Høgskulen på Vestlandet

BACHELOROPPGAVE

En moderne tech-stack i .NET Core

A modern tech stack built on .NET Core

Øystein Knudsen

Sondre Larsen Ovrid

Dataingeniør og Informasjonsteknologi
Institutt for data- og realfag
Fakultet for ingeniør- og naturvitskap

Leveringstidspunkt: 02.06.2019

Høgskulen på Vestlandet

Institutt for data- og realfag
Fakultet for ingeniør- og naturvitskap

TITTELSIDE FOR HOVEDPROSJEKT

Rapportens tittel:	*Dato:*
En moderne tech-stack i .NET Core A modern tech stack built on .NET Core	02.06.2019
Forfatter(e):	*Antall sider u/vedlegg:* 51
Sondre Larsen Ovrid og Øystein Knudsen	Antall sider vedlegg: 7
Studieretning:	*Antall disketter/CD-er:*
Informasjonsteknologi/ Dataingeniør	Ingen
Kontaktperson ved studieretning:	*Gradering:*
Richard Kjepso	Ingen
Merknader:	

Oppdragsgiver:	*Oppdragsgivers referanse:*
Evry AS avd. Bergen	
Oppdragsgivers kontaktperson:	*Telefon:*
Patricio Alexander Castillo	478 43 033

Sammendrag:

Målet med dette prosjektet har vært å utvikle et referanseprosjekt som løser tradisjonelle problemstillinger i forbindelse med programvareutvikling. Gitt et sett med moderne teknologier har prosjektgruppen løst disse problemstillingene på en måte som fremhever god praksis for programvareutvikling. Sluttresultatet er en «verktøykasse» som kan benyttes av nye utviklere som ønsker bedre oversikt over de gitte teknologiene, og som grunnmur i oppstart av nye utviklingsprosjekter.

The goal of this project has been to develop a template project which implements solutions for common problems in the context of software development. Given a set of modern technologies the project group have solved these problems in a way which promotes good practice for software development. The end result is a «toolbox» which can be utilized by developers seeking an overview of the given technologies, as well as a foundation for new software development projects.

Stikkord:

Referanseprosjekt	Tech-stack	.NET Core	RESTful WebAPI	Bygging og utrulling	React

Forord

Denne rapporten er ment som dokumentasjon av arbeidet knyttet til prosjektet «En moderne Tech-stack i .NET Core». Prosjektarbeid og rapportskriving er utført av Sondre Larsen Ovrid og Øystein Knudsen, ved Høgskulen på Vestlandet våren 2019.

Vi ønsker å takke vår oppdragsgiver, EVRY, for å ha gitt oss et spennende og lærerikt prosjekt å basere bacheloroppgaven vår på. Videre ønsker vi også å rekke en stor takk til våre kontaktpersoner, Patricio Alexander Castillo, Tom Solem og Oddmar Sandvik, som har gitt oss god veiledning undervegs i prosjektarbeidet.

Til slutt vil vi få rette en takk til Richard Kjepso for veiledningen han har gitt oss i forbindelse med utarbeiding av rapporten.

Øystein Knudsen

Øystein Knudsen

Sondre Larsen Ovrid

Innholdsfortegnelse

1 INTRODUKSJON

1.1 Mål og motivasjon

Prosjektet gjennomføres av to studenter ved henholdsvis dataingeniørstudiet og informasjonsteknologistudiet ved Høgskulen på Vestlandet. Formålet med selve prosjektarbeidet er at deltakerne skal jobbe selvstendig med å utforme og implementere en løsning for en gitt informasjonsteknologisk problemstilling, samt dokumentere prosessen gjennom en prosjektrapport. I prosjektet søker deltakerne å benytte kunnskap opparbeidet gjennom studiet og anvende denne i de ulike fasene av prosjektet. Samtidig er det et ønske om å undersøke nye verktøyer, teknologier og utviklingsmetoder som kan være relevante for problemstillingen. Et viktig poeng for gruppen er at oppgaven tillater å ta i bruk kompetanse og erfaringer fra samtlige prosjektdeltakere sine spesialiseringer for utdanningen. Basert på disse vurderingene falt valget på oppgaven tilbudt av EVRY Norge avd. Bergen. Gruppen mener oppgaven som EVRY tilbyr gir god dybde innenfor deltakernes kompetanseområder, i tillegg til å by på interessante og ukjente problemstillinger.

EVRY ønsker at prosjektgruppen skal utvikle et referanseprosjekt basert på et utvalg av teknologier. Dette skal videre kunne benyttes både som et verktøy for intern opplæring men også som et utgangspunkt for videre utvikling av kundeprosjekter. Referanseprosjektet skal inneholde løsninger og implementasjoner for et gitt sett med typiske problemstillinger i forbindelse med programvareutvikling. Målet er således å utvikle et slikt referanseprosjekt, som løser de gitte problemstillingene, og som legger til rette for at det enkelt kan brukes av både utviklere og i konteksten av utviklingsprosjekter.

1.2 Kontekst

EVRY er et stort internasjonalt konsulentselskap med mange ansatte som stadig blir eksponert for nye teknologier og problemområder. Dette kan skje både gjennom nyansettelser hvor utviklere blir plassert på sitt første prosjekt, eller gjennom intern prosjektrullering. I disse fasene er det kritisk at utforming og implementasjon igangsettes raskest mulig, da prosjektkostnadene løper allerede fra første dag. De ansatte er derfor helt avhengige av å ha gode interne rutiner og verktøy for å tilegne seg den nye kompetansen som behøves. Samtidig er det nyttig å kunne gjenbruke deler av eller hele løsninger fra tidligere prosjekter.

Mange verktøy og rutiner er allerede utviklet, men utviklingen i IT-bransjen går fort. Det er stadig behov for å oppdatere og utvikle nye verktøy etterhvert som behovene i bransjen endres. Problemstillinger som skal løses i IT-bransjen krever ofte en sammensetning av flere verktøy og teknologier. En sammensetning av teknologier som løser et gitt problem blir ofte referert til som en tech-stack.

1.3 Begrensninger

Prosjektgruppen er i hovedsak begrenset av tre ting; timeplan, spesialiseringer og erfaring. Samtlige prosjektdeltakere har arbeid ved siden av studiene til ulike tidspunkter. Dette fører til at det til tider kan være vanskelig å finne passende tidspunkt hvor gruppen kan fysisk samarbeide på prosjektet. De forskjellige timeplanene kan resultere i at prosjektgruppen har begrensede muligheter til å diskutere løsninger på problemer som dukker opp fortløpende i prosjektet. Videre har gruppemedlemmene også forskjellige spesialiseringer. Dette gjør at gruppen ikke nødvendigvis har godt nok utgangspunkt til å diskutere problemer knyttet til hverandres spesialisering for å finne en god løsning.

