post: post@hvl.no   Hjemmeside: http://www.hvl.no

Forord

Dette prosjektet er Bacheloroppgaven som er skrevet i sammenheng med Dataingeniør utdanningen ved Høgskulen på Vestlandet. Rapporten og løsningen er skrevet av Henrik Engdal og Jan William Jensen gjennom våren 2023 i perioden 02.01.23 - 22.05.23

Ideen bak oppgaven ble først diskutert etter arbeidet som var utført gjennom sommerjobben til Jan William Jensen i Uni Micro hvor tester i TestCafe ble prøvet ut. Med positive resultater kom ideene om mer automatisering fra start til slutt. Ønsket om å kunne kjøre tester i klienten for alle regressjonstester ledet til oppgaven.

Vi ønsker å takke oppdragsgiver for en spennende, læringsrik og relevant oppgave. Vi ønsker også å takke Geir Bjellebø Kvam for støtten og kontinuerlig tilbakemelding i arbeidet som har vært god veiledning for å finne gode løsninger på utfordringene vi har støtt på gjennom prosjektet. Til slutt ønsker vi takke Violet Ka I Pun, veileder ved instituttet, for mange gode tilbakemeldinger og konstruktiv kritikk på rapporten.

Innholdsfortegnelse

1 INNLEDNING **1**

1.1 Kontekst . . . 1
1.2 Motivasjon . . . 1
1.2.1 Prosjekteier . . . 1
1.3 Problembeskrivelse og mål . . . 2
1.4 Oppbygging av rapporten . . . 2

2 PROSJEKTBESKRIVELSE **3**

2.1 Praktisk bakgrunn . . . 3
2.1.1 Tidligere arbeid . . . 3
2.1.2 Initielle krav . . . 4
2.1.3 Initiell løsnings-idé . . . 5
2.2 Avgrensninger . . . 5
2.3 Ressurser . . . 6
2.4 Litteratur om problemstillingen . . . 6
2.4.1 Enhetstesting . . . 7
2.4.2 Integrasjonstesting . . . 8
2.4.3 Regresjonstesting . . . 8
2.4.4 Systemtesting . . . 8
2.4.5 Klienttesting . . . 8

3 DESIGN AV PROSJEKTET **9**

3.1 Forslag til løsning . . . 9
3.2 Alternativer for rapporteringsplugin . . . 9
3.2.1 Spec . . . 9
3.2.2 JSON . . . 9
3.2.3 xUnit . . . 9
3.2.4 Egenutviklet plugin . . . 10
3.3 Alternativer for arkitektur i SoftRigTestTool backend . . . 10
3.3.1 Monolith . . . 10
3.3.2 Delt mikrotjeneste . . . 10
3.4 Alternativer for arkitektur i SoftRigTestTool frontend . . . 10
3.4.1 Angular . . . 11
3.4.2 React . . . 11
3.5 Diskusjon av alternativene . . . 11
3.5.1 Rapporteringsplugin . . . 11
3.5.2 SoftRigTestTool backend arkitektur . . . 11
3.5.3 SoftRigTestTool frontend arkitektur . . . 12
3.6 Valgt løsning . . . 12
3.7 Valg av verktøy og språk . . . 13

3.7.1 Språk og rammeverk . . . 13
3.7.2 Verktøy . . . 13
3.7.3 Plattform . . . 14
3.7.4 Samarbeidsverktøy . . . 14
3.8 Prosjektmetodikk . . . 14
3.8.1 Utviklingsmetodikk . . . 14
3.8.2 Diskusjon av utviklingsmetodikk . . . 17
3.8.3 Valg av utviklingsmetodikk . . . 17
3.8.4 Prosjektplan . . . 18
3.8.5 Risikovurdering . . . 20
3.9 Evalueringsplan . . . 22

4 DESIGN OG UTVIKLING **23**
4.1 Bakgrunn for valgt design . . . 24
4.2 TestCafe . . . 24
4.3 Dataoverføring . . . 25
4.3.1 HTTPs . . . 25
4.3.2 SignalR . . . 26
4.4 Backend . . . 26
4.4.1 Separation of Concerns (SoC) . . . 27
4.4.2 Endepunkter . . . 28
4.5 Dashbord . . . 29
4.5.1 Ag-Grid . . . 30
4.5.2 Chart.js . . . 31
4.6 Database . . . 32
4.6.1 Entity Framework Core . . . 33
4.6.2 SoftrigUITest . . . 33

5 RESULTATER **35**
5.1 Evalueringsmetode . . . 35
5.1.1 Kontinuerlig evaluering . . . 35
5.1.2 Systemtesting . . . 35
5.1.3 Brukertesting . . . 35
5.1.4 Sluttevaluering . . . 36
5.2 Evalueringsresultat . . . 36
5.2.1 Resultat av kontinuerlig evaluering . . . 36
5.2.2 Resultat av systemtesting . . . 36
5.2.3 Resultat av brukertest . . . 37
5.2.4 Resultat av sluttevaluering . . . 38
5.3 Prosjektresultat . . . 39
5.3.1 Frontend . . . 39
5.3.2 Backend . . . 45

5.4 Prosjektgjennomføring 46

6 DISKUSJON 47

6.1 Sluttprodukt og fremgangsmåte 47

6.2 Konsekvenser 47

6.2.1 Konsekvens av valgt utviklingsverktøy 47

6.2.2 Konsekvens av valgt rapporteringsløsning 48

6.2.3 Konsekvens av valgt utviklingsmetodikk 48

6.2.4 Utenforliggende konsekvenser 49

6.3 Forbedringer 49

7 KONKLUSJON OG VIDERE ARBEID 51

7.1 Konklusjon 51

7.2 Videre arbeid 51

8 REFERANSER 53

9 VEDLEGG 55

Ordliste

Application Programming Interface (API)	Et programmeringsgrensesnitt for funksjonalitet som er synlig for andre programmer, også kalt endepunkter
Backend	Koden som kjører servertjenesten av et system som har kontakt med database og har all forretningslogikk
C-nummer	Case nummer som kommer fra TestRail. Dette brukes som ID for å starte tester og finne riktig fil i testmappen
CRUD	Create, Read, Update, Delete - Operasjoner på objekter i en database
CI/CD	Continuous Integration and Continuous Delivery - Kontinuerlig integrasjon og kontinuerlig leveranse
Connection String	En streng som inneholder tilkoblingsinformasjon til en datakilde
Controller	En klasse som styrer hva som blir sendt til det grafiske grensesnittet. Tar imot forespørsler, dirigerer til Service-klasser, og responderer med svar
ControllerBase	Klasse som inneholder logikk og metoder for å returnere HTTP statuskoder med meldinger og objekter
Dependency injection	Et design mønster som tilgjengeliggjør objekter for klasser som er avhengig av andre klasser for å fungere
DOM	Document Object Model - Datamodell og API for HTML som har en trestruktur
Frontend	Koden som kjører grensesnittet som brukeren benytter for å gjennomføre aktiviteter i programmet
GUI	Graphical user interface - Et Grafisk brukergrensesnitt som brukeren benytter for å gjennomføre aktiviteter i programmet
HTTP	Hypertext Transfer Protocol – HTTP er en protokoll for å overføre data mellom systemer på et nettverk.
ID	Identifikasjon - En verifikasjon på sin identitet
Interface	En kontrakt som forteller klasser som implementerer kontrakten hvilken metoder og attributter som skal implementeres
JSON	JavaScript Object Notation - Et tekstformat som representerer et objekt som kan sendes over nett og brukes mellom programmeringsspråk
Model	Modellen som inneholder definisjonen på dataobjektene som systemet benytter seg av
Monolith	En programstruktur hvor hele systemet er en enhet
MVC	Model View Controller - Et design prinsipp som deler ansvar for grensesnitt og data

Namespace	Et navneområde som inneholder klasser som f.eks. ControllerBase. Fungerer som en mappestruktur
Plugin	En komponent som legger til en spesifikk funksjonalitet i et program.
Regex	Regular expression - et utrykk som beskriver en eller flere strenger basert på syntaks og kan matche mot tekst
RESTful API	Representational Entity State Transfer - En struktur stil som følger prinsipper slik at et objekts tilstand representeres
Request	En forespørsel som skjer fra et system til et annet. En forespørsel vil som oftest forvente en respons tilbake
Response	En respons er et svar fra et system som har mottatt en forespørsel
Service collection	En klasse som indekserer alle registrerte klasser for bruk i dependency injection
Service	En klasse som inneholder forretningslogikk og håndterer transaksjoner mot databasen
TCP	Transmission Control Protocol - En av hoved protokollene på internett som oppretter kobling mellom server og klient før data sendes. TCP er en pålitelig måte å sende data, men øker ventetiden
UI	User Interface - Brukergrensesnitt
URL	Uniform Resource Locator - nett adresse som leder til en spesifikk ressurs
UX	User Experience - Brukeropplevelse
View	Template sider som forteller maskinen hvor elementer skal plasseres på skjermen
XML	The Extensible Markup Language - Et markerings språk og fil format for lagring og sending av data

1 INNLEDNING

1.1 Kontekst

Uni Micro er et teknologiselskap som utvikler og leverer forretningslogikk som en tjeneste til sine kunder. De utvikler hovedsakelig løsninger for regnskap, men det finnes også løsninger for kontraktsarbeid, fravær, lønn og det meste som omhandler økonomien til et selskap. Uni Micro utvikler og drifter klienter for sine to største kunder, DNB og SpareBank 1 konsernet, i tillegg til sine egne. Til sammen er det 16 klienter (14 klienter for SpareBank 1 konsernet) som skal opprettholdes til enhver tid. Ved nye versjoner må disse sidene testes av Q&A (”Quality & Assurance”) hvor det verifiseres at all funksjonalitet fungerer som planlagt. Q&A vil da gjennomføre forskjellige tester i klienten som å opprette faktura, bilag, ansatte, legge til ferie, og så videre. Når en test er fullført legges det inn et resultat i TestRail. TestRail er nettsiden hvor oppdragsgiver har lagret oppskriften på alle tester som gjennomføres manuelt og resultatet fra testene som blir gjennomført. De automatiserte testen skal hentes fra et oppbevaringssted på nett slik at det til enhver tid brukes oppdaterte tester.

1.2 Motivasjon

I dag blir all testing i *frontend* hos Uni Micro gjort manuelt av testere. I *backend* er alle tester automatisert og består av enhetstester og integrasjonstester. Enhetstester er enkle tester som tester en “enhet” for hver test. Integrasjonstester tester at disse enhetene fungerer som ønsket når de kobles sammen. Det er per i dag ikke implementert tester som verifiserer funksjonalitet og utseende til hver klient. Da Uni Micros frontend ble utviklet ble det ikke laget enhetstester og integrasjonstester. Uni Micro beskriver grunnen som at det er mer komplekst enn for backend og at da frontend ble startet var rammeverket ikke optimalt. Dette er noe de ønsker å gjøre noe med for en nyere versjon av frontend. Det er av denne grunn stor motivasjon for et verktøy som automatiserer regresjonstester for å kutte kostnader og heve kvalitet på sine produkter. Regresjonstester er tester som må gjøres før hver versjon lanseres, og muliggjør en økt frekvensen av nye versjoner som kan lanseres. Det er ønskelig å benytte et verktøy som kan navigere rundt på siden, skrive og verifisere tekst slik at testerne kan bruke tiden på andre tester.

1.2.1 Prosjekteier

Prosjektets eier er Uni Micro AS, et selskap som ble startet i Modalen 1986 da Hans Jørgen Neset lagde et økonomisystem som skulle forenkle arbeidsdagen til små- og mellomstore bedrifter. I dag er Uni Micro en av de fremste økonomisystemene i Norge og har de siste årene blitt kjøpt opp av DNB og SpareBank 1.

1.3 Problembeskrivelse og mål

I dette prosjektet skal det videreutvikles en løsning som håndterer automatiserte tester og rapporterer resultatet av testene. Løsningen utvikles med et ønske fra Uni Micro om å effektivisere deres prosess knyttet til regresjonstesting av deres klienter. Dette gjennomføres også med tanke på å gjennomføre økt testing og dermed heve kvaliteten på produktene sine.

Løsningen har fra tidligere et konseptbevis utviklet av Jan William Jensen gjennom sin stilling hos Uni Micro. Konseptbeviset kan kjøre en test og få en rapport tilbake med enkel statistikk. Videreutviklingen av løsningen vil innebære å utvikle funksjonalitet for å kjøre flere tester sammen i én spørring, håndtere flere spørringer samtidig, rapportering av resultater og opprette testresultat. Prosjektets spesifikke mål er å automatisere kjøring av regresjonstester, lage rapporter fra resultater, presentere det til brukeren samt vise trender for testene.

1.4 Oppbygging av rapporten

Rapporten er delt opp i syv kapitler: kapittel 2 vil inneholde litt om bakgrunnen for prosjektet, ideene rundt løsningen, avgrensninger og ressurser. Kapittel 3 vil komme med flere alternativer og forslag til løsningens utforming. I tillegg beskrives ulike verktøy og metoder som tas i bruk for å gjennomføre prosjektet. Kapittel 4 omhandler design, utformingen og utvikling av systemet. Kapittel 5 vil gå gjennom prosjektets evalueringsmetoder og resultater. I kapittel 6 drøftes resultatene, problemstillingen og løsningen som er utarbeidet. Til slutt, i kapittel 7, trekkes alt sammen til en konklusjon og det blir presentert ideer og spørsmål til videre arbeid.

2 PROSJEKTBESKRIVELSE

2.1 Praktisk bakgrunn

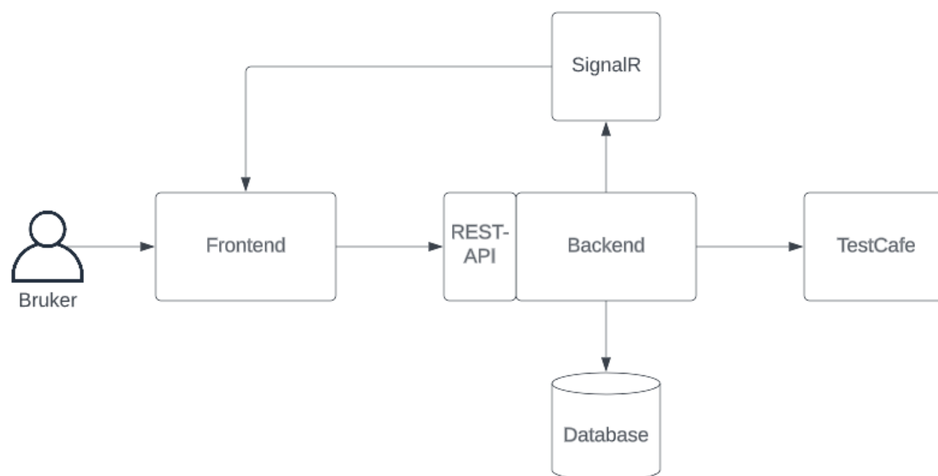
2.1.1 Tidligere arbeid

Det er som tidligere nevnt i kapittel 1.3 laget et konseptbevis som kan kjøre en test og rapportere tilbake rådata. For å kjøre testene brukes det en pakke med navn TestCafe til å automatisere operasjoner i nettleser. I figur 2.1 vises et eksempel på rådata som blir skrevet i terminalen etter at en test er kjørt. Dette blir returnert til backend i form av en liste med strenger. Strengene inneholder informasjon om testnavn, kjøretid, feilmelding og resultat. Dette ble deretter tolket og lagret i databasen. Prosessen av å tolke disse resultatene har tidligere opplevdes som svært tungvint fordi strengene må tolkes med hjelp av *Regex*. Regex er et verktøy for mønstergjenkjenning ved å matche mønstre i strenger mot et ønsket uttrykk.

```
Running tests in:
 - Firefox 113.0 / Windows 11

 New Fixt
 × C58050 - Ny basert på

   1) AssertionError: expected '' to deeply equal 'jljl'

      + expected - actual

      -''
      +jljl


      Browser: Firefox 113.0 / Windows 10

         204 |            await t.click(Selector('input').withAttribute('placeholder', 'Velg kunde'))
         205 |            await util.createNewCustomer(customerName)
         206 |        }
         207 |        await t.expect(name).eql(customerName);
         208 |        //Selector is targeting the input field that "Deres referance" is refering to.
       > 209 |        await t.expect(Selector('span').withText('Deres referanse').sibling(0).child().child('input').value).eql(buyerRef)
         210 |        //Selector is targeting the input field that "E-postadresse" is refering to.
         211 |        await t.expect(Selector('span').withText('E-postadresse').sibling(0).child('input').value).eql(mailAdress)
         212 |        for (let index = 0; index < productList.length; index++) {
         213 |            await t.expect(Selector('span').withText(productList[0]).innerText).eql(productList[0])
         214 |        }

         at <anonymous> (C:\Users\janwill\Documents\GitHub\SoftrigUITest\Tests\Sale\Orders.js:209:110)
         at asyncGeneratorStep (C:\Users\janwill\Documents\GitHub\SoftrigUITest\Tests\Sale\Orders.js:3:39)
         at _next (C:\Users\janwill\Documents\GitHub\SoftrigUITest\Tests\Sale\Orders.js:3:39)



 1/1 failed (26s)
```

Figur 2.1: Eksempel på rådata som returneres fra en test kjørt av TestCafe.

Konseptbevisets arkitektur vist i figur 2.2 er en todelt løsning bestående av en frontend og end backend med et *API* basert på *RESTful*.Backend starter tester og maskinen gjennomfører tester i bakgrunnen. SignalR brukes for å oppdatere visning av tester som kjører i frontend når tester starter og stopper. Kommunikasjon med database er gjort ved hjelp av Entity Framework Core (EF Core). Frontend kan hente alle resultater fra databasen, men uten noe filtrering.



Figur 2.2: Arkitektur brukt i konseptbeviset.

2.1.2 Initielle krav

Produktet som skal utvikles vil ta utgangspunkt i konseptbeviset og benytter derfor samme arkitektur. Produktet vil av den grunn være todelt med en frontend og en backend del. Data som genereres av produktet vil benytte seg av Microsoft sitt databasesystem, SQL Server, som er oppdragsgiverens foretrukne valg. Produktet skal inneholde et brukergrensesnitt i form av et grafisk brukergrensesnitt (*GUI*) som kommuniserer med APIet. Løsningen som skal utvikles gjennom prosjektet kommer også til å benytte de samme utviklingsspråkene og rammeverkene som konseptbeviset. Dette er .NET Core, Angular og SQL Server.

Backend

Det skal være mulig å kjøre en enkelt test, en kategori, alle tester, lage rapporter og lage statistikk på alle kjørte tester. Testene er skrevet i JavaScript og kjøres av TestCafe. Backend er tiltenkt å ha mulighet til å kjøre test fra ulike programmer, men foreløpig er det kun TestCafe (Se vedlegg B, Visjonsdokumentet kapittel 3 for mulige programmer som backend kan ekspanderes til). Det er ikke satt krav til hvordan rapporter blir generert i forhold til format. Det er også ønskelig, dersom det blir tid, å legge til rette for å kunne kjøre et utvalg av tester, gjennom egendefinerte lister og velge hvor mange ganger testene skal kjøres.

Frontend

Dashbordet vil være den delen av prosjektet som lar brukeren samhandle med APIet. Det skal være mulig å nå alle relevante endepunktene som er beskrevet i APIet, som å vise en test basert på *ID*, hente alle resultatene til en test ved hjelp av ID, hente alle resultater, vise enkelt rapporter og statistikk på kjøretid. Dersom det blir implementert funksjonalitet utenfor krav til backend må dashbordet også tilrettelegges for dette. Om

tiden strekker til er det også ønskelig at dashbordet skal ha funksjonalitet for å spesifisere hvor mange ganger testene skal kjøres og mulighet for å kjøre en egendefinert liste med tester.

2.1.3 Initiell løsnings-idé

Løsningen som skal utvikles vil få navnet SoftRigTestTool etter Uni Micros regnskapsplattform SoftRig. SoftRigTestTool skal kunne laste inn data og kjøre testskript på de ulike klientene ved hjelp av TestCafe. Etter testene er blitt kjørt vil backend lage en rapport til hver test. Alle rapportene blir lagret i en database. Tjenesten vil også returnere rapportene som blir generert tilbake til frontend. Hvis testen feiler vil rapporten inneholde en feilmelding med informasjon om hva som feilet i testen og hvor dette befinner seg i koden til testen. En utvikler vil da kunne lese rapporten og sjekke gjennom hvorfor testen feiler og fikse koden som ble testet. Det er også diskutert mulighetene for å supplere testmiljøet med data direkte fra kundenes miljø slik at man kan reprodusere feil som blir meldt inn av brukerne.

Med dette er det satt opp seks brukstilfeller for løsningen: kjøre tester, hente resultat, vise resultat rapport, opprette statistikk, registrere tester og slette tester. Full oversikt over brukstilfeller med brukstilfellediagram kan sees i Kravdokument kapittel 2 (vedlegg A).

2.2 Avgrensninger

Prosjektet har begrensninger i forhold til valg av teknologier, integrasjoner og tid. Produktet som skal utvikles for Uni Micro vil benytte seg av samme teknologier som selskapet bruker i sine produkter. Det vil derfor bli benyttet C# og .NET for backend. For frontend har prosjektgruppen derimot flere alternativer for både språk og rammeverk. Uni Micro bruker SQL Server som databasesystem, og vil også bli brukt i prosjektets løsning.

For backend finnes det forskjellige .NET muligheter. .NET Framework og .NET Core er to av alternativene, men Microsoft anbefaler å bruke .NET Core (Microsoft, 2022). I .NET Core vil det være ønskelig å velge en nyere versjon for bedre ytelse, stabilitet og sikkerhet. Konseptbeviset ble skrevet i .NET Core 6, men det finnes i dag også .NET Core 7. Her vil det være rom for å kunne oppgradere konseptbeviset til den nyeste versjonen.

Det er også begrensninger med integrasjoner mot TestRail og Azure. For at dette skal være mulig må det opprettes en ID til produktet. Uten en slik ID vil TestRail systemet nekte produktet adgang til data. Azure er et nettsted hvor data kan lagres og distribueres ut til systemer som har tilgang. Tidsbegrensninger må også tas hensyn til. Prosjektgruppen består av to personer med begrenset erfaring. Det blir tatt hensyn til

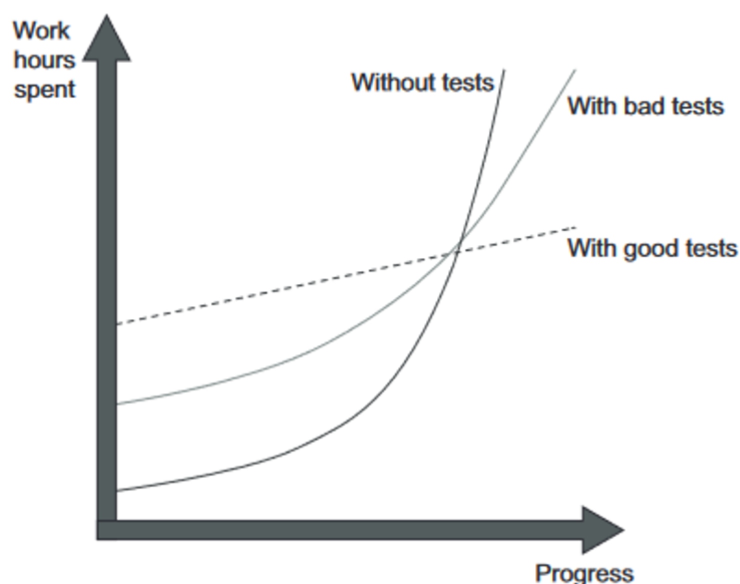
at det må gjennomføres kompetanseheving i teknologiene og rammeverkene som skal benyttes. Det blir ikke tid til å integrere produktet med SoftRig, som er Uni Micros regnskapsplattform, hvor data kan lastes inn til klienten som det testes i.

2.3 Ressurser

Oppdragsgiver har tilgjengeliggjort flere ressurser for prosjektet. Det tidligere utviklede konseptbeviset og all tilknyttet kode er tilgjengelig for bruk og videreutvikling. Klienttestene som skal benyttes i backend vil være tilgjengelig både lokalt på maskinen og i skyen gjennom GitHub/Azure. Prosjektet skal også ta i bruk oppdragsgivers utviklingsplattform Visual Studio og Microsoft SQL Server Management Studio. All kode som skrives vil også bli lastet opp til oppdragsgivers GitHub repository. Kontorplasser ved hovedkontoret i Bergen, samt låne-PC er også blitt gjort tilgjengelig ved behov. Oppdragsgiver har også tilgjengeliggjort Q&A testere for brukertesting. Avslutningsvis vil Geir Bjellebø Kvam i sin stilling som "Head of DevOps" i Uni Micro fungere som ekstern veileder for prosjektet, i tillegg til Violet Ka I Pun som internveileder hos HVL.

2.4 Litteratur om problemstillingen

Testing av kode og funksjonalitet er et tema som er mye diskutert av utviklere. Det finnes mange ulike type tester som har forskjellige formål og utforming. Manuelle tester er en tidkrevende prosess og utføres ved at en person klikker, skriver og leser for å verifisere at programmet fungerer som tiltenkt. Automatiserte tester vil gjennomføre de samme stegene som en person og rapportere tilbake med resultatet. Figur 2.3 illustrerer tidkompleksitet for et system med forskjellig kvalitet av tester.



Figur 2.3: Systemets kompleksitet mot tid det tar å teste manuelt.

Det er skrevet en del litteratur om testing hvor det er beskrevet hvordan en god test er utformet. Når systemets størrelse øker vil effekten av tester også øke. Et stort system vil ha stor effekt av tester i motsetning til manuell testing. Når det skal verifiseres at systemet fungerer som ønsket vil et stort system med automatiserte tester kunne kjøres av maskinen på minutter. Dagens maskiner har en klokkefrekvens som oppgis i GHz (Giga Hertz) som tillater dem å utføre milliarder av instruksjoner per sekund. Et klikk på skjermen tilsvarer flere instruksjoner og det er derfor ikke et en-til-en forhold mellom dem.

Overordnet for alle tester er det flere prinsipper som er ønskelig å følge. Tester har ulike faser: setup, exercise, verify, teardown (Croak, u.å). Disse begrepene brukes for å beskrive de ulike fasene. I setup blir nødvendige deler av systemet koblet til testen, instansiert og nødvendig data tilegnet variabler. I exercise kjøres den del av koden som skal testes med delene som ble opprettet i setup. Resultatet som kommer tilbake fra utførelsen blir så verifisert i verify. Til slutt kjøres det en prosess som kalles teardown hvor alt som er opprettet av programmet blir slettet. Den siste fasen er ofte automatisk og noe en utvikler ikke trenger å tenke på.

2.4.1 Enhetstesting

Enhetstester er små raske tester som verifiserer en enhet med tester som beskrevet tidligere. For enhetstester er det vanlig å kalle disse stegene for Arrange, Act og Assert. Figur 2.4 viser at under Arrange blir forskjellige tall tilegnet variabler og en ny bankkonto blir opprettet med navn og start balanse. Deretter, under Act, kjøres det en metode med navn Debit som tar med debitAmount. Til slutt i Assert tilegnes kontoens balanse til actual og dette blir sjekket mot forventet verdi. 0.001 er feilmargin og teksten "Account not debited correctly" vil være feilmeldingen som vises om testen feiler (Microsoft, u.å-e).

```
C#                                                          Copy

[TestMethod]
public void Debit_WithValidAmount_UpdatesBalance()
{
    // Arrange
    double beginningBalance = 11.99;
    double debitAmount = 4.55;
    double expected = 7.44;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // Act
    account.Debit(debitAmount);

    // Assert
    double actual = account.Balance;
    Assert.AreEqual(expected, actual, 0.001, "Account not debited correctly");
}
```

Figur 2.4: Eksempel på en enkel enhetstest (Microsoft, u.å-e).

2.4.2 Integrasjonstesting

Integrasjonstester er større enn enhetstester og tester flere deler av programmet i samme test. I en slik test vil det være vanlig å teste hele løpet fra data blir sendt inn i systemet til operasjonen er utført, som beskrevet i (Hamilton, u.å-a). Dette gjøres vanligvis ved at testen sender en spørring til seg selv som den utfører. En av utfordringene med å gjøre dette er at det genereres data som ikke ønskes lagret i produksjonsdatabasen, hvor det finnes data fra brukerne. Løsningen er å bruke Mock som gjør det mulig å løsrive komponenter fra andre deler av systemet man ikke ønsker å teste. Løsrivningen gjøres ved å spesifisere hva som skal returneres fra de ulike metodene som testen er avhengig av, men som kommer fra en annen klasse enn det som testes. Dette blir mer og mer nødvendig når systemer vokser og data henger tett sammen.

2.4.3 Regresjonstesting

Regresjonstester gjennomføres for å forsikre at endringer i programmet ikke har skapt nye feil eller endret ønsket funksjonalitet. Kodeendringer kan påvirke andre deler av systemet enn tiltenkt og det er ikke alltid lett å teste med enhetstester eller integrasjonstester. Regresjonstester vil teste samhandling mellom forskjellige deler av systemet. Når regresjonstester gjennomføres er det mulig å bruke automatiserte tester, men manuelle tester er også en viktig del (Hamilton, u.å-b).

2.4.4 Systemtesting

Systemtesting er en ende til ende test hvor hele systemer testes. Det finnes flere type systemtester som brukertester, belastningstester, regresjonstester og gjenopprettingstester. Brukertester er tester hvor en brukere navigere rundt i systemet for å teste funksjonalitet og brukervennlighet. Belastningstester sjekker hvordan systemet håndterer belastning når mange bruker systemet samtidig. En slik sjekk kan gjøres ved å sende mange spørringer til systemet på en gang. Regresjonstester er tester som utføres for å sjekke at endringer og nye deler fungerer som planlagt. Gjenopprettingstester verifiserer at systemet er robust og takler krasj uten å miste data (Hamilton, u.å-c).

2.4.5 Klienttesting

Klienttesting er tester som skjer på brukerens maskin for å teste *UI/UX*. Fokuset i slike tester ligger i funksjonalitet og utseende i brukergrensesnittet når brukeren gjør forskjellige handlinger (Split, u.å). Dagens nettsider skal være hurtige, smarte og enkle. Det finnes snarveier og logikk som hjelper brukeren med aktivitetene som gjøres, og hvis denne funksjonaliteten ikke fungerer som planlagt blir det et hinder for brukeren i stedet. Det er nesten umulig å se for seg alle rekkefølgene brukerne kan utføre operasjoner til en aktivitet, og det kan fremprovosere feil. Stil-tester verifiserer at utseende til siden er som forventet etter en operasjon og layout-tester som sjekker at sidens oppsett er som forventet.

3 DESIGN AV PROSJEKTET

3.1 Forslag til løsning

Før oppstart av prosjektet var det allerede lagt til grunn en god del valg. Løsningen vil som nevnt i kapittel 2.1.2 være delt opp i en backend, skrevet i .NET 7, og en frontend som er skrevet i Angular. For datalagring blir det benyttet SQL Server. APIets konseptbevis er bygget som et RESTful API som eksponerer endepunkter til klienten og bruker *HTTP-protokollen* for å returnere data. konseptbeviset, omdiskutert i kapittel 2.1.1, brukte TestCafe sin standard rapporteringsplugin kalt Spec. Denne oppfattes tidvis som tungvint, og gir grunnlag for å utforske alternativer som kan effektivisere prosessen. Her er det rom for å se på alternative løsninger for å effektivisere prosessen med å tolke resultater. Prosjektgruppen har også flere løsningsalternativer når det kommer til både utforming av backend og frontend.

Løsningen vil etter planen kunne teste hvilken som helst type klient så lenge testene som er skrevet for denne typen klient eksisterer. Løsningen starter testene og tar imot resultatet. Løsningen inneholder ikke funksjon for gjennomføring av testene i seg selv.

3.2 Alternativer for rapporteringsplugin

Prosjektet har diskutert mulige løsninger av rapporteringsplugin. Dette verktøyet er en *plugin* som ligger på TestCafe pakken og formaterer svarene som TestCafe gir etter at tester er gjennomført. I dokumentasjonen til TestCafe ligger det standardformater som kommer med TestCafe og mulige alternativer til andre utvikleres implementasjoner.

3.2.1 Spec

Spec er tilleggsmodulen som er installert med TestCafe. Denne tilleggsmodulen returnerer ren tekst til terminalen og kan leses av brukere. Konvertering til datamodellene i APIet ble i konseptbeviset gjort gjennom å lese tekst og var ikke en effektiv løsning ettersom det må brukes Regex for å finne ord i teksten.

3.2.2 JSON

Denne tilleggsmodulen rapporterer i et *JSON* (JavaScript Object Notation) format som kan konverteres til objekter gjennom JSON konverteringspakker. Det er raskt og den mest populære måten å transportere data mellom systemer. Det er raskere enn mange andre formater, som *XML*, og kan inneholde lister med data.

3.2.3 xUnit

xUnit modulen bruker et XML (Extensible Markup Language) format som også vil være enkelt å konvertere til objekter i backend. Det finnes pakker i .NET som inneholder metoder for å utføre konverteringen til og fra XML. Dette formatet bruker tagger for å

åpne og lukke elementer. Et element kan inneholde andre elementer med data slik som mapper.

3.2.4 Egenutviklet plugin

Med en egenutviklet tilleggsmodul vil det være mulig å tilpasse rapporteringen til et format som passer perfekt med datamodellene til backend. Det er også blitt diskutert om backend skulle bli bygget om til en egen mikrotjeneste hvor en tjeneste tar seg av endepunkter og prosessering av resultater, mens en annen tjeneste kun tar seg av kjøringer av tester.

3.3 Alternativer for arkitektur i SoftRigTestTool backend

APIet vil inneholde all forretningslogikk, ha ansvar for å starte tester, generere rapporter og kommunisere med databasen. Det finnes to måter å designe backend på: *monolith* og *microservice*.

3.3.1 Monolith

Monolith er et program hvor all funksjonalitet ligger i samme prosjekt. Programmet vil måtte håndtere endepunkter, transaksjoner mot databaser, kjøring av tester og all logikk som kreves for å konvertere og opprette resultater. I dette tilfellet vil det da bety at alt ligger i et prosjekt. (Harris, u.å).

3.3.2 Delt mikrotjeneste

Med et microservice program vil ansvaret bli delt til ulike prosjekter som kan håndtere transaksjoner mot database, konvertering og opprettelse av resultater. Det finnes ingen universell måte å dele opp en microservice på. Dette blir et spørsmål som må besvarer for hvert program som skal utvikles. I dette tilfellet vil det være naturlig å dele Backend i to tjenester, hvor ene har ansvar for utførelsen av tester og andre håndterer testadministrasjon inklusive lagring og uthenting av data fra database. (Harris, u.å).

3.4 Alternativer for arkitektur i SoftRigTestTool frontend

Brukergrensesnittet skal gjøre det mulig for brukeren å samhandle med API for å starte tester, lese testrapporter og lage statistikk. Dashbordet kobler seg til APIet slik at forespørsler kan sendes. Det finnes mange ulike alternativer og verktøyer for å lage brukergrensesnitt, alt fra HTML og JavaScript til rammeverk som Angular og React. Det finnes mange andre rammeverk og biblioteker i forskjellige språk som støtter utvikling av webapplikasjoner. Konseptbeviset er som tidligere nevnt utviklet med Angular. React og Angular er relativt like og det er verdt å vurdere om React vil gjøre løsningen bedre sammenlignet med Angular.

3.4.1 Angular

Angular er en utviklingsplattform som baserer seg på Typescript. TypeScript er et supersett av JavaScript som introduserer typesikring. Når programmet kompilerer oversetter kompilatoren fra TypeScript til JavaScript (Microsoft, u.å-b). Koden blir deretter kjørt ved hjelp av Node.js i nettleser (Node.js, u.å). Med Angular kan applikasjoner utvikles med å benytte komponenter og HTML-maler. Komponenter kan settes sammen til mer komplekse sider. På denne måten blir komponenter gjenbrukbare og bidrar til effektiv utvikling. Angular inneholder også et bibliotek med funksjonalitet for modularisering, søppeloppsamler, hendelsesemitter og mer (Google, u.å).

3.4.2 React

React er et Javascript rammeverk med full støtte for å utvikle interaktive webapplikasjoner. På samme måte som Angular har React støtte for komponenter som kan gjenbrukes og funksjonalitet med Node.js (React, u.å).

3.5 Diskusjon av alternativene

3.5.1 Rapporteringsplugin

De forskjellige alternativene har alle fordeler og ulemper. For valg av tilleggsmodul vil det være avgjørende med detaljert informasjon fra testene og enkelt konvertering til datamodellen som brukes i APIet. Spec-modulen er løsningen som ble benyttet gjennom utviklingen av konseptbeviset og er en del som har rom for forbedringer. JSON og XML er begge standardiserte formater hvor det finnes pakker som kan forenkle konverteringen. Forskjellen mellom JSON og XML er små og inneholder litt forskjellige detaljer. JSON modulen har for eksempel starttid og sluttid, mens XML har starttid og total kjøretid. Begge har kjøretid i millisekund på hver enkelt test. En egenutviklet tilleggsmodul vil være best tilpasset til backend fordi det vil åpne muligheter for å definere mer spesifikt hvilken data som skal hentes fra resultatene. JSON og XML kan implementeres hurtig med hjelp av tredjeparts biblioteker som allerede inneholder nødvendig funksjonalitet for å gjennomføre konverteringen.