Prosjektgruppen har lite erfaring med programvareutvikling på profesjonelt nivå. Problemstillingen som skal løses har fokus på beste praksis for utvikling av prosjekter i de gitte teknologiene. På dette punktet begrenses gruppen av manglende erfaring med utvikling i disse teknologiene. Dette kan være spesielt utfordrende med tanke på teknologiene knyttet til for eksempel kontinuerlig integrasjon, et prinsipp som blir brukt i større utviklingsteam, men hvor prosjektgruppen bare har erfaring med prosjekter på mindre skala.

1.4 Ressurser

1.4.1 Dokumentasjon

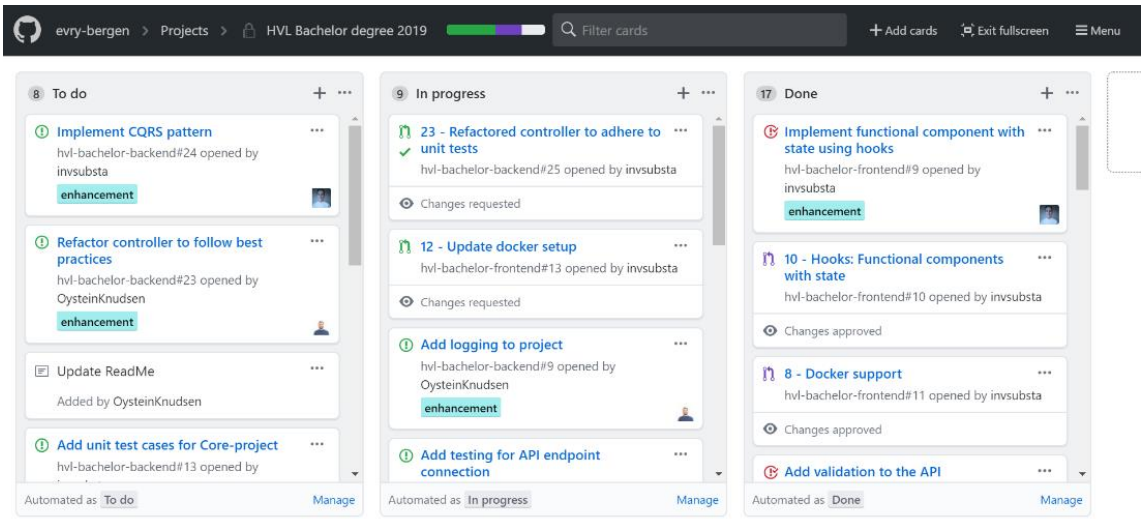
Prosjektgruppen har fått i oppdrag å lage et prosjekt som skal brukes som utgangspunkt i fremtidige prosjekter. Her vil det være fokus på å gjennomføre ting på best mulig måte. Prosjektdeltakerne har i utgangspunktet ikke så mye erfaring å basere seg på og vil være avhengig av informasjon om fremgangsmåter. Prosjektet bygger på en samling utbredte og modne teknologier som er godt dokumentert av de som har utviklet teknologiene, i tillegg til programvareutviklere som har tatt disse teknologiene i bruk. Denne dokumentasjonen vil være en viktig ressurs for gruppen i gjennomføring av prosjektet.

1.4.2 Veiledere

Oppdragsgiver er en teknologibedrift som stiller et bredt og dyktig kompetansemiljø til rådighet. Prosjektgruppen har blitt tildelt en kontaktperson i EVRY som vil være til hjelp ved behov, i tillegg til at andre ansatte med mye erfaring vil være tilgjengelig underveis. Gruppen har også blitt tildelt en veileder fra HVL som ikke bare har teknisk erfaring men også flere års erfaring med gjennomføring av tilsvarende oppgaver. Dette vil være en nyttig ressurs dersom prosjektdeltakerne har tekniske og formelle spørsmål knyttet til oppgaven.

1.4.3 Prosjektorganisering og oppgavehåndtering

I gjennomføring av prosjektet vil det være hensiktsmessig å ha god oversikt over de ulike oppgavene og deloppgavene. Gruppen har valgt å ta i bruk Github Project Boards som en tjeneste for oppgavehåndtering. Her vil det være mulighet for å definere oppgaver og delmål, samt sette status på oppgaver, med mer. Dette vil gi prosjektdeltakerne en god oversikt over fremgangen i prosjektet og hvilke oppgaver som til enhver tid skal gjøres.



Figur 1.1 - GitHub Project Boards

1.5 Rapportstruktur

Rapporten har som formål å beskrive prosessen med utviklingen av prosjektet, fra initiell problemstilling og motivasjon til sluttresultat. Rapporten følger en generell standard for akademiske oppgaver som spesifisert av Høgskulen på Vestlandet. Strukturen er som følger:

Kapittel 1 – Beskriver motivasjonen for selve prosjektet og dets mål, samt hvilke begrensninger og ressurser som er relevante.

Kapittel 2 – Presenterer prosjekteiers innledende krav til prosjektet og hvordan prosjektets resultater er av verdi for prosjekteier.

Kapittel 3 – Tar for seg ulike alternative tilnærminger og valg av endelig løsning. Planlegging og organisering av prosjektet samt risikohåndtering og prosjektevalueringer blir også diskutert.

Kapittel 4 – Her blir utforming og implementasjon av løsningen presentert i detalj. Et av hovedpoengene er å klarlegge hvordan ønskede resultater blir realisert.

Kapittel 5 – Belyser metoder som blir brukt for evaluering samt en evaluering av selve prosjektresultatet.

Kapittel 6 – Beskriver prosjektresultatet i detalj.

Kapittel 7 – Forklarer konsekvensene av tilnærmingen som ble valgt i prosjektet. Her blir det diskutert hvordan prosjektdeltakernes valg påvirket resultatet, og hva som eventuelt kunne blitt endret i lys av ny informasjon og revurdert tilnærming.

Kapittel 8 – Oppsummerer prosjektets mål og konkluderer hvorvidt måloppnåelsen var i samsvar med disse. Belyser også hvordan prosjektresultatet eventuelt kan brukes i videre arbeid.

Kapittel 9 – Liste over referanser som er brukt i prosjektet.

Kapittel 10 – Appendikser.