3.5.2 SoftRigTestTool backend arkitektur

Fordelene med å benytte Monolith strukturen er at det er en enklere måte å utvikle og bygge til produksjon ettersom alt ligger på et sted. Dette gjør det enklere å teste kodebasen, samt feilsøking. Når en slik applikasjon vokser vil den derimot bli mer kompleks og det kan være vanskelig å skalere systemet. Det må også tas hensyn til at hvis et slikt system går ut i produksjon vil det være tyngre å oppgradere det når nyere teknologi kommer.

Fordelene med microservice er at den oppdelte strukturen gjør det mulig å separere ansvar samt ha selvstendige *servicer* som kan oppgraderes uten å påvirke andre deler av

systemet. En microservice arkitektur er mer fleksibel og kan bygges til produksjon separat. Nedsiden med et selvstendig system er at når det vokser vil utviklingskostnaden vokse eksponentielt. Da må det også ha egne produksjonssetting og overvåking over systemet (Harris, u.å).

Alternativene til strukturering av backend vil ha større påvirkning på prosjektet. Hvis backend blir delt opp i mikrotjenester vil det være mulig å skalere tjenesten mer effektivt. Testene som kjører tar uansett lang tid og vil bruke ressurser på serveren over lengre tid. Ved å dele opp backend vil delen som har ansvar for testene kunne skalere seg selv uten at resten må skaleres samtidig. Valget har oppsider som kan være gode for et effektivt program, men det vil føre til mer kompleksitet. Dette er derimot unødvendig for et prosjekt i en så tidlig fase. Det vil være mer naturlig å utvide løsningen til å takle større belastning når det er nok brukere og dette blir en alternativ for å effektivisere løsningen.

3.5.3 SoftRigTestTool frontend arkitektur

Uni Micros frontend og konseptbeviset er skrevet i Angular. Dette gir prosjektgruppen kode som kan videreutvikles. I følge Hiren Dhaduk er både React og Angular komponentbasert, men Angular tilbyr et fult rammeverk med to-veis binding av objekter. Dersom Angular benyttes vil Uni Micro i større grad kunne tilby støtte knyttet til utviklingen. Angular kan ha høyere læringskurve, men inneholder funksjonalitet som kan være nyttig, for eksempel *dependency injection*(Dhaduk, u.å).

3.6 Valgt løsning

Oppdragsgiver ønsker å vite om løsningen kan effektivisere utviklingsprosessen. Det er derfor ikke nødvendig å bygge løsningen til å kunne skaleres dynamisk ut i fra belastning. Backend besluttes derfor å følge Monolith arkitekturen fordi det frigjør tid til utvikling av mer sentrale funksjonaliteter for systemet.

For å bygge dashbordet velges det å ta i bruk Angular fordi Uni Micro sin frontend og konseptbeviset er skrevet i Angular. Prosjektgruppen får dermed kode som kan brukes og videreutvikles samt større tilgjengelighet til ressurser og hjelp knyttet til utviklingen.

Valget av rapporteringsplugin blir JSON-modul fordi det er et populært og velkjent format med pakker som inneholder en standardisert konverteringmåte for data til ønsket datamodell. I tillegg til å frigjøre mer tid til sentral funksjonalitet i dashbordet vil det også gjøre systemet enklere å sette seg inn i ved eventuell videreutvikling.

3.7 Valg av verktøy og språk

3.7.1 Språk og rammeverk

C#

Backend er skrevet i C# (Microsoft, u.å-b) som er et objektorientert programmeringsspråk utviklet av Microsoft i 2001. Språket er type sikret og er basert på C-familien. C# har innebygd funksjonaliter som gjør språket robust og skalerbart gjennom avvikshåndtering, søppeloppsamling og kjører på .NETs virtuelle miljø som heter "Common Language Runtime" (CLR)

.NET Core

.NET Core er et open-source rammeverk utviklet av Microsoft som er utviklet for en mer moderne hverdag hvor det er skybasert og kompatibelt på forskjellige plattformer som MacOS, Linux og Windows. .NET Core er også bakoverkompatibel med .NET Framework (et eldre rammeverk fra Microsoft).

TypeScript

TypeScript er et open-source programmeringsspråk utviklet og vedlikeholdt av Microsoft (Microsoft, u.å-c) . Språket er et supersett av JavaScript som legger på statisk type sjekking og gjør det lettere å oppdage feil.

Angular

Angular (Google) er et TypeScript-basert, open-source webapplikasjons rammeverk utviklet av Google. Rammeverket er komponentbasert og gjør det enkelt å skalere applikasjonen ved gjenbruk av komponenter og HTML-maler. Rammeverket støtter både SPA ("Single-Page Application") og MPA ("Multi-Page Application"), men det er blitt mest vanlig å benytte SPA med *URL-ruting* for større applikasjoner.

Node.js

Node.js er et asynkront, JavaScript runtime miljø designet for å utvikle skalerbare nettverksapplikasjoner.

3.7.2 Verktøy

Visual Studio 2022

Visual Studio (Microsoft, u.å-d) er et integrert utviklingsmiljø utviklet av Microsoft for programvareutvikling i .NET og C#. Utviklingsverktøyet blir brukt av oppdragsgiver og var anbefalt for utvikling av .NET applikasjoner. Visual Studio har innebygd verktøy for å bygge og feilsøke gjennom alle faser og stadier av et programs livsløp.

Git

Git er et versjonskontrollsystem for prosjekthåndtering og tillater utviklere å samarbeide på samme kodebase. Alle utviklerne i et prosjekt kan opprette sin egen gren av prosjektet, foreta endringer, og senere slå sammen sin gren med hovedgrenen igjen.

3.7.3 Plattform

Microsoft Azure

Microsoft Azure er en skyplattform laget av Microsoft for infrastruktur, databehandling og beregninger i skyen. Azure tillater levering av programvare og tjenester via internett.

3.7.4 Samarbeidsverktøy

Microsoft Teams

Microsoft Teams er en kommunikasjons- og samarbeidsplattform for blant annet videomøter, chat, kalender og fillagring. Teams er hovedkommunikasjonskanal med oppdragsgiver hvor alle møter blir holdt. Informasjon blir postet på en felles kanal hvor partene kan stille spørsmål og oppdragsgiver kan komme med nødvendig informasjon.

Discord

Discord er en sosial plattform med muligheter for kommunikasjon i form av lyd/video samtaler, tekstmeldinger og fildeling. For prosjektet er det satt opp en privat Discord-server som er en samling av chattekanaler for diskusjon, planlegging og samling av dokumentasjon brukt i utviklingsprosessen og rapporten.

Office 365

Office 365 er en samling av produktivitets-programvare med samarbeidstjenester utviklet av Microsoft. Her benyttes programmene Word og Excel for arbeid med dokumentasjon. I Excel er det laget Gantt-diagram og timeliste hvor hvert medlem fører arbeidstimene sine i forskjellige kategorier ut ifra arbeidsoppgavene.

Overleaf / LaTeX

Overleaf er et nettbasert LaTeX redigeringsprogram som inneholder funksjonalitet for å strukturere og designe dokumenter. Tanken bak LaTeX er at forfattere skal kunne fokusere på å skrive, mens dokument-designere kan fokusere på å få utseende til å se bra ut. Overleaf benyttes for å skrive hovedrapporten.

Trello

Trello er et webbasert Kanbanverktøy for organisering av oppgaver som gir en oversikt over hva som blir jobbet med i prosjektet til enhver tid. Her er oppgavene blitt tildelt de ulike gruppemedlemmene og delt inn i kolonner med navn To Do, In Progress, Review og Done.

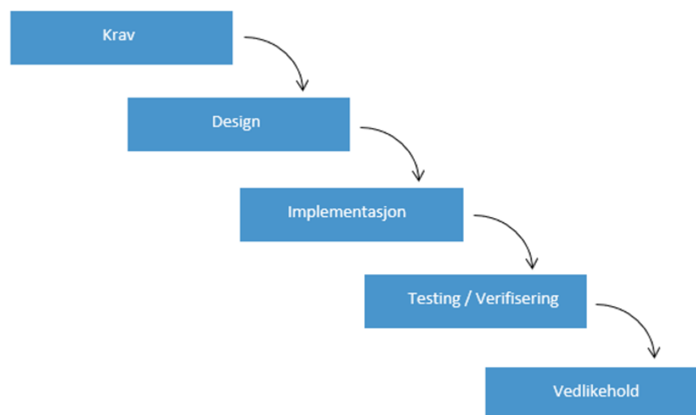
3.8 Prosjektmetodikk

3.8.1 Utviklingsmetodikk

Prosjektets utviklingsmetodikk er en blanding av prinsipper og praksiser ført av oppdragsgiver, samt hva prosjektgruppen anser som mest nyttig og relevant for utviklingsprosessen. Oppdragsgiver benytter seg av en smidig (agile) utviklingsprosess

ved bruk av Scrum og Kanban, mens prosjektgruppen har i tillegg sett på bruken av Fossefallsmodellen.

Fossefallsmodellen, illustrert av figur 3.1, er en lineær utviklingsmetode som består av fem faser: Krav, Design, Utvikling, Verifisering og Vedlikehold. Alle fasene utføres sekvensielt og avhenger av arbeidet gjort i den foregående fasen.



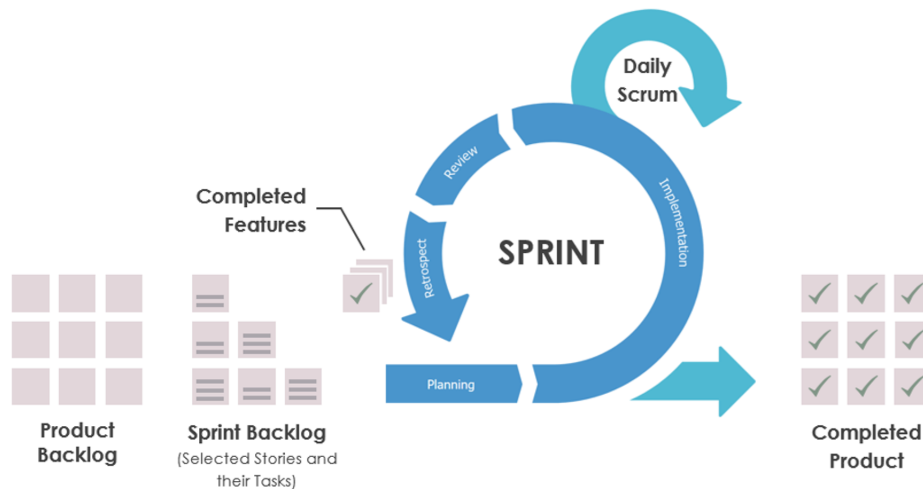
Figur 3.1: Visuell representasjon av Fossefallsmodellen.

I første fase defineres alle kravene til systemet som skal utvikles. Det dannes deretter et helhetsbilde av prosjektet og hva som skal gjennomføres. Videre blir det i andre fase bestemt hvordan kravene skal oppnås og hvilke løsninger som vil best egne seg til å møte kravene. I tredje fase velges én av fremgangsmåtene fra designfasen, og så begynner selve utviklingen av løsningen. Når løsningen er ferdig utviklet blir den testet i fjerde fase hvor det skal verifiseres at løsningen møter kravene som ble satt i forkant av utviklingen. Til slutt skal løsningen vedlikeholdes i fase fem. Her lages det nye løsninger for oppdatering og oppgraderinger av det tidligere systemet (Adobe, 2022).

Scrum er et rammeverk som benyttes innenfor prosjektledelse og systemutvikling, og som er basert på smidige prinsipp (Beck mfl., 2001). Scrum-metodikkens kjerne ligger i at det er en iterativ utviklingsprosess med tett dialog og kommunikasjon med brukerrepresentanter gjennom prosjektet. Hver iterasjon kalles for en sprint. Figur 3.2 viser et eksempel på hvordan en sprint kan være strukturert. Hver sprint har en liste med oppgaver som er satt til å skulle gjennomføres i løpet av sprint-perioden, som vanligvis har en varighet på en til fire uker. En sprint avsluttes vanligvis med å gjennomføre et sprintreview der man ser over arbeidet som ble utført i sprinten.

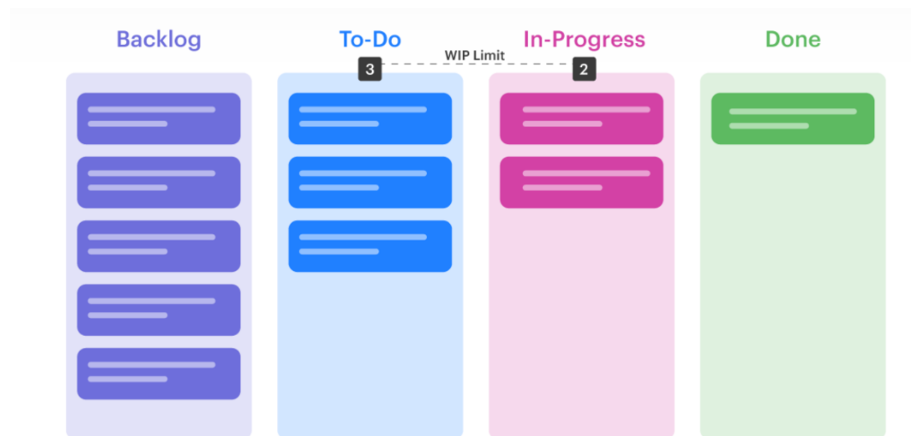
Et Scrum Team består av en Scrum-leder, Produkteier og utviklere. Scrum-lederen har

ansvaret for å hjelpe teamet med å organisere oppgaver på en mest mulig effektiv måte. Produkteier representerer oppdragsgiver og har ansvaret for at teamet utvikler det mest mulig verdifulle produktet gitt tidshorisonten. Utviklerne er de resterende medlemmene i Scrum teamet som arbeider sammen for å skape produktet.



Figur 3.2: Visuell representasjon av en sprint i en Scrum-syklus (VisualParadigm, u.å).

Kanban er en metode som brukes innen prosjektledelse for å forbedre flyten av arbeid. Metoden baserer seg på bruken av en Kanban-tavle med arbeidsoppgaver som representeres i form av et Kanban-kort. Hvert kort plasseres i en dedikert seksjon på tavlen ut ifra hva statusen er på oppgaven. En Kanban-tavle inkluderer hovedsakelig seksjonene *To-Do*, *Doing* og *Done*, men kan også inneholde andre seksjoner basert på teamets behov og arbeidsprosess. Figur 3.3 viser hvordan seksjonene også kan ha en overordnet "Work in Progress" (WIP) -begrensning for å sikre at påbegynte oppgaver fullføres før nye tas opp. Seksjoner markeres da med en maksgrense på hvor mange arbeidsoppgaver den kan inneholde til enhver tid. Dette bidrar til å redusere multitasking og heller øke fokuset på gjennomføring (Rehkopf, 2022).



Figur 3.3: Kanban-tavle med "Work in Progress" (WIP) -begrensninger (Rehkopf, 2022).

3.8.2 Diskusjon av utviklingsmetodikk

Fossefallsmodellen har sine fordeler ved at det er en lineær prosess som tidlig danner en god oversikt over kravene som stilles til systemet som skal utvikles. Det blir satt klare mål og milepæler i forkant av utviklingsprosessen som fører til mindre tid brukt på utvikling. På den andre siden har Fossefallsmodellen begrenset fleksibilitet til tilpasning når kunnskapen til teamet øker i løpet av prosessen, og/eller kundens behov endrer seg.

Scrum rammeverkets positive sider ligger i at det er en fleksibel og tilpasningsdyktig utviklingsprosess. Prosessen til Scrum har noen gode momenter som prosjektgruppen kan benytte seg av for å holde utviklingshastigheten oppe, samt holde riktig kurs i utviklingen. Scrum-metodikken kan samtidig være kompleks til tider, noe som kan kreve mer tid og ressurser for å kunne implementere. Det kreves også en teamledelse med dedikerte roller og arbeidsoppgaver.

Kanban sine sterke sider er oversiktlighet, og gir til enhver tid informasjon om planlagte oppgaver som skal gjøres for å oppnå progresjon. Ved å dele opp alle underveis-oppgavene med detaljer, vet medlemmer alt de trenger for å begynne på oppgavene. Kanban gir også prosjektmedlemmene innsikt i hva de andre i prosjektet jobber med. På denne måten slipper man å starte på oppgaver som andre holder på med.

3.8.3 Valg av utviklingsmetodikk

Prosjektets utviklingsmetodikk ble bestemt i en tidlig fase av prosjektet gjennom en prosess med testing og tilpasning. Som nevnt i kapittel 2.1.1, bygger prosjektet på å videreutvikle en påbegynt løsning. Det har derfor vært en klar idé til hva systemet skal inneholde i lang tid før hele prosjektgruppen har blitt kjent med oppgaven. Prosjektet trenger heller ikke å forholde seg til *CI/CD*, ettersom løsningen ennå ikke er lansert.

Ettersom det ikke er nødvendig med kontinuerlig leveranse av produktet, og

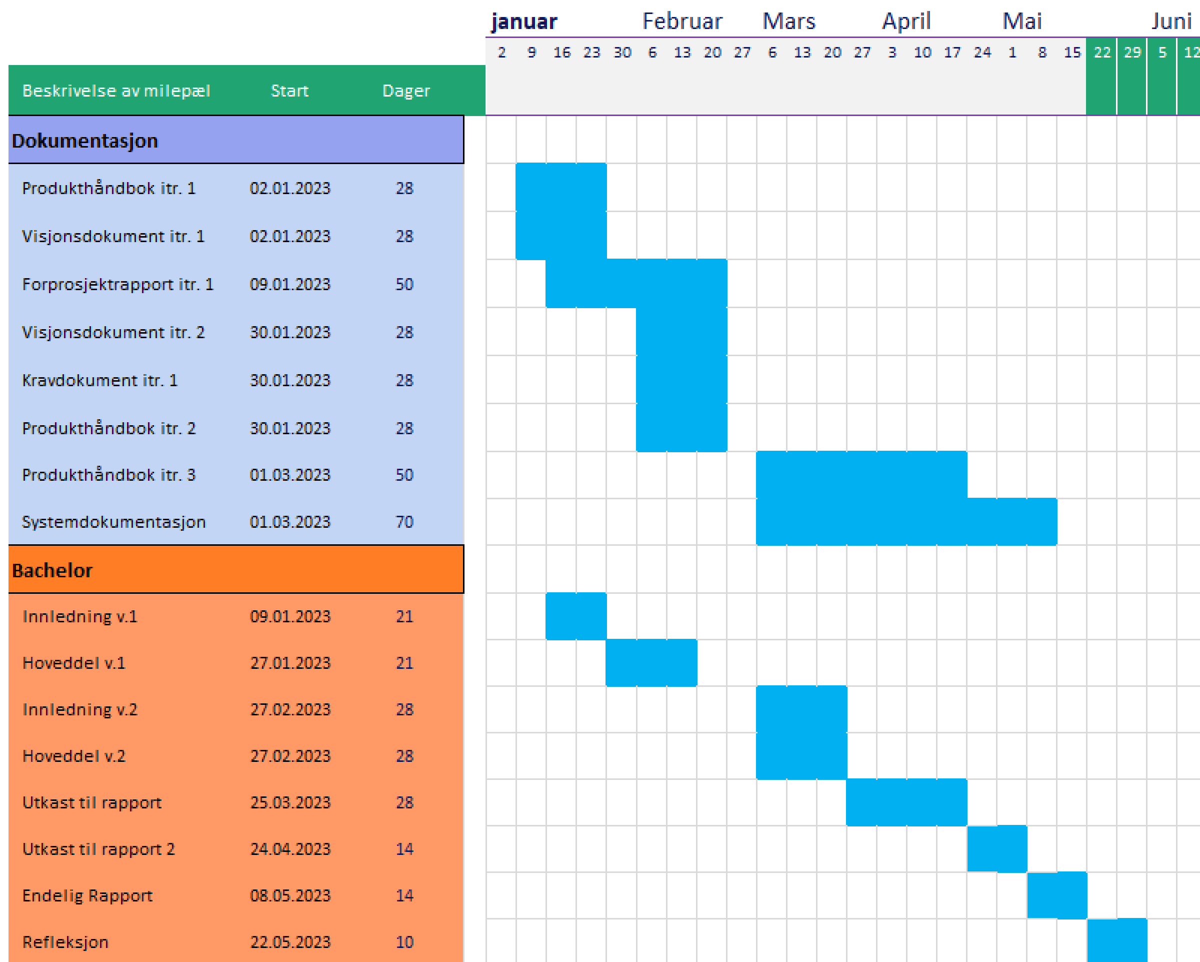
oppdragsgiver ikke har stilt andre tidsfrister enn sluttdato for prosjektet, ble Kanban valgt for å gi en mer fleksibel utviklingsprosess. Prosjektgruppen består av to personer, noe som gjør Kanban til en enkel og gunstig løsning for å holde oversikt over arbeidsoppgavene innad i teamet.

I starten av prosjektet virket det intuitivt å ta i bruk en mer lineær utviklingsprosess som Fossefallsmodellen. Men på grunn av ukentlige møter med oppdragsgiver for statusoppdatering og diskusjoner rundt løsningen ble det underveis i utviklingsprosessen et tydelig behov for å kunne kontinuerlig endre på planer og teste alternative løsninger.

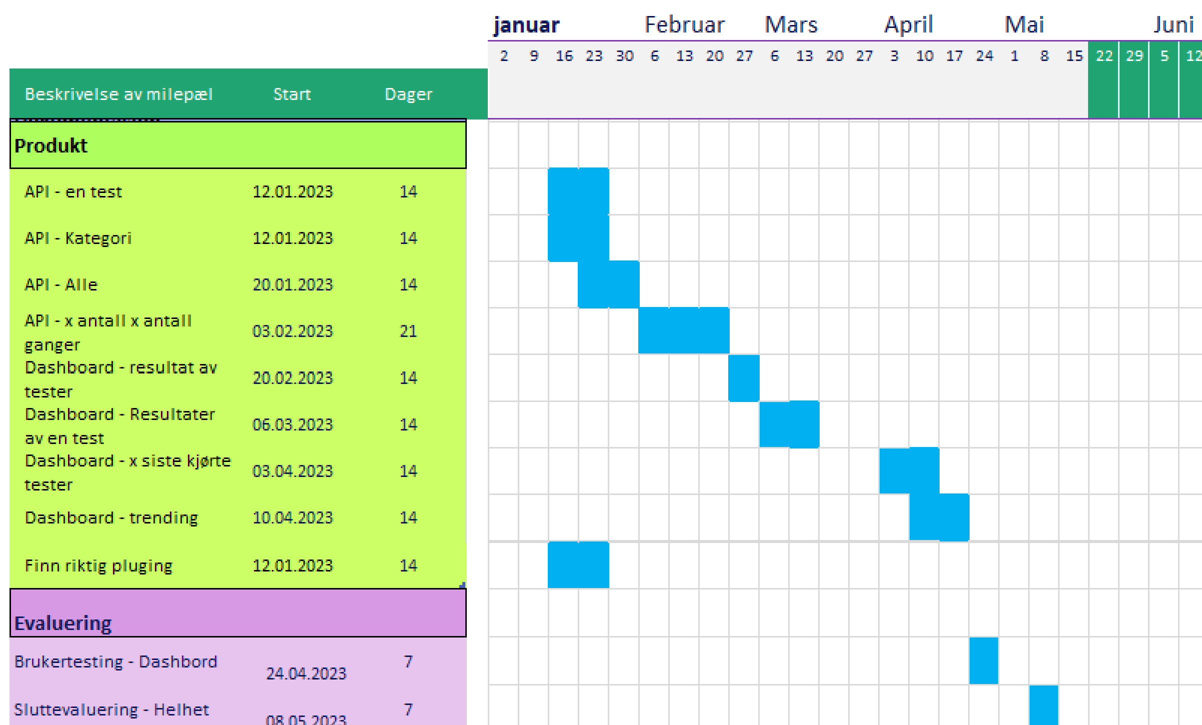
Det ble derfor bestemt å ta i bruk en iterativ og inkrementell utviklingsprosess med inspirasjon fra Scrum. Grunnet prosjektgruppen størrelse på to medlemmer ble det besluttet å ikke ta i bruk Scrum sine roller, men heller holde fokuset på sprints og sprintreviews med noen endringer. Sprintene har en varighet på en uke og avsluttes med en gjennomgang med tilbakemeldinger fra oppdragsgiver.

3.8.4 Prosjektplan

Prosjektet er planlagt sammen med oppdragsgiver. Det er utarbeidet fremdriftsplan, kravdokument og et visjonsdokument for overordnede planer. Et Kanban-brett er benyttet for å holde oversikt over arbeidsoppgaver fra dag til dag. Fremdriftsplanen, vist av figur 3.4 og figur 3.5, er delt opp etter kategorier som rapport, utvikling og dokumentasjon. Hver kategori har oppgaver som har startdato og forventet varighet. Fullstendig versjon av fremdriftsplan kan sees i prosjekthåndboken (vedlegg C)



Figur 3.4: Del 1: Fremdriftsplan for dokumentasjon og rapport.



Figur 3.5: Del 2: Fremdriftsplan for produkt og evaluering.

3.8.5 Risikovurdering

Risikoanalysen illustrert av tabell 3.1, ble brukt for å kartlegge risikoer knyttet til prosjektet, samt årsaker, sannsynlighet og konsekvenser. Tabell 3.2 viser hvordan en risiko sin sannsynlighet og konsekvens rangeres fra 1 til 5 og multipliseres med hverandre for å få risikofaktor. Fullstendig vurdering kan sees i Prosjekthåndboken kapittel 2 (Vedlegg C).

Risiko	Årsak	S	K	R	Tiltak
Løsningen klarer ikke å kjøre en stor mengde tester	Servere/løsning klarer ikke å håndtere store mengde data	1	2	2	Stressteste API på forskjellige maskiner og på server
Samarbeidsproblemer	Det oppstår splid mellom partene i prosjektet og progresjonen i prosjektet senkes	1	4	4	Signere samarbeidsavtale og ha innledende avklaring for enighet om takhøyde i diskusjoner
Sykdom	Uforutsett sykdom som leder til at parten som er syk ikke klarer å bidra	1	5	5	Ikke så mange tiltak som kan gjøres
Løsning fungerer ikke som planlagt	Kravene til produktet har endret seg eller produktet oppfyller ikke ønsket spesifikasjon	2	3	6	Være forberedt på å omstille seg for endringer
Dårlig prioritering av arbeidsoppgaver	Prioritering av arbeidsoppgaver blir ikke fulgt eller er planlagt dårlig	2	3	6	Gjennomgang av mål på prosjektet med oppdragsgiver og planlegging med Gant-diagram
Læringskurven er for høy	oppgaven viser seg å være for vanskelig og det blir problemer med å levere planlagt løsning	2	4	8	Bringe begge studentene opp til riktig nivå for å kunne utføre oppgaven
Løsning oppfyller ikke ønsket mål	Løsningen klarer ikke å kjøre tester slik som det er ønsket	2	5	10	Verifisere at det er mulig å kjøre tester tidlig i prosjektet
Løsningen blir ikke brukt av utviklere og testere	Løsningen fungerer ikke godt nok til å være interessant nok for brukerne til å endre arbeidsflyten	3	5	15	Brukerteste applikasjonen

Tabell 3.1: utdrag av Risikoanalyse fra Prosjekthåndbok

Sannsynlighet	Svært Høy (5)	5	10	15	20	25
	Høy (4)	4	8	12	16	20
	Middels (3)	3	6	9	12	15
	Lav (2)	2	4	6	8	10
	Svært Lav (1)	1	2	3	4	5
		Svært Lav (1)	Lav (2)	Middels (3)	Høy (4)	Svært Høy (5)
	Konsekvens					

Tabell 3.2: Risikodiagram som benyttes for å finne risikofaktoren til hver risiko basert på sannsynlighet multiplisert med konsekvens.

3.9 Evalueringsplan

Løsningen og valg foretatt i utviklingsprosessen vil bli evaluert kontinuerlig gjennom hele prosjektet. Som tidligere nevnt er det satt opp ukentlige sprintmøter med oppdragsgiver hvor prosjektgruppen forteller om planer for utviklingen og oppdragsgiver kommer med innvendinger.

Løsningen vil også bli jevnlig validert gjennom systemtesting under utviklingen. Brukstilfellene vil da bli testet for å sikre at initielle krav er møtt og at all funksjonalitet fungerer som tiltenkt.

Når en stabil versjon er blitt utviklet vil løsningen lastes opp på en server, slik at det kan gjennomføres en brukertest av Q&A ansatte i Uni Micro. Testerne vil kunne gi tilbakemeldinger på hvordan GUIet oppleves å bruke, og om det skulle være funksjonelle eller visuelle problemer med det.

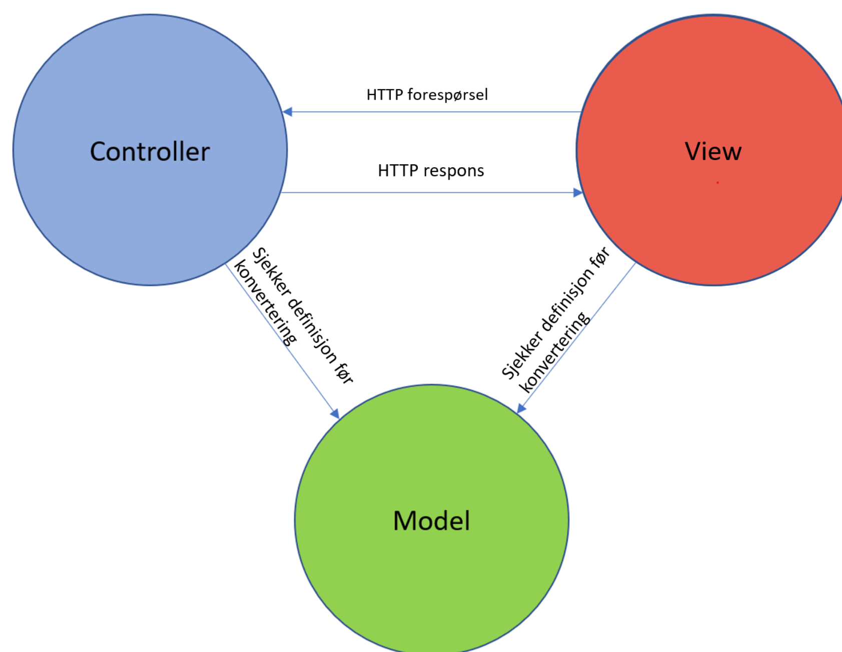
Sluttevaluering sammen med oppdragsgiver vil bli gjennomført etter brukertesting som en avslutning på arbeidet med løsningen. Her kan oppdragsgiver komme med tilbakemeldinger på prosjektgruppen og egen innsats samt levert løsning.

4 DESIGN OG UTVIKLING

Den valgte løsningen er en webapplikasjon som automatiserer kjøring og rapportering av tester som i dag gjøres manuelt i UniEconomy.no. For å utføre selve testene blir TestCafe benyttet. Dette er skript som automatiserer operasjoner som klikk, scrolling, skriving og lesing av tekst i nettleseren. Løsningen på prosjektet er inspirert av Uni Micro sin systemarkitektur hvor frontend og backend er separate prosjekter. Systemet baseres på *MVC* og REST-prinsippene hvor det finnes en definisjon på modellene (se vedlegg D, Systemdokumentasjon kapittel 4) for frontend og backend som samsvarer med hverandre. I figur 4.1 vises det hvordan systemet spør de ulike delene om informasjon.

Frontend har hovedansvar for *view* og backend har *controller* for ressursene. Både frontend og backend har tilgang til *model* som fungerer som definisjonen på hva et objekt skal konverteres til når det kommer i en HTTP-forespørsel eller respons. Frontend vil spørre backend om informasjon som brukes for å fylle malene til brukergrensesnittet. Backend kan også sende direkte til frontend uten at frontend har spurt om dette med hjelp av SignalR WebSocket.

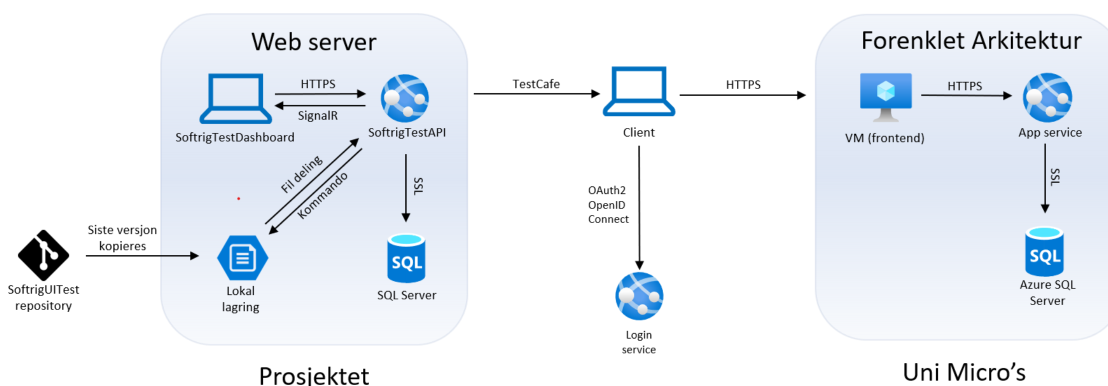
Hvis frontend blir sett på i mer detalj har den også sine egne controllere som håndterer forespørsler fra brukerens interaksjon gjennom klikk på knapper, input av tekst og andre interaksjoner. Controllere i frontend er ikke diskutert i detalj i rapporten ettersom rammeverket Angular håndterer kobling mellom view (HTML) og controller (TypeScript) ved å spesifisere metodenavn.



Figur 4.1: MVC designmønster hvor controller og view får en definisjon fra model og forespørsler går fra view til controller og data returneres som JSON.

4.1 Bakgrunn for valgt design

Prosjektets design er valgt for å kunne videreutvikles av Uni Micro og kobles til arkitekturen som allerede eksisterer i SoftRig. Uni Micro bruker en microservice struktur hvor prosjektets løsning vil kunne bli en av tjenestene som eksisterer i denne arkitekturen dersom den legges til. Figur 4.2 viser prosjektets løsning koblet opp mot Uni Micro sin arkitektur.



Figur 4.2: Uni Micros arkitektur med prosjektets kobling mot Uni Micro.

Løsningen vil i første omgang bruke tester som er lagret lokalt. Valgt løsning vil kunne endres til å bruke en kobling som enten henter lokalt fra maskin eller på nett fra skyen om det blir lagt til rette for dette. En lokal database som inneholder alle tester og testresultater skal benyttes, men kan kobles om til TestRail om det blir tilgang. For å hente data fra skyen må det endres på *connection string* og opprette en tjeneste som henter data fra TestRails API. Figur 4.3 viser et eksempel på en connection string som inneholder informasjon om server, navn på databasen og om Trusted connection skal benyttes. Trusted connection benytter Windows sertifikater

```
@"Server=(local);Database=EfCoreSample;Trusted_Connection=True;"
```

Figur 4.3: Et eksempel på en connection string

4.2 TestCafe

For å automatisere handlinger på nettsider finnes det ulike alternativer. Selenium WebDriver er det mest kjente rammeverket som lar brukeren lage testskript som utfører handlinger i nettleseren (Selenium, u.å). TestCafe er det nye alternativet som baserer seg på Node.js, og er langt enklere å implementere enn Selenium. TestCafe støtter JavaScript, TypeScript og CoffeeScript ((BrowserStack, udatert), u.å). Testene som backend starter er skrevet i JavaScript og kan kjøre på alle nettlesere.

4.3 Dataoverføring

Kommunikasjon mellom frontend og backend vil basere seg på REST som er en arkitekturstil som har flere forutsetninger. Den første er klient-server som baseres på *Separation of Consern (SoC)*. Dette gjør at brukergrensesnittet og forretningslogikken ligger på to forskjellige plasser og kan utvikles uavhengig av hverandre. Den andre forutsetningen er *Stateless*. Dette referer til at data som sendes mellom to systemer må inneholde tilstrekkelig informasjon til at det mottakene system kan tolke den mottatte dataen uten ytterligere informasjon om det andre systemet. Det finnes flere forutsetninger, men disse to er de viktigste for designet av prosjektets løsning.

Prosjektet kommer til å benytte både HTTPs og WebSocket for å sende data mellom klient og backend. For å kunne sende data må objektene konverteres til JSON, og ettersom to forskjellige kanaler skal brukes for å sende data må man passe på at disse kanalene har like innstillinger. Dersom det ikke tas hensyn til standard innstillinger kan objektenes attributter ende opp med forskjellige navn. C# bruker stor bokstav på alle objekt attributter, og siden backend er hovedapplikasjonen vil Dashbordet rette seg etter APIet. JSON-innstillinger vil derfor bli slått av slik at serialiseringen er sensitiv til stor og liten bokstav.