2 PROSJEKTBESKRIVELSE

2.1 Praktisk bakgrunn

2.1.1 Prosjekteier

Prosjektets eier er EVRY, et ledende nordisk IT og -programvareselskap med mer enn 10 000 kunder i privat og offentlig sektor. Hovedkontoret ligger på Fornebu utenfor Oslo og selskapets aksjer er notert på Oslo Børs. Som Norges største IT-selskap har EVRY omfattende leveranser til norsk og nordisk næringsliv, finanssektoren og offentlige virksomheter innen stat, kommune og helsesektor (EVRY NORGE AS, 2019). EVRY har ansvaret for omtrent en tredel av alle IT-tjenesteleveranser i Norge og har et stort antall kunder både i offentlig sektor og privat næringsliv, som for eksempel: DNB, Telenor, Posten Norge, Sparebank 1 Gruppen, Statoil, Hydro, Storebrand, Gjensidige, Oslo Kommune, Trondheim Kommune og NAV (EVRY NORGE AS, 2019). Våre kontaktpersoner i EVRY sitter på kontoret deres i Bergen, som er lokalisert i Fyllingsdalen.

Figur 2.1 - EVRY sin logo

2.1.2 Opprinnelig kravspesifikasjon

Oppdragsgiver er ikke endelig i kravene sine, men har noen idéer om hva de ønsker som utgangspunkt. Prosjektet har som hensikt å løse klassiske problemstillinger i forbindelse med programvareutvikling. Som minimum ønsker oppdragsgiver at følgende problemstillinger løses i oppgaven:

- Prosjektstruktur
- Autentisering
- Logging
- Databasehåndtering
- Bygging
- Utrulling

Oppdragsgiver har satt øvrige krav om å utvikle et referanseprosjekt i .NET Core som følger god kodepraksis, løser de nevnte problemstillingene, benyttes og som er lett gjenbrukbart for fremtidige prosjekter.

2.1.3 Opprinnelig løsning

Etter rådgivning fra oppdragsgiver ble det i fellesskap utarbeidet en opprinnelig løsning hvor følgende teknologier og prinsipper tas i bruk for å løse problemstillingene:

- En prosjektstruktur med fokus på modularitet hvor det lett kan byttes ut en modul uten å gjøre omfattende endringer i andre moduler i applikasjonen.
- Et Web API utviklet I ASP.NET Core
- Selvdokumentasjon ved hjelp av Swagger
- Databasehåndtering ved hjelp av Entity Framework
- Datalagring i MS SQL
- Automatisk vedlikehold av database ved hjelp av EF Migrations
- Kontinuerlig Integrasjon ved hjelp av CircleCI
- Mulighet for kontainerisering ved hjelp av Docker
- Organisering av kontainere ved hjelp av Kubernetes

Teknologiene og prinsippene er ikke endelige, men heller et utgangspunkt for prosjektgruppen i oppstart av utvikling. Prosjektdeltakerne ønsker å være fleksible under utviklingen og dersom bedre egnede løsninger blir utarbeidet, vil deltakerne være åpne for endring.

3 PROSJEKTUTFORMING

3.1 Mulige tilnærminger

Basert på problemstillingen til oppdragsgiver kan utformingen av den overordnede løsningen deles i to hovedpunkter:

- valg av generell systemarkitektur og prosjektstruktur
- valg av domene og domenemodell

Tradisjonelle programvareprosjekter tar som regel for seg et spesifikt problemdomene skal modelleres og utvikles funksjonalitet for. I konteksten av dette prosjektet er det derimot ikke gitt et konkret problemdomene, noe som avgrenser mulige tilnærminger. Typisk vil et problemdomene i stor grad påvirke den endelige arkitekturen til et tradisjonelt programvareprosjekt, og det er gjerne flere nyanser i valg av endelig arkitektur og utforming av tilhørende funksjonalitet. For prosjektgruppen sin del vil det være vanskelig å kunne ta de samme vurderingene da et fremtidig prosjekt som bygger på referanseprosjektet kan variere i utforming. Som tidligere forklar er også mange utformings- og teknologispesifikke valg allerede tatt på vegner av gruppen.

Det prosjektdeltakerne derimot kan bedømme er hvordan ulike arkitekturer måler seg opp mot hverandre basert på faktorer som smidighet, gjenbrukbarhet, og skalerbarhet. Programvarearkitektur er et komplekst fagfelt som krever mye erfaring fra generell programvareutvikling. Prosjektgruppen vil ikke sette av ressurser til å analysere ulike arkitektoniske utformingsmønstre, da dette blir utenom omfanget av oppgaven. I stedet vil gruppen legge utvalgt litteratur til grunn for de arkitektoniske valgene.

Et viktig hovedpunkt i oppdragsgivers problemstilling er at et referanseprosjekt må kunne tas raskt i bruk i ulike prosjekter. For å gjøre systemet gjenbrukbart kan det være en fordel at å benytte en modulær tilnærming til utviklingen. Dette betyr at et system utformes som en sammensetning av flere selvstendige programvareenheter. De enkelte enhetene har videre en veldefinert oppgave, er uavhengig av andre enheter og er enkle å substituere. Resultatet av dette er høyere potensiale for gjenbruk i andre prosjekter, da de ulike enhetene benyttes basert på hva behovet. Samtidig skal referanseprosjektet kunne fungere som et opplæringsverktøy. Basert på disse vurderingen har prosjektgruppen valgt å fokusere på smidighet og skalerbarhet som primærfaktorer i valg av arkitektur, samt at arkitekturen må legge til rette for modulbasert utvikling.

3.1.1 Alternative tilnærminger: Spesialisert arkitektur

Mange programvareprosjekter faller innen et avgrenset utvalg av mye utbredte arkitekturer. Noen av de vanligste arkitekturene i dag er blant annet lagdelt-, hendelsesdreven, rombasert-, mikrokjerne- og mikrotjenestearkitektur (Richards, 2015). Det disse har til felles er at de typisk passer spesielt godt for spesifikke

programvaresystemer og -applikasjoner, og derfor et spesifikt problemdomene. Siden disse arkitekturene er mye i bruk i dag, blir de også til stadighet sammenlignet basert på faktorer som for eksempel smidighet og skalerbarhet. I følge Richards scorer spesielt mikrotjeneste- og rombasert arkitektur høyt på flere av de faktorene som prosjektgruppen har lagt til grunn for oppgaven.

3.1.2 Alternative tilnærminger: Arkitektur basert på generelle prinsipper

Ikke alle problemstillinger lar seg løse med spesialiserte verktøy, noe som i stor grad også gjelder for programvaresystemer. Når dette er tilfelle kan det være nyttig å basere seg på et sett med prinsipper for god utforming. Et sett med prinsipper som er mye brukt i denne sammenhengen, ofte referert til som «SOLID», har sitt utspring i objektorientert analyse og -utforming. Prinsippene har som formål å utforme programvare på en måte slik at vedlikehold, fleksibilitet og forståelighet blir vektlagt (Martin, Agile Software Development, Principles, Patterns, and Practices, 2002). Disse prinsippene har videre lagt grunnlaget for et rammeverk for utarbeidelse av gode programvarearkitekturer, kalt «Clean Architecture» (Martin, Clean Architecture, 2018).