4.3.1 HTTPs

HTTP-meldinger skjer i to steg: *request* og *response*. I figur 4.2 vises det hvordan klienten spør APIet i backend om informasjon eller om å starte tester. Endepunktene som mottar en slik spørring er basert på en klasse som heter *ControllerBase* som ligger under et *namespace* ved navn Microsoft.AspNetCore.Mvc. Denne klassen inneholder metoder for å returnere HTTP-responser uten noe view.

Når kontrollerne mottar en request vil de først utføre standard sjekker på forespørselen. En viktig sjekk vil være at forespørselen inneholder forventet type data og ikke er tom. Etter at dette er godkjent vil kontrolleren asynkront dirigere til serviceklassene. Når et svar kommer tilbake vil passende HTTP-statuser legges til før det returneres til sender. En passende respons kan være ”200 OK”, ”301 Redirect”, ”400 Bad Request”, ”404 Not Found” eller ”500 Internal Server Error” (Mozilla, u.å). HTTP-meldingene brukes for å gi informasjon om hva som er skjedd i backend slik at frontend kan håndtere dette for brukeren og gi forklaring på hva som har skjedd. Istedet for å returnere et view returneres objekter som inneholder data. Denne dataen vil bli konvertert til JSON og returneres som en tekststreng til frontend, som konverterer tilbake til riktig objekt ved hjelp av modellen som vist i figur 4.1. Grunnen til konvertering frem og tilbake er at HTTP kun kan sende tekst og tall.

4.3.2 SignalR

Når tester er ferdige og resultater er generert vil det være ønskelig for brukeren å få en notifikasjon fra backend om at resultatene er ferdige. Dette kan gjøres ved bruk av HTTPS, men vil være lite ressurseffektivt da det krever at forespørsler må sendes gjentatte ganger frem til et resultat eksisterer. Programvarebiblioteket *SignalR* inneholder funksjonalitet som forenkler dette og lar APIet fortelle klienten når det har informasjon og sende denne direkte til klienten.

SignalR har funksjonalitet for to-veis kommunikasjon mellom server og klient og er utviklet av Microsoft. Denne pakken inneholder forskjellige lag med teknikker for to-veis kommunikasjon som benyttes ut i fra kompatibilitet til klient og server eller hvis et av de nyere teknologilagene feiler. De ulike lagene i pakken er *WebSocket*, *Long Polling* og *Server Sent Events*. SignalR vil også håndtere oppkobling og holder tilkobling i live så lenge klienten ønsker. Dersom det skulle oppstå nettverksproblemer vil SignalR automatisk koble server og klient opp igjen.

WebSocket er en protokoll som åpner en toveis kommunikasjonskanal over *TCP*. Når WebSocket-protokollen opprettes blir det gjennomført et håndtrykk mellom klient og server som kobler dem og gjør det mulig å fritt sende data mellom partene. *Server Sent Events* (SSE) er neste lag SignalR vil prøve å bruke hvis WebSockets feiler. SSE er en-veis kommunikasjon fra server til klient. *Long-Polling* er det siste som SignalR kan falle tilbake til hvis alle de andre teknologiene ikke støttes. Long-Polling fungerer ved at klienten sender en spørring til serveren som står åpen helt til serveren kommer med et svar. Når klienten mottar et svar sender den en ny spørring med det samme. Dette er basert på webapplikasjonsmodellen *Comet* (Oracle, u.å).

SignalR bruker *hubs* for å utføre kommunikasjonen mellom server og klienter. Hubs lar klienten kalle metoder på serveren og omvendt. Det er også mulig å sende parametre av en type til metoder hos den andre parten. Når en hub skal kalle metoder hos en klient vil den sende en beskjed som inneholder navn og parametre på metoden hos klienten. Objektene som blir sendt blir deserialisert basert på en av to hub protokoller: tekst basert (JSON) eller binært, *MessagePack*. MessagePack er på samme måte som JSON en serialiserer som håndterer konvertering til og fra binær. SignalR støtter *Remote Procedure Calls* (RPC) som lar serveren sende informasjon til alle klienter før det er spurt om. Dette er veldig nyttig for applikasjoner som har chat-funksjonalitet eller som skal vise data fortløpende (Microsoft, u.å-a).

4.4 Backend

Backend er delt inn i 3 forskjellige deler: controller (figur 4.4), service og model som har hver sine ansvarsområder. Kontrolleren har endepunktene som er eksponert ut via en URL. Service har all forretningslogikk som utfører ønsket oppgave og modell klassene inneholder definisjonen på objektene. Når en tester skal kjøres, oppretter backend en

terminal og sender kommandoer til et repository som ligger lokalt på maskinen eller serveren. Dette starter en klient som kobler seg til UniEconomy.no og utfører skriptet. Prosessen med å opprette terminaler og sende kommandoer skjer asynkront. Dette leder til at backend kan starte en test mens den venter på andre. Når TestCafe er ferdig med kjøring av en test vil backend kunne prosessere resultatene og levere tilbake en rapport til brukeren.

```
[HttpGet("category/{category}")]
0 references | Jan William Jensen, 1 day ago | 1 author, 3 changes | 1 work item
public IActionResult getCategoryResults(string category)
{
    if(category == null)
    {
        return BadRequest();
    }
    var result = _resultService.GetCategoryResults(category);
    if(result == null)
    {
        return NotFound();
    }
    return Ok(result);
}
```

Figur 4.4: Et endepunkt som tar inn en streng og returnerer en HTTP-statuskode med resultat.

4.4.1 Separation of Concerns (SoC)

Utformingen av backend tar inspirasjon fra SoC, slik som MVC, hvor hver klasse vil ha definerte ansvarsområder. Backend kan da deles opp i kontroller og service. Serviceklassene inneholder forretningslogikken som utfører oppstart av tester, sanitering av data som kommer tilbake fra TestCafe, generering av data til rapporter og kommunikasjon med databasen. Serviceklassene er delt opp i forskjellige mapper etter funksjonalitet. ConfigureService er en service som registrerer *interface* med klassene i en *Service collection* som håndterer opprettelse og sletting av objekter som er nødvendig og muliggjør dependency injection. Dersom det hardkodes inn en avhengighet ved å opprette et nytt objekt manuelt vil klassene måtte modifiseres når en av avhengighetene endres. Når et prosjekt vokser og det er avhengigheter rundt i programmet vil dette bli vanskelig å håndtere. I figur 4.5 er kontrolleren avhengig av et interface av en service som utfører logikk. Når denne klassen instantineres blir det ikke gitt parametre for interfacet. I figur 4.6 er det vist at serviceklassen er avhengig av DatabaseContext"og ProgressHub".

```
public class TestCaseResultCtrl : ControllerBase
{
    private readonly ITestCaseResultService _resultService;

    0 references | Jan William Jensen, 161 days ago | 1 author, 1 change
    public TestCaseResultCtrl(
        ITestCaseResultService testCaseResultService
        )
    {
        _resultService = testCaseResultService;
    }
```

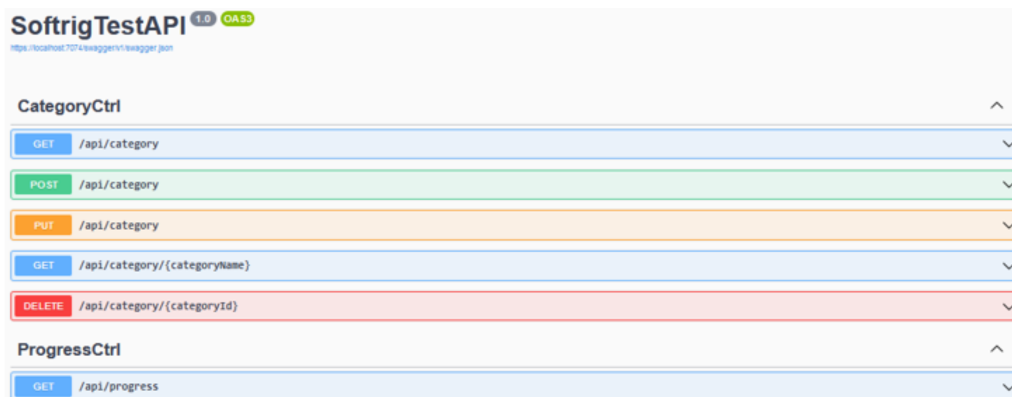
Figur 4.5: Kontroller klassen henter et objekt av TestCaseResultService uten å gi parametre.

```
public class TestCaseResultService : ITestCaseResultService
{
    public IDbContextFactory<DatabaseContext> _dbFactory;
    public IHubContext<ProgressHub> _hubContext;

    0 references | Jan William Jensen, 2 days ago | 1 author, 1 change | 1 work item
    public TestCaseResultService(
        IDbContextFactory<DatabaseContext> contextFactory,
        IHubContext<ProgressHub> hubContext
        )
    {
        _dbFactory = contextFactory;
        _hubContext = hubContext;
    }
```

Figur 4.6: Konstruktøren til TestCaseResultService som tar inn to parametre.

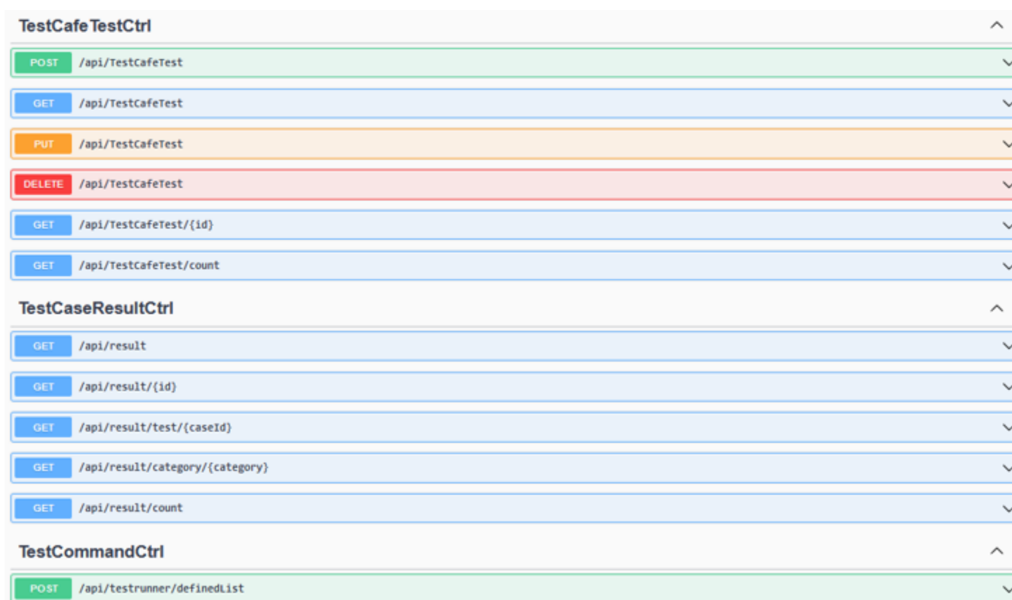
4.4.2 Endepunkter

Som tidligere nevnt er APIet basert på REST-prinsippene. Kontrollerene er delt opp etter ressurs og inneholder de forskjellige endepunktene som er tilgjengelig for hver ressurs. Alle endepunkter er dokumentert med hjelp av Swagger, som er et dokumentasjon og utviklingsverktøy. I likhet med serviceklassene håndterer klassene med endepunktene kun delegering til serviceklasser og retur av HTTP-meldinger slik som beskrevet i kapittel 4.3.1. Alle endepunkter ligger under URL: "./api" med videre klassifisering ut i fra ressurs. I figur 4.7 vises et eksempel på dette for kategori med ruten "./api/category". Starten av ruten begynner med "./" beskrive den relative URL-stien slik at uavhengig om dette kjøres lokalt (localhost), på et intranett eller det ligger på et eget domene så vil det alltid være samme relative URL-sti til de ulike ressursene.



Figur 4.7: Del 1, Endepunkter som eksisterer i APIet, vist med hjelp av Swagger.

I figur 4.8 vises de andre endepunktene som benyttes for å gjennomføre *CRUD* operasjoner på tester i TestCafeTestCtrl og de ulike endepunktene for å hente resultater fra databasen. Til slutt er det også et endepunkt som starter testene.



Figur 4.8: Del 2, Endepunktene kan åpnes og testes ut ved å gi parametre.

4.5 Dashbord

Frontend er et Angular prosjekt som bruker HTML-maler, CSS og TypeScript klasser som utgjør en komponent. Komponenter bygger opp de forskjellige sidene og kan gjenbrukes. Det finnes tre alternativer å sende data mellom to komponenter på. I figur 4.9 er det vist de tre alternativene hvor parametre markert med klammeparenteser er input, vanlige paranteser er output. Til slutt finnes det to-veis binding som brukes for variabler som kan endres både fra forelder og barn eller mellom HTML og TypeScript komponent.

```
1    (output)
2    [input]
3    [(to-way-binding)]
```

Figur 4.9: Eksempel på de ulike bindingene mellom komponenter.

Outputs er events som skjer i en komponent og sender et signal til foreldre-komponenten for å håndtere dette. Det kan for eksempel være markering av en boks eller trykk på en knapp. Et signal sendes da ut fra komponenten slik at foreldrekomponenten håndterer hendelsen. Figur 4.10 viser hvordan to komponenter blir brukt til å bygge opp en foreldrekomponent. Barnekomponentene, Checkbox-selector og Line-Chart tar inn ulike parametre som blir håndtert inne i hver av komponentenes TypeScript-fil. Komponenten kan brukes flere steder og har varierende bruksområder. Av den grunn kan ikke alt håndteres i Checkbox-selector komponenten og events sendes ut fra komponenten. Ved å bruke dette designmønsteret blir koden mer gjenbrukbar og hver komponent har sitt eget ansvar, som følger SoC.

```
<div class="grid">
  <checkbox-selector
    [testLibrary]="testLibrary"
    (clickedBox)="isCheckedBox($event)">
  </checkbox-selector>
  <line-chart
    [testSummary]="testSummary"
    [isCheckedList]="isCheckedList">
  </line-chart>
</div>
```

Figur 4.10: Utdrag fra en HTML-mal som benytter seg av komponentene Checkbox-selector og line-chart.

4.5.1 Ag-Grid

Ag-Grid er en JavaScript-komponent designet for å vise og håndtere store datasett på en effektiv måte. Det tilbyr flere funksjonaliteter som blant annet sortering, filtrering, gruppering, redigering og mer. Ag-grid kan brukes med de fleste JavaScript-rammeverk som Angular, React, Vue og Solid. Ag-grid finnes i to ulike versjoner, en open-source løsning og en lisensbasert enterprise versjon som tilbyr flere avanserte funksjonaliteter. Det finnes flere alternativer som tilbyr mye av den samme funksjonaliteten som ag-grid enterprise, men som også krever lisens for å brukes.

I dashbordet brukes Ag-grid for å trekke frem, strukturere og søke på test resultater fra kjørte tester som ligger i databasen. I figur 4.11 er hver rad et resultater fra en kjørt test. Kolonnene i tabellen forteller noe om alle attributtene knyttet til en test slik som ID, kategori, testnavn, dato, kjøretid og eventuelle feilmeldinger dersom testen feiler.

ID	Date Time	Case ID	Passed	Run Time	Result Message
1	01.02.2023	C129513	false	50	The specified selector does not match any element in the DOM
2	01.02.2023	C28736	false	38	TypeError: _utilities.util.pressTabTimes is not a function Browser:
3	01.02.2023	C28739	false	21	TypeError: _utilities.util.pressTabTimes is not a function Browser:
4	01.02.2023	C47260	false	14	TypeError: _utilities.util.pressTabTimes is not a function Browser:
5	01.02.2023	C67964	false	38	TypeError: _utilities.util.pressTabTimes is not a function Browser:
6	01.02.2023	C93610	false	40	TypeError: _utilities.util.pressTabTimes is not a function Browser:
7	01.02.2023	C129513	false	50	The specified selector does not match any element in the DOM

Figur 4.11: Tidlig versjon av visning for resultatlisten.

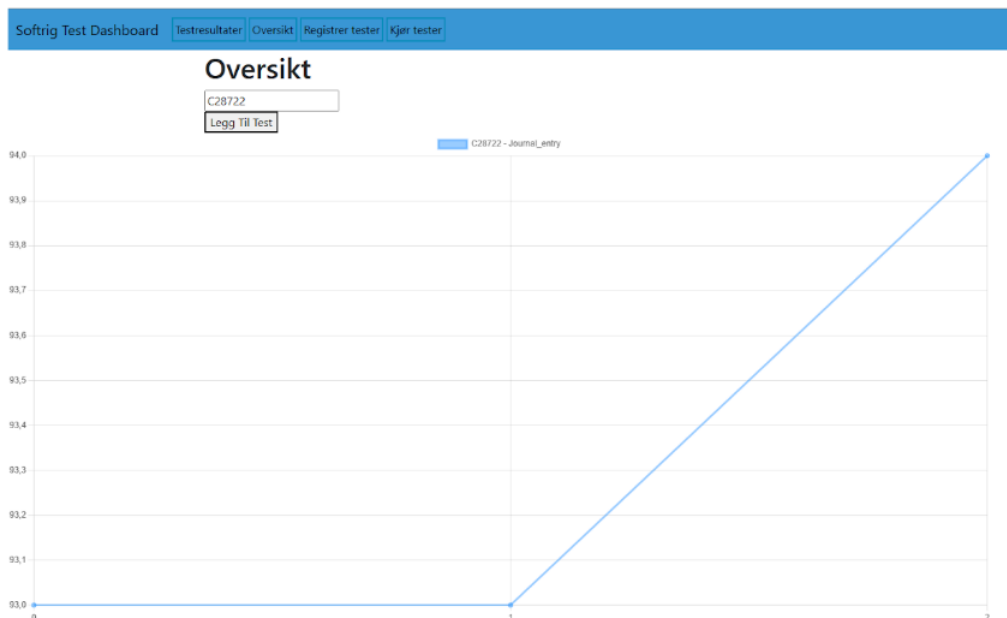
Ag-Grid brukes også sammen i komponenten for å starte testkjøringer. I figur 4.12 vises en liste med valg hvor brukeren kan skrive ulike kategorier eller *C-nummer*. Etter valg av nettleser og antall kjøringer er gjort kan testene startes. APIet vil returnere en liste med alle tester som kjører til tabellen på høyre side i figur 4.12.

Figur 4.12: Tidlig versjon av visning for test kjøring hvor bruker kan velge og starte tester

4.5.2 Chart.js

For å kunne vise frem test- statistikk og trending i frontend brukes det et open-source bibliotek med navn *Chart.js*. Biblioteket har et fleksibelt API som gjør det enkelt å lage interaktive grafer og diagrammer med mange muligheter for tilpassing av blant annet farge, skrift, avgrensninger og mer. Det finnes flere alternative biblioteker, slik som SciChart.js, ApexCharts og Apache ECharts, som alle tilbyr mye av de samme funksjonalitetene. Chart.js er derimot det mest populære blant dem, krever ingen lisens

for å brukes og anses også for å være en av de mer brukervennlige pakkene å bruke (G2, u.å). I figur 4.13 er det skrevet inn et C-nummer og fra dette genereres det fortløpende en graf som viser kjøretid i Y-aksen og test nummer i X-aksen.



Figur 4.13: Tidlig versjon av visning for statistikk siden.

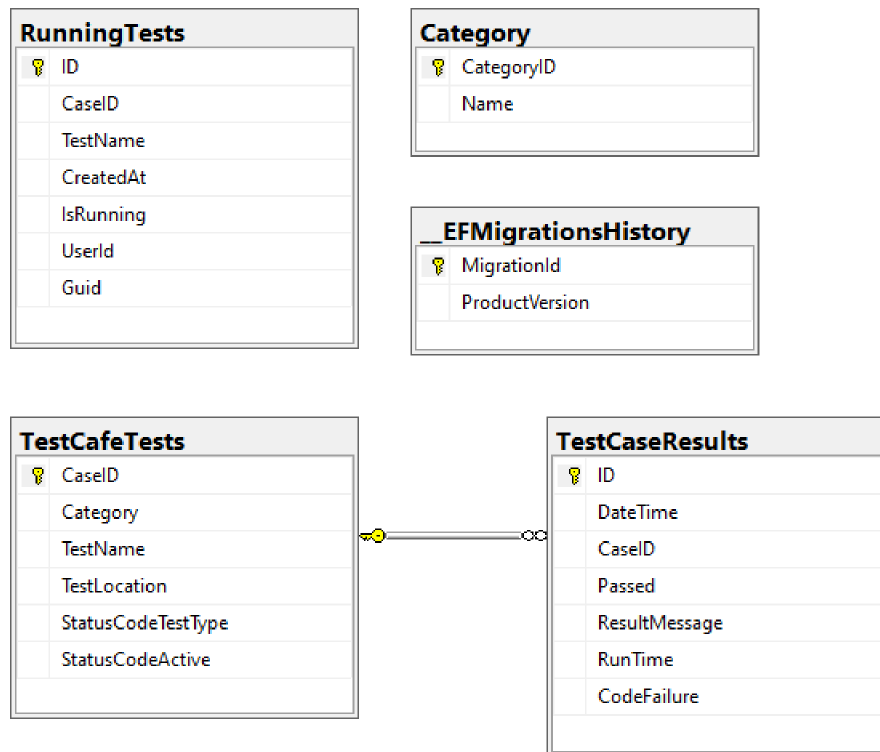
Chart.js har innebygde TypeScript-typer og er kompatibelt med de fleste JavaScript-rammeverk, slik som React, Vue og Angular. I tillegg finnes mye god dokumentasjon, API-referanser og eksempler tilgjengelig på deres nettside. En av de store fordelene ved å bruke Chart.js er at det viser diagramelementer ved bruk at HTML5 sitt *Canvas* element i motsetning til andre diagrambiblioteker som fremstiller de i SVG (Scalable Vector Graphics). Canvas elementet i HTML brukes for å tegne grafikk på websider, og dette gjør Chart.js svært effektiv til å behandle store datasett ettersom det med SVG ville krevd å opprette flere tusen SVG-noder i *DOM-treet* (Nikoloff, u.å).

4.6 Database

Lagring av data skjer ved hjelp av SQL Server, som er et relasjonsdatabasesystem utviklet av Microsoft. SQL Server blir benyttet slik at samme databasesystem brukes på tvers av Uni Micro sine løsninger. Hvilken data som var nødvendig ble diskutert sammen med oppdragsgiver. For prosjektet var det kun nødvendig å lagre tester og resultater.

Figur 4.14 viser løsningens databasetabeller RunningTests, Category, EFMigrationsHistory, TestCafeTest og TestCaseResults. RunningTest ble implementert for å vite hvilken tester som kjører i systemet. Category er en egen tabell som lagrer alle C-nummer slik at de er unike. EFMigrationsHistory er en tabell som inneholder en logg over hvilke migreringer som er gjort på databasen. Når kodens modeller endrer seg, må

databasen oppdateres. Slike filer sikrer at uavhengig av når databasen blir opprettet, flyttet eller bygget på nytt så går den gjennom alle stegene og vil bli helt identisk til alle andre implementasjoner av databasen. TestCafeTests og TestCaseResults er tabellene som inneholder testene og resultatene til testene. Det er her en relasjon mellom CaseID slik at alle resultater kan ha med seg testen som resultatet tilhører



Figur 4.14: Tabeller som eksisterer i databasen.

4.6.1 Entity Framework Core

For å utføre transaksjoner mellom databasen og backend benyttes Entity Framework Core (EF Core) som er en pakke som fungerer som en *Object-Relational Mapper* (ORM). Med denne pakken kan man utvikle applikasjoner med en av to forskjellige konsepter: database først eller kode først. Database først tilnærming baserer seg på å tegne opp modeller og deres relasjoner før EF Core genererer modeller. Kode først er motsatt, hvor man lager modellene i prosjektet og får EF Core til å generere tabeller og relasjoner mellom tabellene. Sistnevnte er tilnærmingen prosjektet er basert på. Hvis endringer i modellen skjer kan disse enkelt migreres til databasen ved hjelp av EF Cores ”Add-Migration MigrationName” og deretter ”Update-Database”.

4.6.2 SoftrigUITest

SoftrigUITest er et repository hvor alle testfilene er lagret. Disse inneholder fremgangsmåten maskinen følger for å gjennomføre tester på UniEconomy.no. Disse

testene er kun importert og lagret slik at backend har tilgang og kan starte opp TestCafe i bakgrunnen. Når prosjektet kjøres lokalt ligger testskriptene i en mappe ved siden av prosjektet. Når prosjektet kjøres på en server ligger testskriptene som en fildeling slik at det fungerer helt likt uavhengig om det kjøres på server eller på egen maskin.

5 RESULTATER

5.1 Evalueringsmetode

Som tidligere nevnt i kapittel 3.9 evalueres prosjektet på fire forskjellige måter. Kontinuerlig evaluering gjennom sprintmøter sammen med oppdragsgiver, systemtesting med utgangspunkt i brukstilfellene, brukertesting utført av en testoperatør ansatt hos Uni Micro og en sluttevaluering hvor oppdragsgiver tar hele systemet til vurdering og ser på hvordan produktet stiller seg i forhold til kriteriene som ble satt i forkant av prosjektet.

Det ble også vurdert på å lage tester for de ulike klassene. Opprettingen av disse testene var utfordrende fordi en stor del av dem var avhengig av Moq som er et bibliotek for mocking. Løsrivingen av komponenter knyttet til databasen var mer komplisert enn først antatt og ledet til at utviklingen ble stoppet for å kunne ferdigstille rapporten. Det ble kun fullført et fåtall tester til generering av resultat. Mocking av oppstart av TestCafe viste seg å ikke være enkelt.

5.1.1 Kontinuerlig evaluering

Løsningen har blitt kontinuerlig evaluert hver uke i form av sprintmøter sammen med oppdragsgiver. Det har da blitt presentert hvilket arbeid som er lagt ned siden sist og oppdragsgiver har hatt muligheten til å komme med tilbakemeldinger på produktets utvikling. Sprintmøtene har også fungert som en mulighet for prosjektgruppen til å kunne oppklare utviklingsrelaterte spørsmål som tidligere har vært uklare.

Evalueringsmetoden har hatt som mål å sikre fremdrift i utviklingsprosessen, og at oppdragsgiver har kunnet styre prosjektet i en retning som møter de forventningene som ble satt til prosjektet i forkant.

5.1.2 Systemtesting

Løsningen har jevnlig blitt validert gjennom utviklingsprosessen i form av systemtesting. Valideringen har bestått av å gjennomgå løsningens brukstilfelle og sjekke at systemet fungerer som tiltenkt. Denne evalueringsmetoden har hatt som mål å sikre at alle initielle krav definert i Kravdokumentet blir møtt (se vedlegg A, kapittel 2).

5.1.3 Brukertesting

Som nevnt i kapittel 3.9 ble det planlagt at løsningen skulle lastes opp på en intern server for at en testoperatør i Uni Micro skulle kunne gjennomføre en brukertest. Brukertesten hadde som mål å teste at løsningen møter oppdragsgiverens ønsker, samt å kartlegge hvordan funksjonaliteten fungerer og oppleves å bruke i arbeidssammenheng. I sammenheng med brukertesten fikk testoperatør utlevert et evalueringsskjema hvor de kunne gi tilbakemeldinger på hvordan GUIet oppleves å bruke og om det skulle være

funksjonelle eller visuelle problemer med det.

Evalueringsskjemaet ble utformet av prosjektgruppen og kan sees i vedlegg (E). Skjemaet består av fem spørsmål; tre kortsvarspørsmål og to spørsmål med påstander hvor svaret rangeres mellom 1 og 5, der 1 er lavest og 5 er høyest. Testerne hadde skjemaet tilgjengelig under brukertesten.

5.1.4 Sluttevaluering

For å enklere kunne kartlegge oppdragsgiver sin oppfatning av sluttproduktet i forhold til kravene som ble satt i forkant av prosjektet ble det avtalt å gjennomføre en sluttevaluering av produktet og prosjektet. Oppdragsgiver fikk utlevert et evalueringsskjema som skulle fylles ut. Evaluering hadde som mål å identifisere oppdragsgiver sin oppfatning av resultatet, eget bidrag til prosjektet, samt prosjektgruppens innsats.

Evalueringsskjemaet ble utformet av prosjektgruppen og kan sees i vedlegg(E). Skjemaet består av seks spørsmål med åpne svar som ser på innsats og påvirkningskraft til partene, samt om resultatet leverer på kravene som var stilt til oppgaven.

5.2 Evalueringsresultat

5.2.1 Resultat av kontinuerlig evaluering

Den kontinuerlige evalueringen har vært med å sikre at prosjektet har blitt styrt i best mulig retning for å nå kravene. Ukentlige sprintmøter med gjennomgang og diskusjon av hva som er gjort siden sist, har ført til forbedringer på løsningen som asynkron kjøring av tester og "Checkbox-selector" komponenten, omtalt i kapittel 4.5. Gjennom statusmøter er det også diskutert endring av alle komponenter til å presentere navn og C-nummer sammen for å forbedre brukeropplevelse.

5.2.2 Resultat av systemtesting

Systemtestingen ble gjennomført jevnlig gjennom hele utviklingsløpet. Resultatene fra systemtestingen har bidratt til økt forståelse og forbedringer både i frontend og backend. For backend har dette ført til en forenklet kodestruktur hvor endepunkter har kunnet fjernes for bruken av andre. I frontend har det gitt forståelse for blant annet nettleserkapasitet, flyt og brukeropplevelse.

TestCafe har blitt testet for å utforske kapasitet knyttet til kjøring av tester i parallel. Kjøringen av seks eller flere tester medførte at TestCafe begynte å rapportere feil under oppstart. Som et resultatet ble det satt en maksgrense i backend som ikke tillater TestCafe å kjøre mer enn fem tester parallelt. Kapasitet ble også testet på lokal maskin ved å sende inn forspørspørsler til det påvirket kjøretid. Ved kjøring av totalt 35 tester

ville nye forespørsler påvirke kjøretiden til de resterende testene. Dette er avhengig av prosessoren på maskinen og er ikke en universal grense. Det er også blir gjort tester for å forsikre at backend ikke venter på svar fra TestCafe. Backend ble etter testing konvertert til å starte tester asynkront.

5.2.3 Resultat av brukertest

Brukertesten ble gjennomført av én testoperatør i midten av mai. Resultatene fra brukertesten har vært med på å kartlegge hvordan løsningen har fungert i arbeidspraksis og til hvilken grad den har opplevdes som nyttig. Figur 5.1 viser at testoperatør rangerer løsningen som 4 av 5 ved nyttighet i arbeidssammenheng, og 5 av 5 ved spørsmål om hvor villig de er til å bruke et slikt verktøy i fremtiden for å gjennomføre regressjonstester.

Hvor nyttig vil du anse denne løsningen for å være i jobb sammenheng *

1 2 3 4 5

Svært lite nyttig ○ ○ ○ ◉ ○ Svært nyttig

Hvor sannsynlig er det at du ville brukt et slikt verktøyet i fremtiden for å gjennomføre regressjonstester *

1 2 3 4 5

Svært lite sannsynlig ○ ○ ○ ○ ◉ Svært sannsynlig

Figur 5.1: Utdrag fra evalueringsskjema for brukertest (vedlegg E).

I det ene spørsmålet blir testoperatør spurt om å kort beskrive sine tanker om løsningen. Figur 5.2 testoperatør anser løsningen for å gi en god oversikt over kjørte tester og tilhørende statistikk, samt enkel å anvende for utvelging og oppstart av tester.

Beskriv kort dine tanker om programmet

God oversikt over kjørte tester med statistikk over hvor lang tid testene har brukt og en liste over hvor mye de har feilet.
Lett å lage et utvalg med tester man ønsker å kjøre og lett å sette igang alle testene.

Figur 5.2: Utdrag fra evalueringsskjema for brukertest (vedlegg E).

Ved spørsmål om forslag til videre utforming og forbedringer, vist i figur 5.3, gir testoperatør uttrykk for at løsningen kunne hatt behov for flere muligheter for filtrering i teststatistikk, samt bedre visninger for hvor mye en test har blitt godkjent eller har feilet. I tillegg påpeker testoperatør at tester avsluttes brått uten å ha verken feilet eller blitt godkjent. Dette skyldes litt manglende håndtering av responsen APIet returnerer når en test ikke eksisterer i databasen.

Er det noe annen funksjonalitet du skulle ønske eksisterte i programmet?

Mulighet til å filtrere på om en test er godkjent eller har feilet i statistikken. (For å kunne f. eks vise kjøretiden på testene uten å ta med dem som har feilet.)
Bedre visning av hvor mye en test har blitt godkjent eller har feilet. (Kakediagram? En slags tidslinje?)

Har du noen andre forslag til forbedringer?

Noen tester avslutter brått uten å ha verken feilet eller blitt godkjent, kanskje en bedre håndtering her?

Figur 5.3: Utdrag fra evalueringsskjema for brukertest (vedlegg E).

5.2.4 Resultat av sluttevaluering

Sluttevalueringen ble gjennomført av oppdragsgiver i etterkant av prosjektets siste sprintmøte. Fra spørsmålene rundt prosjektets resultat og prosjektgruppens arbeid, vist i figur 5.4, gir oppdragsgiver uttrykk for at løsningen overgikk forventningene. Utførte krav og tilleggspunkter diskuteres mer om i kapittel 5.3.

Oppfyller utviklet løsning kravene som var satt i forkant av prosjektet? Dersom ikke, beskriv mangler.

Løsningen som er utviklet, mer enn oppfyller alle krav som ble etablert i forkant av prosjektet.

Hvordan vil du vurdere arbeidet gruppen har utført?

Gruppen har gjennomført/levert et veldig profesjonelt prosjekt. De har levert på prosjekt planlegging, gjennomføring, rapportering og produkt.

Figur 5.4: Utdrag fra evalueringsskjema for sluttevaluering (vedlegg F).

Ved spørsmål om hvorvidt det var deler av prosjektet hvor prosjektgruppen kunne ha gjort det bedre, vist i figur 5.5, gir oppdragsgiver uttrykk for at det ikke var noen områder som ikke oppnådde ønsket resultat i henhold til de kravene som ble gitt i

forkant. Svaret fra oppdragsgiver gir indirekte et inntrykk om at det er ting utenfor kravene som gjerne skulle vært forbedret, men som siden oppstart av prosjektet har blitt sett på som tilleggsoppgaver dersom tiden skulle strekke til. Av disse ønskene inngår blant annet å arrangere et møte med Uni Micro sine UX designere for å tilpasse produktet mot bedriftens styling, samt integrasjon mot TestRail.

Er det deler av prosjektet gruppen kunne gjort bedre?

Ikke forhold til de kravene som var satt til prosjektet.

Figur 5.5: Utdrag fra evalueringsskjema for sluttevaluering (vedlegg F).

Figur 5.6 viser samtidig at oppdragsgiver gir uttrykk for at Uni Micro burde ha etablert plattformen som leveransen skulle kjørt på mye tidligere i prosjektet.

Er det deler av prosjektet hvor Uni Micro kunne bidratt mer for å forbedre resultatet av prosjektet?

Vi burde ha etablert plattformen som leveransen skulle kjørt på mye tidligere i prosjektet.

Figur 5.6: Utdrag fra evalueringsskjema for sluttevaluering (vedlegg F).

5.3 Prosjektresultat

Prosjektets resultat er en løsning som forenkler frontend testing på et nettsted. Løsningen som er utviklet lar brukeren starte tester som utfører script i nettleser. Testresultatene blir formatert og lagret slik at brukeren kan se feilmeldinger og trender på kjøretid. Som beskrevet i kapittel 4 består løsningen av en frontend og en backend.