3.1.3 Diskusjon og valg av tilnærming

Fordelen med å velge en spesialisert programvarearkitektur er først og fremst at disse er godt utprøvde og at de har blitt nøye analyserte basert på ulike kriterier. På den andre siden er det fare for at å følge en slik arkitektur slavisk vil medføre begrensninger, siden de i stor grad er tilpasset spesifikke problemområder. Dette er et spesielt viktig moment siden det er høy sannsynlighet for at prosjektgruppens implementasjonen vil måtte tilpasses etter hvert. Dersom det blir tilfelle at arkitekturen må endres undervegs, vil dette medføre ekstra ressursbruk i overgangen til en ny arkitektonisk tilnærming.

Clean Architecture sitt rammeverk er i større grad fleksibelt, siden det i stedet baseres på et sett av prinsipper. Prosjektdeltakerne ser for seg at denne tilnærmingen vil kunne ta høyde for tilpasningene som gruppen sannsynligvis må gjøre undervegs, sammenlignet med en spesialisert programvarearkitektur. Samtidig er gruppen selvsikre på at rammeverkets prinsipper ivaretar de viktige momentene som spesifisert av oppdragsgiver og deres problemstilling.

3.2 Spesifikasjon

Som nevnt innledningsvis er prosjektgruppen gitt relativt frie tøyler i selve utformingen og implementasjonen av referanseprosjektet. De teknologimessige og en del av de utformingsmessige kravene som er stilt er derimot konkrete. Etter videre dialog med oppdragsgiver har det blitt gjort noen konkretiseringer utover det som opprinnelig lå til grunn. Blant annet er det blitt spesifisert at referanseprosjektet skal fremstå som en kjørbar applikasjon, som heretter vil refereres til som serverapplikasjonen. Med dette menes at programvaren som implementeres skal kunne kjøres og interageres med gjennom en klient og tilby en enkel dataflyt som involverer de øvrige programvareenhetene. Det skal også utvikles en enkel nettleserbasert klientapplikasjon som kan kommunisere med serverapplikasjonen.

3.2.1 Serverapplikasjonen

Valg av domene, domenemodell og tilhørende entiteter er opp til prosjektdeltakerne å bestemme. Det samme gjelder datamodellen og dens tilhørende representasjon i databasekonteksten. Det foreligger ingen spesielle krav til dette utover det som allerede er nevnt om modularitet, gjenbrukbarhet og mulighet til utvidelse og videre påbygging på sluttproduktet. Prosjektdeltakerne er blitt oppfordret til å implementere forretnings- og applikasjonsspesifikk logikk som tillater interaksjon med applikasjonen og dataflyt internt, og som kan fungere som et mellomledd mellom de øvrige modulene som skal implementeres. Dette må baseres på valgt domene og domenemodell.

Databasetilkobling skal implementeres ved hjelp av Microsoft Entity Framework Core som en modulær, frittstående enhet i den overordnede applikasjonen. Dette skal videre kunne aksesseres fra en databaseimplementasjon gjennom et eget grensesnitt. Hva gruppen velger av databaseimplementasjon er likegyldig så lenge dette er i samsvar med databasetilkoblingens grensesnitt.

Videre skal det implementeres et web-basert programmeringsgrensesnitt for applikasjonen (webAPI). Grensesnittet skal følge REST som overordnet arkitektonisk utformingsmodell og benytte HTTP til kommunikasjon, og må kunne nåes av en vilkårlig klient som støtter HTTP. En tilhørende frittstående web-server modul må implementeres i applikasjonen, med formål om å synliggjøre grensesnittet for internett.

3.2.2 Klientapplikasjonen

Klientapplikasjonen skal baseres på et komponentbasert rammeverk for utforming av web-brukergrensesnitt. Applikasjonen har som formål å illustrere interaksjon med webAPI-et til serverapplikasjonen. Som minimum har prosjektgruppen i enighet med oppdragsgiver landet på at navigasjon, tilstandshåndtering og styling skal implementeres. Klientapplikasjonen skal kunne kjøres uavhengig av serverapplikasjonen.

3.3 Valg av verktøy og programmeringsspråk

3.3.1 ASP.NET Core

Selve webAPI-et vil bli utviklet med ASP.NET Core, som er et kryssplatform og «åpen kildekode»-rammeverk for utvikling av moderne webapplikasjoner (Microsoft, 2019). Rammeverket tilbyr mange fordeler som tilrettelegger for at prosjektgruppen kan møte kravspesifikasjonene som er gitt av oppdragsgiver. Rammeverket tilbyr blant annet automatisk validering i endepunkter og avhengighetsinjeksjon for å lettere bygge en modulær applikasjon med utbyttbare moduler.

3.3.2 Swagger

For at en klient enkelt skal kunne ta i bruk og konsumere et API er det viktig at API-ets funksjonalitet og oppsett er godt dokumentert. WebAPI-er er ofte i rask utvikling og det kan derfor være utfordrende å kontinuerlig oppdatere dokumentasjonen til API-et manuelt i takt med utviklingen. For å løse dette problemet benyttes et verktøyet Swagger. Swagger tilbyr automatisk generering av dokumentasjon for API-er basert på hvilke endepunkter som er satt opp, med tilhørende data og spørringer. Med andre ord så vil API-et i serverapplikasjonen være selvdokumenterende.

3.3.3 Entity Framework

For databasehåndtering brukes Microsoft Entity Framework (EF). Dette er et rammeverk som åpner for muligheten til å jobbe med data i form av domenespesifikke objekter uten å måtte fokusere på den underliggende databasen hvor dataen er representert som tabeller og kolonner. Resultatet er at arbeidet med utforming og vedlikehold av applikasjonen krever mindre programkode enn i tradisjonelle applikasjoner, samtidig som den underliggende datalagringen er abstrahert bort fra programutviklere (Microsoft, 2013).

Et annet viktig punkt her er at EF støtter et bredt utvalg av underliggende databasetjenester (Microsoft, 2018). Det vil si at i fremtidige prosjekter kan databasetjeneren enkelt substitueres ut til fordel for en annen databasetjener dersom ønskelig.

3.3.4 Entity Framework Migrations

Under utvikling av nye applikasjoner er det høy sannsynlighet for at datamodellen vil endres hyppig. Hver gang en endring blir gjort i datamodellen må databasen oppdateres tilsvarende. Tradisjonelt sett så ville en utvikler ha løst dette ved å slette eksisterende databaseskjema, lage et nytt som stemmer med den nye datamodellen for så å legge inn testdata. Problemet med denne fremgangsmåten er at data som ligger i databasen går tapt. Dette kan være data som i utgangspunktet skal beholdes. For å løse dette har Microsoft utviklet et verktøyet EF Migrations. Dette verktøyet gir utviklere muligheten til å gjøre endringer i

datamodellen uten å gå gjennom den tradisjonelle prosessen med å slette og gjenskape databasen. EF Migrations vil automatisk håndtere oppdatering av databasen uten å sette opp databaseskjemaet på nytt.

3.3.5 Microsoft SQL Server

Det finnes mange alternativer til databasesystem. Mest vanlig i dag er kanskje relasjonsbaserte varianter. Men det finnes også databasesystemer som baserer seg på ikke-relasjonsbaserte strukturer av data. Prosjektgruppen har gjennom sitt studie i stor grad brukt relasjonsbaserte databasesystemer og vil derfor bruke det relasjonsbaserte databasesystemet, «Microsoft SQL Server» i prosjektet.

3.3.6 React JS

React er et moderne og mye brukt komponentbasert programmeringsbibliotek med fokus på utvikling av brukergrensesnitt for nettleserapplikasjoner. Biblioteket er utviklet av Facebook og bygger på JavaScript. JavaScript blir regnet som dagens standard for utvikling av dynamiske nettsider og applikasjoner for nettlesere. JavaScript er et skriptingspråk med stor utbredelse og støttes av tilnærmet samtlige nettlesere må markedet i dag. JavaScript blir, i kombinasjon med HTML og CSS, regnet som et av hovedverktøyene for utvikling av nettleserbaserte applikasjoner. Utvikling i React foregår derimot som regel i Facebook sin utvidelse til JavaScript, kalt JSX. JSX gjør utviklere i stand til å implementere brukergrensesnitt deklarativt, i motsetning til JavaScript sin imperative syntaks.

3.3.7 Docker

For utrulling av applikasjonene har det blitt besluttet å bruke kontainerteknologi. En kontainer i IT-sammenheng er en programvareenhet som pakker programkode sammen med alle dens avhengigheter. Docker har blitt valgt som verktøy for enklere håndtering og konfigurering av kontaineriseringen i denne oppgaven. Docker pakker en applikasjon til en kontainer som inkluderer alt som trengs for å kjøre applikasjonen: programkode, runtime, systemverktøy, systembiblioteker og innstillinger. En kontainer vil isolere programvaren fra dens miljø for å sikre at den alltid fungerer likt uten å bli påvirket av miljøet den blir installert på.

3.4 Utviklingsmetode

3.4.1 Utviklingsmetode

Som et lite utviklingsteam på bare to personer og et relativt lite prosjekt, ønsker prosjektgruppen en lett utviklingsmetodikk som tillater prosjektdeltakerne å fokusere på programvareutviklingen. Det finnes mange forskjellige rammeverk for gjennomføring av programvareutviklingsprosjekter. Gruppen har sett på flere muligheter som SCRUM, RUP, XP og Kanban. Det har blitt besluttet at Kanban er passende til gruppens formål. Sammenlignet med de øvrige utviklingsmetodikkene er Kanban lite ressurskrevende å implementere i et utviklingsteam. Kanban gir

prosjektgruppen et sentralisert verktøy for arbeidsfordeling og visualisering av arbeidsflyt. Prosjektdeltakerne har tro på at dette vil være gunstig for å maksimere produktiviteten.

Kanban sin kjerne er en tavle som blir brukt for visualisering av arbeid og optimalisering av arbeidsflyten innad i et utviklingsteam. En standard Kanban-tavle har en tre-stegs arbeidsflyt: «Å gjøre», «under arbeid» og «ferdig». Metodologien baserer seg på full åpenhet og tavlen bør bli sett på som den gjeldende representasjonen av gruppen sitt arbeid. GitHub sin tjeneste kalt Project Boards vil bli brukt til dette formålet.

3.4.2 Prosjektplan

Prosjektplanen kan grovt sett deles i tre faser: planlegging, implementasjon og analyse. En mer detaljert plan er illustrert som et GANTT-diagram i Appendiks B. Planleggingsfasen består av flere aktiviteter men med hovedfokus på å få en god forståelse av kravspesifikasjonen gitt av oppdragsgiver samt utforskning av potensielle løsninger. Målet for denne fasen er å få oversikt og forståelse for oppgaven samt å gjøre nødvendige teknologi- og arkitekturvalg for grunnmuren til applikasjonen.

Implementasjonsfasen vil i hovedsak bestå av implementasjon av de valgene som ble gjort i planleggingsfasen. Prosjektgruppen vil her utvikle applikasjonen iterativt og opprettholde god kommunikasjon med oppdragsgiver for å sikre at ønskede resultater oppnås.

I avsluttende fase vil prosjektdeltakerne gjennomføre en evaluering av prosjektet i henhold til valgt evalueringsmetode som beskrevet i 3.5. Etter gjennomført evaluering av prosjektet vil fokus rettes mot ferdigstilling av prosjektrapporten.

3.4.3 Risikovurdering

For få prosjektdeltakere

Siden prosjektgruppen bare består av to deltakere er det naturlig at det blir høy arbeidsbelastning på hver enkel deltaker. Dette er noe gruppen har vært bevisst på hele tiden, men deltakerne føler seg komfortable med relativt høye arbeidsmengder. Samtidig mener gruppen at god planlegging vil resultere i en belastning som er mer behagelig.

På den andre siden vil det, som i de fleste prosjekter, likevel være en viss usikkerhet rundt estimater. Det er med andre ord høy sannsynlighet for at faktisk tids- og ressursbruk vil kunne overskride det som er estimert. Dersom dette viser seg å bli tilfelle vil prosjektgruppen være ekstra sårbare da det er et veldig begrenset antall deltakere å fordele eventuell ekstra arbeid på.

Manglende kompetanse/ for mye ukjent teknologi

Prosjektdeltakerne har et ønske om å benytte opparbeidet kompetanse gjennom studiet, men et mål med prosjektet er også å kunne ta i bruk nye teknologier. Som resultat av dette vil store deler av prosjektet kreve en god del selvlæring før implementasjonen kan starte. Gruppen mener at de nye teknologiene har et omfang som er akseptabelt i konteksten av prosjektet. Det er likevel en risiko for at omfanget blir større enn antatt da det er vanskelig å få et fullstendig bilde over hva som kreves av tidsressurser for å forstå ny teknologi.