5.3.1 Frontend

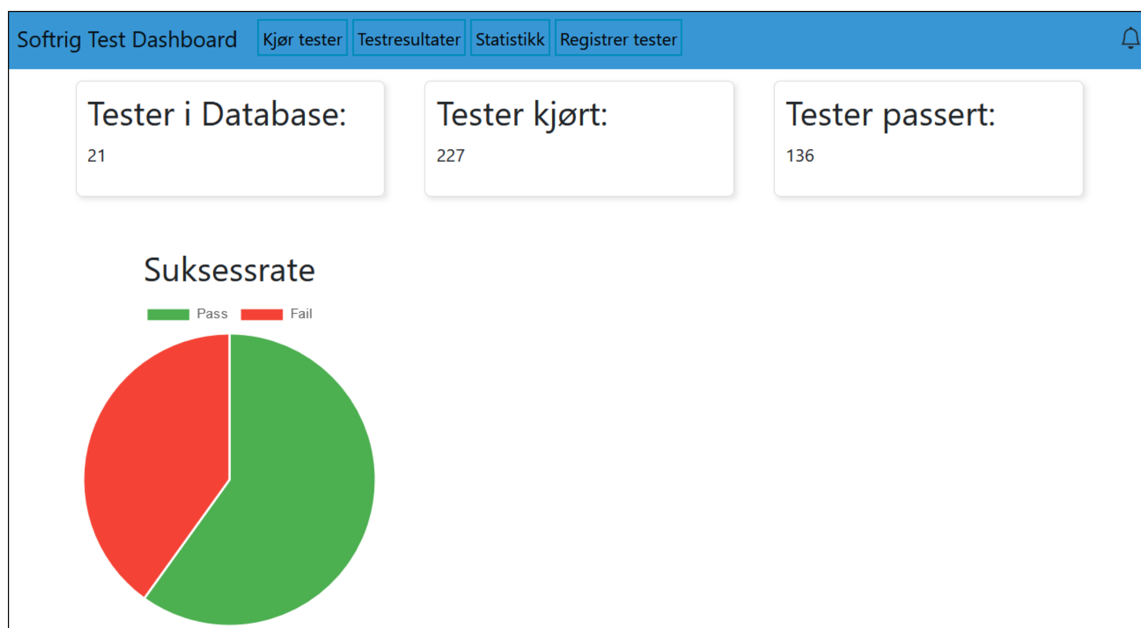
I frontend er alle kravene dekket nevnt i tabell 5.1 og Visjonsdokumentet (se vedlegg B, kapittel 5). Tester kan startes ved å velge kategori, spesifikke tester eller begge, og deretter spesifisere antall kjøringer og ønsket nettleser for kjøringen. Utenfor kravene er det også utviklet en bjelle som gir en notifikasjon når tester er ferdige med en link til resultatene.

Krav - Frontend	Resultat
Kjøre tester	Fullført
Se oversikt av testresultater	Fullført
Registrere nye tester	Fullført
Se statistikk på testresultater over tid	Fullført
Utenfor krav - Frontend	Resultat
Fremside med data om tester og resultater	Fullført
Spesifisere antall kjøringer	Fullført
Notifikasjon på resultater	Ikke Fullført

Tabell 5.1: Frontend kravene som er spesifisert i visjonsdokumentet og resultatene

Fremside med data om tester og resultat

Figur 5.7 viser løsningens forside. Denne siden har som formål å gi brukeren informasjon og en oversikt over tester og resultater.



Figur 5.7: Fremside med data om tester og resultat. I toppen er det en navigasjonslinje som leder til de ulike komponentene.

Test registrering

Alle CRUD operasjoner er implementert slik at det også er mulig å slette og oppdatere testene ved hjelp av komponenten som er vist i figur 5.8. Statuskode for en test forteller om en test er under utvikling, aktiv eller avviklet. Dette var noe som var vurdert å tas i bruk sammen med TestRail, men har ikke noe funksjon enda ettersom det aldri ble mulighet til koblet opp mot TestRail. En test legges til ved å angi testen sitt C-nummer, kategori, test navn og statuskode, etterfulgt av å trykke på "Lagre test". En test slettes ved å angi C-nummer og deretter trykke på "Slett test".

Figur 5.8: Skjermutklipp som viser komponenten for å registrere tester i systemet.

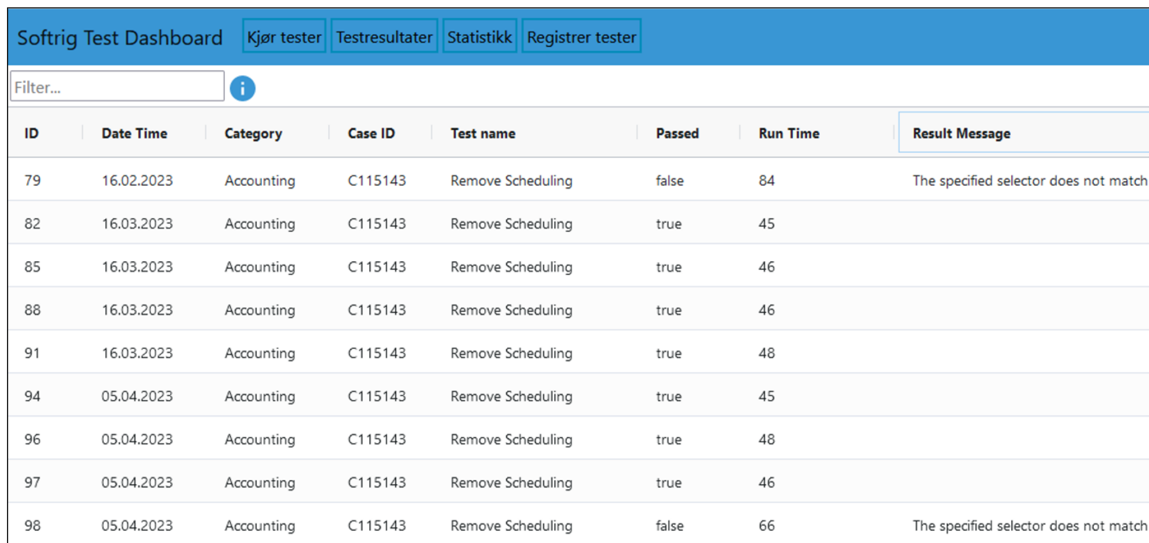
Kjøring av tester

Figur 5.9 viser siden hvor bruker kan starte og kjøre tester. Bruker angir ønskede tester for kjøring, spesifiserer nettleser og antall kjøringer før submit. Når en test er startet vil det bli vist informasjon om hvor lenge testene har kjørt.

Figur 5.9: Siden som brukeren kan velge og starte tester fra. Her med to kjørene tester.

Testresultat visning

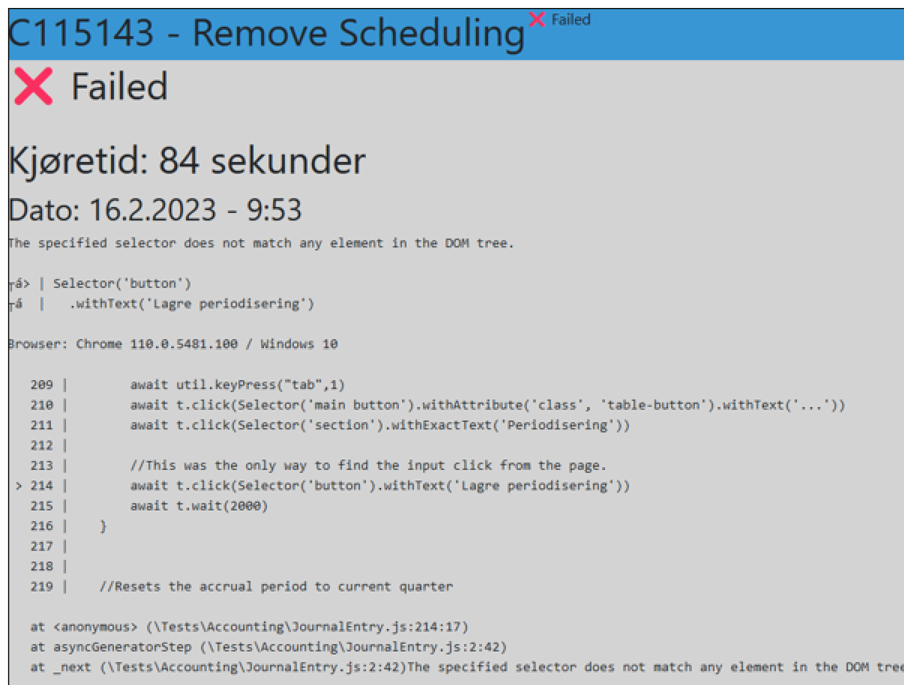
Figur 5.10 viser siden hvor bruker får en oversikt og kan søke over alle testresultater i systemet. Søkingen kan sorteres etter alle attributtene knyttet til en test. Det er mulig å klikke på de ulike cellene som leder til en visning av test resultatet med formatert feilmelding.



ID	Date Time	Category	Case ID	Test name	Passed	Run Time	Result Message
79	16.02.2023	Accounting	C115143	Remove Scheduling	false	84	The specified selector does not match
82	16.03.2023	Accounting	C115143	Remove Scheduling	true	45	
85	16.03.2023	Accounting	C115143	Remove Scheduling	true	46	
88	16.03.2023	Accounting	C115143	Remove Scheduling	true	46	
91	16.03.2023	Accounting	C115143	Remove Scheduling	true	48	
94	05.04.2023	Accounting	C115143	Remove Scheduling	true	45	
96	05.04.2023	Accounting	C115143	Remove Scheduling	true	48	
97	05.04.2023	Accounting	C115143	Remove Scheduling	true	46	
98	05.04.2023	Accounting	C115143	Remove Scheduling	false	66	The specified selector does not match

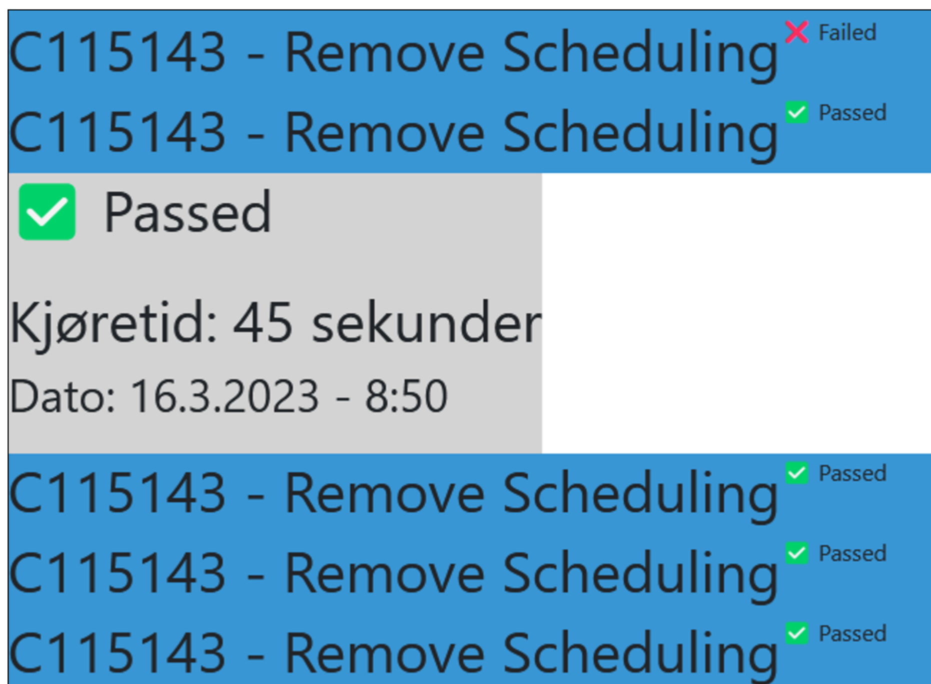
Figur 5.10: Siste versjon av TestResult komponenten som lister opp resultater og lar brukeren søke i alle kolonner.

I figur 5.11 vises et resultat hvor testen har feilet og en feilmelding fra TestCafe viser hvilken linje som feilet i testen. Dataene som behandles gir informasjon om alle attributtene knyttet til en test slik som ID, kategori, testnavn, dato, kjøretid og eventuelle feilmeldinger dersom testen feiler. For å komme til denne siden kan det klikkes på ID i resultatlisten (figur 5.10).



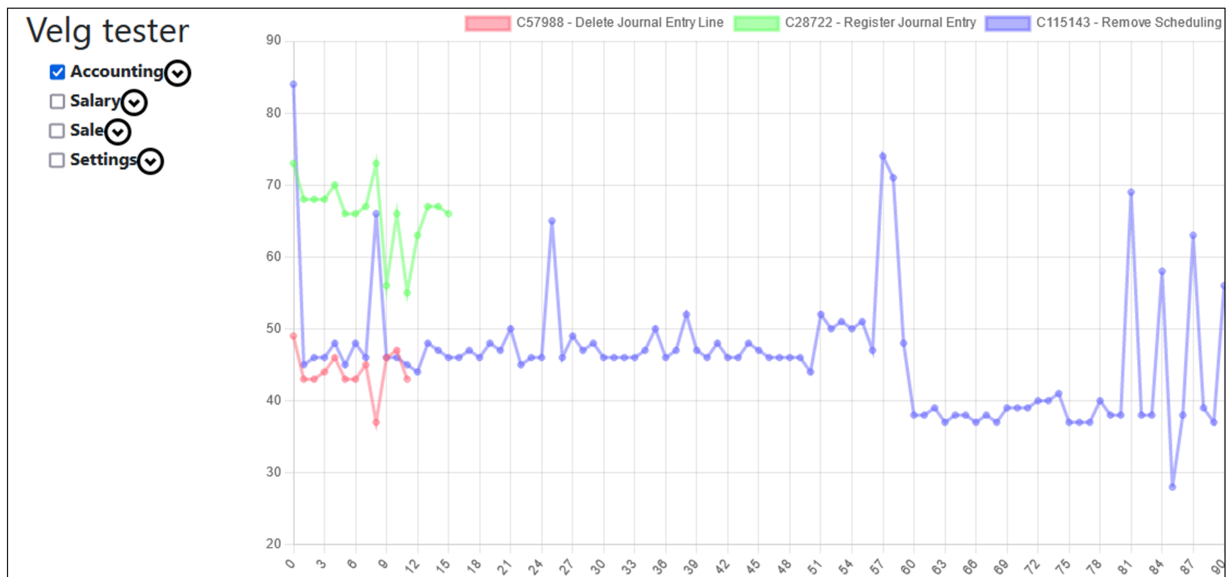
Figur 5.11: Siste versjon av TestResultView komponenten som viser resultat.

I figur 5.12 er det illustrert siden som viser når brukeren klikker på CaseID i figur 5.10. Dette viser da alle resultater til testen resultatet tilhører eller på en kategori for å se alle resultatene til den kategorien.



Figur 5.12: Skjermutklipp som viser visning av resultat fra en kategori hvor det kan klikkes på hver linje for å åpne resultatet.

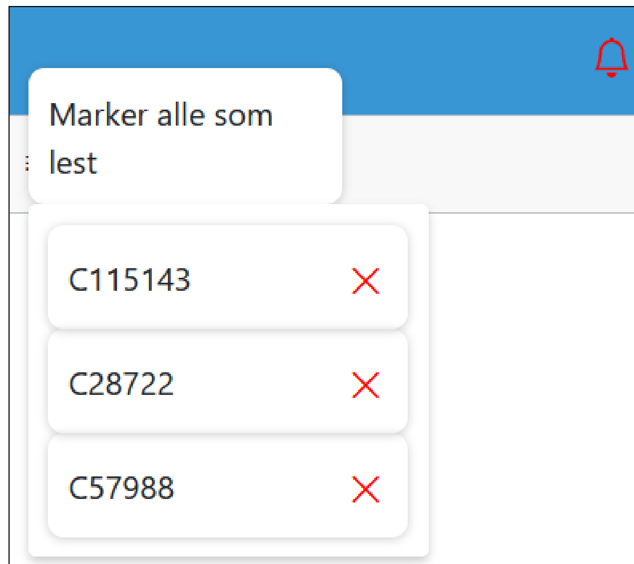
Statistikk visning Figur 5.13 viser siden for å se statistikk. Hver testkjøring representeres som en node i en graf og viser testens kjøretid. Tester kan enkelt legges til i grafen ved å klikke på testene eller kategoriene. Grafen viser en test med kjøretid for hver gang testen er kjørt.



Figur 5.13: Skjermklipp fra statistikk siden som benytter Chart.js.

Notifikasjon på resultater

Notifikasjonen som kommer når tester er klart, vist av skjermutklippet i figur 5.14, benytter seg av SignalR som beskrevet i kapittel 4.3.2. Denne komponenten ble ikke helt ferdig utviklet ettersom den ble påbegynt helt mot slutten av utviklingsperioden. Funksjonalitet fungerer som ønsket, men utseende og brukervennlighet er ikke helt ferdig.



Figur 5.14: Skjermutklipp fra notifikasjons-modalen. Ikonet er rødt når resultater er klar og svart når det ikke er nye resultater.

5.3.2 Backend

I Backend er også alle kravene utviklet (se oversikt i tabell 5.2). Tester kan startes med ulike innstillinger som nettleser og antall kjøringer av hver test. Testresultater blir generert og formatert, og disse resultatene kan sendes ut. Det er også implementert SignalR WebSocket for både live sporing av tester som kjører og notifikasjon når testene er ferdige.

Krav - Backend	Resultat
Starte en test	Fullført
Starte en kategori med tester	Fullført
Starte alle tester	Fullført
Lage rapporter på testene	Fullført
Starte tester i forskjellige nettlesere	Fullført
Hente resultater fra tester	Fullført
Se hvor lenge en test har kjørt	Fullført
Utenfor Krav - Backend	Resultat
Kjøring av egendefinert liste	Fullført
Repetere tester (dersom spesifisert i frontend)	Fullført
Enhetstester	Ikke fullført
Kobling til TestRail	Ikke fullført

Tabell 5.2: Backend kravene som er spesifisert i visjonsdokumentet og resultatene

5.4 Prosjektgjennomføring

Prosjektet ble gjennomført i perioden fra januar 2023 til mai 2023. Det ble ved oppstart satt et timebudsjett på 540 timer per medlem, noe som tilsvarer et totalbudsjett på 1080 timer. Arbeidsoppgaver har blitt fordelt jevnt ut over perioden blant begge gruppemedlemmene. Det ble satt opp ulike kategorier i timelisten som det føres timer mot. Av dem har det blitt ført 302 timer mot utvikling. Samlet for dokumentering og rapport er det brukt 350 timer. Det er også verdt å nevne at det er gått med 64 timer til kommunikasjon og presentasjon som innebærer møter, mailer og levering av obligatoriske oppgaver. Oversikt over ukentlige timelister og statusrapporter finnes i prosjekthåndboken (se vedlegg C, kapittel 4).

Prosjektet har hatt en jevn fremdrift i løpet av hele prosjektperioden. Prosjektgruppen kom i gang tidlig med utviklingen. Dette viste seg å være en stor fordel etter hvert som prosjektgruppen møtte på utfordringer med kjøring av flere tester parallelt. En del komponenter har også blitt skrevet på nytt for å kunne gjenbrukes. Det var også forventet at utvikling på backend skulle ta mindre tid enn frontend, men her ble det mer utfordrende enn planlagt ettersom kompleksiteten var større i Angular enn .NET, og prosjektgruppen ikke hadde like mye erfaringen med Angular som .NET. Disse utfordringene ble løst, men det gjorde at det fortsatt eksisterer mindre feil i dashbordet. Komponenter skulle også hatt mer tid dedikert til utseende.