Manglende dialog med oppdragsgiver

At oppdragsgiver har gitt prosjektgruppen relativt frie tøyler i utforming og implementasjon er i seg selv også en risiko. Det er ikke usannsynlig at prosjektdeltakernes visjon for prosjektet mister kontakt med den opprinnelige kravspesifikasjonen som oppdragsgiver har gitt. Dette kan spesielt utfolde seg dersom gruppen ikke har god og kontinuerlig dialog med oppdragsgiver.

3.5 Evalueringsmetode

Som spesifisert i 1.1 er det satt som mål å produsere et produkt som involverer spesialiseringene til samtlige prosjektdeltakere. Samtidig ønsker prosjektgruppen at produktet skal kunne løse oppdragsgivers problemstilling på en tilfredsstillende måte. Evalueringen av sluttresultatet vil derfor fokuseres rundt tre hovedpunkt:

1. Det skal utformes og implementeres en enkel og kjørbar applikasjon i klientteknologi
2. Det skal utformes og implementeres en kjørbar applikasjon i serverteknologi
3. Oppdragsgivers teknologi- og funksjonalitetsmessige krav skal overholdes i de to øvrige punktene

Måloppnåelsen for punkt 1 og 2 vil kontinuerlig overvåkes i kontekst av utviklingsmetoden Kanban. Applikasjonene vil deles i selvstendige enheter som skal implementeres, og deretter plasseres i Github Project Boards som selvstendige oppgaver. Dette vil gjøre prosjektgruppen i stand til å følge applikasjonenes livsløp fra kravspesifikasjon til ferdig implementert produkt. Når det gjelder punkt 3 vil dette i stor grad avhenge av tett dialog med og oppfølging fra oppdragsgiver. Et øvrig krav er at prosjektdeltakerne skal publisere kildekoden undervegs i en åpen versjonshåndteringstjeneste som oppdragsgiver har tilgang til. Dette fører til full åpenhet mellom gruppen og oppdragsgiver, og resulterer i at oppdragsgiver også kan følge utviklingen kontinuerlig. Videre har gruppen sett et behov for å utføre manuell testing undervegs, for å verifisere at generell funksjonalitet fungerer. Det er også rom for å utarbeide automatiserte tester der dette er hensiktsmessig og dersom tiden strekker til.

4 DETALJERT UTFORMING OG IMPLEMENTASJON

4.1 Arkitektur og prosjektstruktur

For prosjektet vårt spiller arkitektur og prosjektstruktur en nøkkelrolle for hvorvidt applikasjonen kan bli brukt på ønsket måte av oppdragsgiver. Som nevnt i punkt 3.1 er det viktig at prosjektet kan bli brukt som et utgangspunkt for fremtidige prosjekter. Da er det viktig at systemet er utviklet på en måte som fremmer gjenbrukbarhet, skalerbarhet, videreutvikling og som er lett å vedlikeholde. Det å bygge et system med mål om å ivareta de nevnte systemegenskapene kan være utfordrende, og det finnes veldig mange ulike tilnærminger for å oppnå dette.

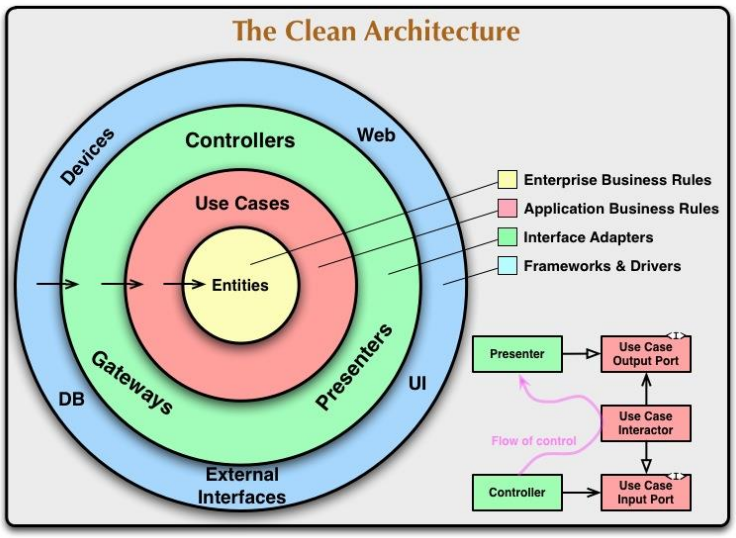
4.1.1 Clean Architecture

For å sikre at serverapplikasjonen blir bygget på en måte som oppfyller de ønskede egenskapene som beskrevet, har prosjektgruppen valgt å følge «Clean Architecture». Clean Architecture er en konseptuell arkitektur utviklet av Robert C. Martin, en amerikansk programvareingeniør og instruktør. Han er best kjent for å være en av forfatterne av «Agile Manifesto» og for å ha utviklet flere utformingsprinsipper innen programvareutvikling.

Robert C. Martin observerte at det fantes en håndfull velkjente arkitekturer som alle hadde samme mål: å separere programvareenheter inn i lag som har sitt eget ansvarsområde. Hver av de observerte arkitekturene hadde det samme målet, men med litt forskjellige fremgangsmåter. Han valgte å lage en arkitektur basert på dette som han døpte Clean Architecture. Målet med denne arkitekturen var at om de gitte retningslinjene ble fulgt, vil et system kunne få følgende kjennetegn (Martin, Clean Architecture, 2018, ss. 195-196):

- Uavhengig av rammeverk: Arkitekturen avhenger ikke av rammeverk, dette tillater utviklere å bruke rammeverk som verktøy, ikke noe som begrenser systemet.
- Testbart: Logikken til systemet kan testes uten grensesnitt, database, web-tjener eller andre eksterne elementer.
- Uavhengig av brukergrensesnitt: Grensesnittet kan lett byttes ut uten at det krever større endringer i resten av systemet.
- Uavhengig av database: Databasetjener kan byttes ut uten at det krever større endringer i resten av systemet. Eksempelvis fra en SQL Server til MongoDB.
- Uavhengig av eksterne tjenester

Clean Architecture blir ofte illustrert ved hjelp av en sirkelformet lagdelt figur som i Figur 4.1.



Figur 4.1 - The Clean Architecture. Fra *Clean Architecture (s.196),* Av Martin, R. C. (2002). Pearson Education, inc.

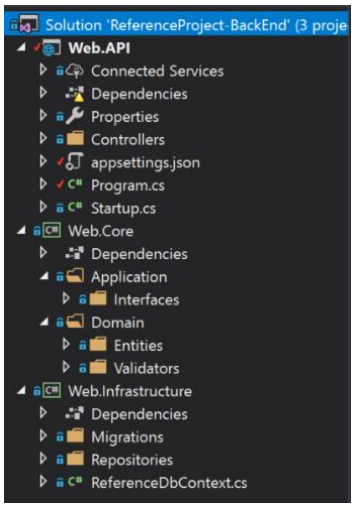
Hver av de konsentriske sirklene i Figur 4.1 representerer forskjellige områder av programvaren. Generelt sett er det slik at jo lengre inn mot kjernen i sirkelen jo høyere nivå vil programvaren være (Martin, Agile Software Development, Principles, Patterns, and Practices, 2002, s. 197). Det vil si at i kjernen av sirkelen så vil koden være mer abstrahert og påvirke systemet på makronivå. De ytterste lagene vil være mer spesifikke individuelle komponenter av systemet som påvirker systemet på mikronivå.

En av de grunnleggende prinsippene som ligger til grunn for denne arkitekturen er avhengighetsregelen, som Martin beskriver slik: «Kildekodeavhengigheter må bare peke innover, mot høyere nivå-retningslinjer». Mer konkret betyr dette at ingenting fra den ytre sirkelen kan bli nevnt i form av funksjoner, klasser, variabler eller andre programvareentiteter i den indre sirkelen. Dersom denne avhengighetsregelen brytes så vil avhengigheten peke fra kjernen og utover, som resulterer i at de ytre lagene ikke kan byttes ut med andre implementasjoner, som for eksempel databaseimplementasjonen.

4.1.2 Serverapplikasjonen sin implementasjon av Clean Architecture

Prosjektgruppen har valgt å utforme serverapplikasjonen i tett tilknytning til Clean Architecture og tilhørende prinsipper. Samtlige aspekter av idéen om Clean Architecture har derimot ikke blitt adoptert, men heller en variasjon. Prosjektgruppens implementasjon tar i bruk konseptet om en lagdelt arkitektur, hvor hvert lag har et veldefinert ansvarsområde for programmet. Samtidig har det blitt vektlagt å følge avhengighetsregelen som forklart i 4.1.1. Helt konkret så deles serverapplikasjonen i 3 prosjekter med hvert sitt ansvarsområde, som illustrert i Figur 4.2: Web.Core, Web.API og Web.Infrastructure.

Figur 4.2 - Illustrasjon av arkitektur til serverapplikasjon



Figur 4.3 - Prosjektstruktur sett i Microsoft Visual Studio

De tre prosjektene har hver sine følgende ansvarsområder:

Web.Api*:* Inngangspunktet til applikasjonen.

- `Controllers`*:* Ansvarlig for å ta imot og håndtere forespørsler mottatt fra klient i form av HTTP-forespørsler.
- `Startup.cs`*:* Ansvarlig for konfigurering av applikasjonen, og for å koble sammen implementasjonstyper til grensesnitt, som tillater avhengighetsinjeksjon å fungere under kjøretid av applikasjonen.

Web.Core: Kjernen av applikasjonen.

- `Interfaces`*:* Abstraksjoner for operasjoner som skal bli utført ved hjelp av Web.Infrastructure. I vårt tilfelle er dette grensesnitt for repositoriet og `DbContext`.
- `Entities`*:* Enkle klasser som representerer konsepter knyttet til domenet.
- `Business Logic`*:* Logikken i programmet som reflekterer domenet sine forretningsregler.

Web.Infrastructure: Implementasjon av datatilgang og migrering.

- `Repositories`: Implementerer grensesnittet fra Web.Core som definerer hvilke metoder applikasjonen krever knyttet til datatilgang.
- `DbContext`: Implementerer grensesnittet fra Web.Core som definerer broen mellom entiteter i Web.Core og databasen. Har direkte tilknytning til databasen og har hovedansvar for å hente ut data og persistere data til databasen.
- `Migrations`*:* Inneholder filer brukt av EF for å oppdatere databasen i henhold til datamodellen som er definert i Web.Core.

En av de viktigste aspektene ved den valgte implementasjonen er hvordan avhengigheter blir håndtert mellom de tre prosjektene. Som nevnt har

prosjektgruppen tatt i bruk avhengighetsprinsippet, hvor Web.API og Web.Infrastructure er direkte avhengige av Web.Core, mens Web.Core er en selvstendig modul uten avhengigheter til de ytre lagene. Dette resulterer i at databaseimplementasjon eller implementasjoner av grensesnitt lett kan byttes ut uten at dette krever at det blir gjort gjør større endringer i resten av systemet.

Denne utformingen tilrettelegger også for å lage enhetstester i alle lag. Logikken som ligger i kjernen har ingen avhengigheter og det vil derfor være rett frem for en utvikler å lage tester. I de ytre lagene refereres grensesnitt fra kjernen i stedet for konkrete implementasjoner. Dette åpner blant annet for bruk av objektetterligning som er nyttig når klasser og metoder med eksterne avhengigheter skal testes.

4.2 Web API

Et av hovedpunktene i oppdragsgivers kravspesifikasjon var å implementere et API som eksponeres mot internett. Dette skulle videre følge beste praksis for utforming som definert i den arkitektoniske utformingsmodellen REST (Representational State Transfer).

4.2.1 HTTP og ASP.NET Core

Innenfor konteksten til dette prosjektet ble HTTP (Hyper Text Transfer Protocol) benyttet for ende-til-ende kommunikasjon. Det finnes andre protokoller som kan oppnå samme formål, men HTTP er grunnlaget som internett bygger på (Berners-Lee, et al., 1999; Bradley, Jones, & Sakimura, 2015) og ble derfor det naturlige valget. HTTP følger en forespørsel-respons modell for kommunikasjon, der en klient sender en forespørsel hvorpå en tjener svarer med en respons. Et API vil typisk eksponere ett eller flere endepunkter til offentligheten. Disse endepunktene representeres gjennom å implementere en eller flere av forespørselsmetodene som definert i HTTP. For å fange opp og respondere på forespørsler fra en klient trengs det egne mekanismer. ASP.NET Core er et rammeverk for webutvikling og tilbyr blant annet alle nødvendige verktøyer for implementasjon av HTTP baserte web-APIer.