6 DISKUSJON

6.1 Sluttprodukt og fremgangsmåte

I kapittel 5.3 ble prosjektresultatene presentert (se tabell 5.1 og 5.2) og kravene som ble bestemt sammen med oppdragsgiver er oppfylt. Utover kravene er ikke alle oppgaver ferdigstilt. Med tanke på målet som var beskrevet i kapittel 1.3 har løsningen nådd målet med å utvikle et verktøy som forenkler testing og rapportering. Tilbakemeldinger fra evaluering viser at oppnådd resultat er en løsning som kan effektivisere arbeidet med å verifisere kvaliteten til Uni Micros frontend. Tilbakemeldingene viste at løsningen ikke er perfekt, noe som har vært tydelig siden utvikling ble stoppet for å forberede til brukertesting.

Ved starten av prosjektet var det fra tidligere laget et konseptbevis som kunne kjøre en test og lagre beskjedene som TestCafe skrev i terminalen. Gitt viktigheten av å kunne kjøre tester parallelt og ta imot forespørsler uten å lage kø mens den venter på svar fra TestCafe, ble videreutvikling av backend startet først. Dette var en av prosjektgruppens større tekniske utfordringen og krevde tid å implementere. I kapittel 3 ble det sett på alternativer om å dele backend opp slik at det var et eget prosjekt som håndterte oppstart av TestCafe. Valget om å ha alt i en backend har fungert bra for prosjektet, men det kan godt være at om dette skal skaleres vil det være ønskelig å dele det opp i mikroservicer. Resterende krav i backend ble også møtt og var mye enklere ettersom det kun handlet om konvertering til og fra JSON og transaksjoner med databasen.

Frontend utviklingen begynte i mars og var det mest ukjente med prosjektet for prosjektgruppen. Sett i etterkant av koding på frontend kunne planlegging av arkitektur og strukturering av komponenter vært bedre. Allerede ved ferdigstillelse av løsning kan det ses små tegn til teknisk gjeld fordi hierarkiet av CSS ikke prater godt sammen. Dette kommer mest av prosjektgruppens manglende erfaring og planlegging av styling. Det samme gjelder til dels også komponenter utformet underveis gjennom kontinuerlig evaluering sammen med oppdragsgiver. Det er naturlig at komponenter endrer seg over tid og omskriving er noe som må tas hensyn til. Prosjektets begrensede tid gjør det derimot utfordrende å lage flere iterasjoner av hver komponent. Valg av tester ble redesignet gjennom prosjektet og det nye designet kan ses i figur 5.9.

6.2 Konsekvenser

6.2.1 Konsekvens av valgt utviklingsverktøy

For å bygge Backend var prosjektgruppen begrenset til å bruke C# og .NET rammeverket etter ønske fra oppdragsgiver. Hvis utviklingen ikke skulle tatt i bruk .NET og Angular ville det mest sannsynlig ikke vært mulig å fullføre alle kravene fra oppdragsgiver. Uten koden fra konseptbeviset ville det krevd en lengre utviklingsperiode for å nå samme resultat.

For å bygge Frontend ble det valgt å ta i bruk Angular. Dette fordi Uni Micro sin frontend er skrevet i Angular og at det tidligere utviklede konseptbeviset også var skrevet i Angular. Dette gav prosjektgruppen kode som kunne brukes og videreutvikles, samt større mulighet for hjelp knyttet til utviklingen.

Angular viste seg å være ekstremt kraftfullt med mulighet for å lage generiske komponenter som kan gjenbrukes. Prosjektgruppen klarte ikke å ta i bruk all funksjonalitet som eksisterer i Angular, men dette var å forvente med tanke på prosjektgruppens begrensede erfaring med rammeverket. Sammenlignet med om prosjektgruppen skulle ha brukt et annet rammeverk, slik som React, hadde trolig resultatet blitt relativt likt. Det hadde derimot krevd mer tid satt av til utviklingen av Frontend, noe som likevel endte opp med å utgjøre mesteparten av tid knyttet til utvikling.

6.2.2 Konsekvens av valgt rapporteringsløsning

I kapittel 3.6 konkluderes det med at rapporteringsløsningen skal inneholde en backend utviklet etter monolith formatet med en tilleggsmodul som rapporterer i et JSON-format.

Valget av å designe backend etter monolith strukturen har, sammenlignet med et distribuert backend-system slik som en microservice, vært med på å forenklet utviklingsprosessen. I tillegg til at systemet er enklere å forstå og arbeide med vil det også ha en fordel i at det vil være enklere å videreutvikle. Beslutningen ble tatt på bakgrunn av prosjektets størrelse og løsningens omfang. Av den grunn ble det bestemt å ikke gå for en mikrostruktur for å ikke overdimensjonere prosjektet. Dersom løsningen vokser og videreutvikles kan det være hensiktsmessig å endre arkitektur for å håndtere store trafikkvolum. I et slikt tilfelle kan det være gunstig å vurdere en mikrostruktur.

For rapporteringsplugin ble det bestemt å bruke et JSON-format. Dette valget har på samme måte som monolith strukturen i backend, vært med på å forenkle utviklingsprosessen. JSON-formatet er som nevnt tidligere et svært kjent og populært format, noe som også forenkler videreutvikling av systemet. Dersom prosjektgruppen hadde valgt å utvikle en separat rapporteringsplugin ville det ha krevd betydelig mer tid for en relativt liten del av systemet. En egenutviklet løsning ville derimot ha kunne vært bedre tilpasset datamodellene til backend og dermed mer ideelt i et scenario hvor systemet skal skaleres. Det vil derfor kunne være et alternativ til videre arbeid.

6.2.3 Konsekvens av valgt utviklingsmetodikk

Gjennom hele prosjektet har Scrum og Kanban blitt brukt for å fordele oppgaver og holde kontroll på hvilken deler av prosjektet som skal gjennomføres innenfor hver sprint. Sammen med oppdragsgiver har de ukentlige sprintmøtene vært et sted for tilbakemeldinger og en mulighet til å diskutere utviklingsrelaterte utfordringer. Den

kontinuerlige evalueringen og optimaliseringen av prosjektet har vist seg å være svært nyttig ettersom prosjektgruppen møtte på utfordringer underveis i utviklingsløpet og trengte rom for å kunne omprioritere arbeidsoppgaver. Utfordringene har primært handlet om mangel på erfaring og kunnskap knyttet til utvikling i TypeScript.

Sammenlignet med en lineær tilnærmet utviklingsprosess, slik som fossefallsmodellen, ville trolig mer tid ha blitt brukt på å utvikle produktet i en retning som senere hadde måttet endres. I tillegg ville en slik tilnærming begrenset oppdragsgivers mulighet til å bidra og gi tilbakemeldinger etter at selve utviklingsprosessen hadde begynt.

Scrum har likevel hatt noen negative aspekter knyttet til at prosjektgruppen har brukt mye tid på møter og kommunikasjon sammen med oppdragsgiver. Selv om dette i stor grad har bidratt til økt klarhet i prosjektet har det også i perioder vært gjennomført møter hvor det ikke er noe nytt å diskutere eller vise. Disse møtene kunne kanskje vært redusert for å spare tid.

Prosjektgruppen har også benyttet Kanban for å strukturere og fordele oppgaver relatert til rapportskrivingen. Denne delen av prosjektet har hatt en jevn fremdrift hvor prosjektgruppen i oppstartsfasen ble enig om å sette av én dag i uken til skriving. Kanban har gitt gruppemedlemmene god oversikt over oppgaver og gjort det enkelt å se når det skal gjennomgås for å samkjøre delene og holde en rød tråd gjennom rapporten.

6.2.4 Utenforliggende konsekvenser

Prosjektgruppen har også blitt utsatt for utenforliggende risikoer i løpet av prosjektperioden. I tillegg til lengre perioder med sykdom og mye tid tapt til å måtte sette seg inn i språk, rammeverk og teknologier, så anser prosjektgruppen den forsinkede leveransen av løsningsplattformen som den som utgjorde en størst konsekvens for prosjektet.

Som nevnt i kapittel 5.2 resulterte dette i at brukertesten, som var planlagt å gjennomføres i utgangen av april, ikke ble utført før midten av mai. I tillegg var brukertesten planlagt å utføres av flere testoperatører, men ble på grunn av ressursmangel i perioden kun testet av én testoperatør. Dette gjorde til at prosjektgruppen ikke fikk tid til å vurdere tilbakemeldingene fra brukertesten og forbedre løsningen i etterkant.

6.3 Forbedringer

De klareste områdene for forbedring er hvordan tester blir kjørt uten at det blir for mye belastning i systemet og planlegging av utformingen til de ulike komponentene.

Systemet mangler testdekking i form av enhetstester og integrasjonstester. Dette er som nevnt tidligere ikke et krav for oppgaven, men likevel noe systemet kunne trengt.

Prosjektgruppen prøvde å utforme enhetstester uten å få til noe av substans. Mesteparten av koden vil være integrasjonstester ettersom utformingen av klassene ofte kommuniserer med database eller formaterer tilbakemelding fra TestCafe. Det er et område for forbedring i utformingen for å legge til rette for at metoder er separate og kan enhetstestes.

Videre burde det i oppstartsfasen vært satt en spesifikk dato for når brukertesten skulle avholdes. På denne måten hadde både oppdragsgiver og prosjektgruppen hatt et strengere ansvar for å levere i tide. Dette kunne sikret at resultatene fra brukertesten kunne brukes videre i utviklingen. Alternativt burde det blitt tatt opp en diskusjon om midlertidige løsninger når det var kjennskap til problemer med plattformen som løsningen skulle kjøre på. Brukertesten burde i dette tilfellet blitt gjennomført lokalt maskin tidligere. Dette kunne på samme måte sikret at prosjektgruppen fikk resultatene fra brukertesten tidligere og bidratt til videre optimalisering.

Dersom lignende oppgave skal foretas i fremtiden bør prosesser for integrasjon og testing startes langt tidligere enn mot slutten av prosjektet. Det vil spare mye tid på utviklingen da det ikke blir nødvendig å lage midlertidige løsninger for å kunne fortsette med resterende utvikling.

Avslutningsvis, burde prosjektgruppen ha hevet sin generelle kompetanse innen utviklingsverktøyene i forkant utviklingen. Dette gjelder hovedsakelig for Typescript og Angular utvikling i frontend. Dersom alle gruppemedlemmene hadde større kjennskap til språk og rammeverk i forkant av utviklingsfasen ville det trolig kunne blitt dedikert flere effektive timer til utviklingen. Dette kunne også indirekte gitt mer tid til gjennomføringen av andre utenforliggende oppgaver.

7 KONKLUSJON OG VIDERE ARBEID

7.1 Konklusjon

Prosjektets hovedmål var å videreutvikle en webapplikasjon som kunne være med på å forenkle prosessen knyttet til regresjonstesting av klienter. Løsningen som er utviklet inneholder en backend med et API som gjennom sine endepunkter tilbyr funksjonalitet for oppstart av tester, oppretting av rapporter, henting av resultater, registrering og sletting av tester. APIet blir tatt i bruk av et brukergrensesnitt som tillater en bruker å dra nytte av disse.

Løsningen forenkler testprosessen knyttet til regresjonstesting med sin mulighet for parallell testing og rapportering. Ettersom tester kan startes i selvdefinerte grupper kan en kjøring enkelt inneholde kun nødvendige tester for å teste en endret del av nettsiden eller kjøre alle tester. Løsningen er et godt utgangspunkt for videre utvikling ettersom all sentral funksjonalitet er implementert og testet.

Sett opp mot problemstillingen anses prosjektets endelige løsning som et godt løsningsforslag, hvor alle initielle krav har blitt nådd. Oppdragsgiver har gitt uttrykk for å være svært fornøyd med det leverte produktet, selv med kjennskap til områder som fremdeles har rom for forbedring.

7.2 Videre arbeid

Selv om Løsningen oppfyller målene som var satt i forkant av prosjektet, er det likevel kjente ønsker om forbedring. Dersom løsningen skal videreutvikles med et utgangspunkt i slik den er i dag bør videre arbeid forsøke å oppfylle disse ønskene. Det bør også settes av tid til enhetstester og integrasjonstester. Av disse er de mest sentrale styling, enhetstester og integrasjonstester, integrasjon mot TestRail og forslag til forbedringer fra brukertesten, omdiskutert i kapittel 5.2.

Styling av dashbordet kan enten gjøres ved å tilrettelegge HTML for stilpakken som eksisterer til Uni Micro sin klient eller designe en stil fra bunnen av. Dette arbeidet vil inkludere UX designere og andre personer for å finne en riktig stil som gjør brukeropplevelsen bedre. Med dette inkluderes det hvor enkelt det er å forstå de ulike flytene som var beskrevet i kravdokumentasjonen kapittel 2 (vedlegg A).

Det neste steget i utvikling av løsningen vil være å sikre kodekvalitet ved hjelp av enhetstester og integrasjonstester. Det vil også være en mulighet å lage tester ved hjelp av TestCafe til å teste systemet. Da vil systemet kunne kjøre tester på seg selv og lage en statusrapport på sin egen kvalitet.

Om løsningen skal skaleres vil det bli naturlig å ta stilling til arkitekturen i Backend med tanke på hvorvidt det er nødvendig eller ønskelig å gå fra en monolith til en microservice

arkitektur. I en slik sammenheng vil en microservice være mer egnet for et større team med utviklere, eller flere team, som har vært sitt ansvarsområde for utvikling. Da vil det også være mulig å se på om et arkitekturskifte vil gi applikasjonen en boost i effektivitet.

TestCafe sin rapporteringsplugin er også et punkt som kan være verdt å se nærmere på. Det kan være data tilgjengelig i TestCafe som kan brukes på nye måter og gi løsningen et løft i innsyn på testing. På lengre sikt kan det også ses på om det er noe å hente på å lage en egen pakke for testing for å maksimere løsningens gevinst, samt unngå noen av manglene til TestCafe i form av begrensninger.

Avslutningsvis vil integrasjon mot TestRail være et viktig steg for å gjøre løsningen om til et produkt. Ved å kunne hente tester fra TestRail vil det på sikt bli mulig å flytte ytterligere arbeidsflyt fra å være manuell testing og manuell rapportering til en automatisk prosess som tester og finner feil, men manuell oversikt og gransking av feil som dukker opp. Integrasjon med andre verktøy åpner også døren for å kunne legge verktøyet til i pipelinen for koden til Uni Micros frontend. Ved å automatisk kjøre alle tester før det bygges til produksjon vil det øke tillit til at produktet som leveres av Uni Micro er av høyeste kvalitet.

8 REFERANSER

Adobe. (2022). *Waterfall Methodology: A Complete Guide.* https://business.adobe.com/blog/basics/waterfall (Hentet: 22.03.2023)

Beck, K., Beedle, M., Bennekum, A. v., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for Agile Software Development.* https://agilemanifesto.org/ (Hentet: 22.03.2023)

BrowserStack. (udatert). *Getting Started with TestCafe Framework: Tutorial.* https://www.browserstack.com/guide/testcafe-framework-tutorial (Hentet: 24.04.2023)

Croak, D. (udatert). *Four-Phase Test.* https://thoughtbot.com/blog/four-phase-test (Hentet: 16.01.2023)

Dhaduk, H. (udatert). *Angular vs React 2023: Which Framework to Choose for Your Project?* https://www.simform.com/blog/angular-vs-react/ (Hentet: 18.03.2023)

G2. (udatert). *Best Data Visualization Libraries Software.* https://www.g2.com/categories/data-visualization-libraries?utf8=%E2%9C%93&order=g2_score (Hentet: 10.05.2023)

Google. (udatert). *Angular.* https://angular.io/ (Hentet: 3.02.2023)

Hamilton, T. (udatert-a). *Integration Testing: What is, Types with Example.* https://www.guru99.com/integration-testing.html (Hentet: 16.01.2023)

Hamilton, T. (udatert-b). *What is Regression Testing? Test Cases (Example).* https://www.guru99.com/regression-testing.html (Hentet: 16.01.2023)

Hamilton, T. (udatert-c). *What is System Testing? Types with Example.* https://www.guru99.com/system-testing.html (Hentet: 16.01.2023)

Harris, C. (udatert). *Microservices vs. monolithic architecture.* https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith (Hentet: 18.03.2023)

Microsoft. (2022). *.NET vs .NET Framework.* https://learn.microsoft.com/en-us/dotnet/standard/choosing-core-framework-server (Hentet: 08.02.2023)

Microsoft. (udatert-a). *Introduction to SignalR.* https://learn.microsoft.com/en-us/aspnet/signalr/overview/getting-started/introduction-to-signalr (Hentet: 06.04.2023)

Microsoft. (udatert-b). *A tour of C# - Overview.* https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/ (Hentet: 3.02.2023)

Microsoft. (udatert-c). *TypeScript: JavaScript With Syntax For Types.* https://www.typescriptlang.org/ (Hentet: 3.02.2023)

Microsoft. (udatert-d). *Visual Studio: IDE and Code Editor for Software Developers and Teams.* https://visualstudio.microsoft.com/ (Hentet: 3.02.2023)

Microsoft. (udatert-e). *Walkthrough: Create and run unit tests for managed code.* https://learn.microsoft.com/en-us/visualstudio/test/walkthrough-creating-and-running-unit-tests-for-managed-code?view=vs-2022 (Hentet: 16.03.2023)

Mozilla. (udatert). *HTTP response status codes.* https://developer.mozilla.org/en-US/docs/Web/HTTP/Status (Hentet: 19.03.2023)

Nikoloff, G. (udatert). *Too Many SVGs Clogging Up Your Markup? Try 'use'.* https://css-tricks.com/too-many-svgs-clogging-up-your-markup-try-use/ (Hentet: 10.05.2023)

Node.js. (udatert). *Node.js.* https://nodejs.org/en (Hentet: 18.03.2023)

Oracle. (udatert). *Introduction to Comet.* https://docs.oracle.com/cd/E19798-01/821-1752/ggrgy/index.html (Hentet: 07.04.2023)

React. (udatert). *React - The library for web and native user interface.* https://react.dev/ (Hentet: 18.03.2023)

Rehkopf, M. (2022). *What is a kanban board?* https://www.atlassian.com/agile/kanban/boards (Hentet: 22.03.2023)

Selenium. (udatert). *Webdriver | Selenium.* https://www.selenium.dev/documentation/webdriver/ (Hentet: 16.05.2023)

Split. (udatert). *Client Side Testing.* https://www.split.io/glossary/client-side-testing/ (Hentet: 16.01.2023)

VisualParadigm. (udatert). *What is a Sprint in Scrum.* https://www.visual-paradigm.com/scrum/what-is-sprint-in-scrum/ (Hentet: 10.03.2023)

9 VEDLEGG

Alle vedlegg er tilgjengelig som eksterne dokumenter.