Prosjektgruppen startet med å implementere en generisk kontrollerklasse som har som formål å ta i mot og respondere på forespørsler. Videre valgte gruppen å implementere følgende responsmetoder: `GET`, `POST`, `PUT` og `DELETE`. HTTP spesifiserer flere metoder enn dette, men disse er ofte de mest brukte spesielt når web-APIet skal kommunisere med et datalager. I forbindelse med datalagre er det vanlig at følgende handlinger tilbys: lesing, oppretting, oppdatering og sletting av data. Disse handlingene kan representeres med de nevnte responsmetodene på følgende vis:

- `GET`: hent data fra en ressurs
- `POST`: opprett en dataressurs
- `PUT`: oppdater en dataressurs
- `DELETE`: slett en dataressurs

Ved å implementere nettopp disse responsmetodene åpnes det for at eventuelle klientapplikasjoner som konsumerer API-et kan utføre standardhandlinger for en typisk CRUD-basert applikasjon. Som nevnt var en ønskelig egenskap ved referanseprosjektet at det skal kunne tas i bruk raskt. Prosjektgruppen valgte derfor å begrense seg til disse responsmetodene, for å unngå i introdusere unødvendig kompleksitet til prosjektet samt å unngå å legge for strenge føringer på utforming.

WebAPI-et har så langt blitt implementert på grunnlag av enkle statiske responsdata. Dersom det er et ønske om å kunne behandle dynamiske objekter som representerer faktiske ressurser, kan det for eksempel utarbeides en domenemodell. Prosjektgruppen valgte å utarbeide en enkel domenemodell og tilhørende entiteter med det formål å kunne tilby dynamiske data. Dette vil i sin tur gjøre at serverapplikasjonen i større grad ligner en reell applikasjon. Gruppen valgte i første omgang å refaktorere den generiske kontrollerklassen til en konkret kontrollerklasse, `StudentsController`, som representere ressursen Students. WebAPI-et består på dette tidspunkt av én ressurs med fire endepunkter representert ved HTTP responsmetodene.

Alle ressurser som eksponeres mot internett må kunne identifiseres unikt gjennom en URI. URI-en til ressursen som kontrollerklassen representerer, spesifiseres ved hjelp av `Route`-attributtet på kontrolleren, med parameter `api/[controller]`. Alle forespørsel som heretter sendes til URI-en `http://<domene>/api/Students` vil bli fanget opp at `StudentsController`-klassen. De ulike endepunktene spesifiseres med henholdsvis `HttpGet`, `HttpPost`, `HttpPut` og `HttpDelete`-attributtene. `StudentsController`-klassen ender opp med å bli seende ut som i Figur 4.4.

```
[Route("api/[controller]")]
[ApiController]
public class StudentsController : ControllerBase
{
    public StudentsController(…) {…}

    [HttpGet]
    public ActionResult<…> GetStudents() {…}

    [HttpGet]
    public ActionResult<…> GetStudent(…) {…}

    [HttpPost]
    public ActionResult CreateStudent(…) {…}

    [HttpPut]
    public ActionResult UpdateStudent(…) {…}

    [HttpDelete]
    public ActionResult DeleteStudent(…) {…}
}
```

Figur 4.4 - StudentsController-klassen

4.2.2 Representational State Transfer (REST)

REST er en arkitektonisk modell for utforming og implementasjon av internett som helhet og tilhørende tjenester og systemer (Fielding, 2000, ss. 76-105, 116-134). Modellen ble utviklet parallelt med HTTP men har ingen direkte sammenheng med hverken HTTP protokollen eller webtjenester og web API-er generelt. REST definerer helt enkelt et sett med arkitektoniske avgrensninger som kan benyttes der det er ønskelig, som for eksempel i utformingen av et webAPI. En tjeneste eller et system som anvender REST-prinsippene blir ofte omtalt som RESTful.

Fielding viser til flere fordeler ved adapsjon av REST som overordnet utformingsmodell, blant annet skalerbarhet, generaliserte grensesnitt, uavhengig utrulling av komponenter og separasjon av klient- og serverapplikasjoner. Dette kan oppsummeres som at applikasjoner og systemer som forholder seg til REST-prinsippene kan kommunisere og interagere med hverandre på tvers av underliggende faktorer som programmeringsspråk og datarepresentasjoner. Å implementere samtlige arkitektoniske retningslinjer som Fielding presenterer ville vært utenfor omfanget til denne oppgaven. Samtidig er det flere misoppfatninger og feiltolkninger av Fieldings opprinnelige spesifikasjon (Avram, 2016). For å unngå å gå i denne fellen har det blitt lagt fokus på det prosjektgruppen mener er et absolutt minimum for å kunne tilnærme seg et RESTful webAPI på sikt. Etterhvert som applikasjonen utvides er det derimot høyst sannsynlig at implementasjonen må revideres for å forsikre at denne er i samsvar med retningslinjene.

Et av de arkitektoniske kravene går ut på separasjon mellom klient og server, i en klient-server arkitektur. Fielding beskriver rollen til en klient som et brukergrensesnitt som kan interagere med en server, hvorpå serveren har ansvar for å lagre og prosessere dataressurser. Dette kravet blir overholdt ved at der er en fysisk separasjon mellom eventuelle klienter og serverapplikasjonen gjennom et webAPI. Videre setter Fielding krav til at serveren ikke kan lagre noen form for sesjonsdata om en klient. All kommunikasjon skal foregå tilstandsløst, noe serverapplikasjonen også overholder. Dette betyr at en eventuell klient må inkludere fullstendig informasjon som er nødvendig for å utføre en operasjon i forespørselen den sender.

Videre har gruppen fokusert på Fieldings retningslinje om et uniformt grensesnitt. Dette kan ansees som selve grunnmuren for REST, og får implikasjoner for de øvrige retningslinjene. Denne retningslinjen har videre fire krav for å oppfylles. Det første kravet spesifiserer at alle dataressurser skal identifiseres med en statisk identifikator som ikke endres. I serverapplikasjonen er det i hovedsak én ressurs som er eksponert, nemlig `Student`-ressursen. Prosjektgruppens implementasjon av webAPI-et gir hver instans en unik identifikator på formen `http://<domene>/api/Students/{studentId}`. På denne måten kan enhver instans av denne ressursen interageres med ved å spesifisere dens ID. IDen genereres per dags dato av databasen hvor ressursen blir persistert.

De enkelte ressursene skal videre kun manipuleres gjennom representasjoner av ressursen. Eksempelvis dersom en klient ønsker å endre en attributt til et `student`-objekt må det gis en fullstendig representasjon av objektet med det nye attributtet. Dette løses i serverapplikasjonen gjennom henholdsvis ASP.NET Core sin «model binding»-mekanisme samt prosjektgruppens egen valideringsimplementasjon. Som standard for representasjon av ressurser i klient-server kommunikasjonen har gruppen valgt JSON. Dette er en universell måte å representere dataressurser i tekstformat. Tekstformatet er nyttig i web-sammenheng siden det er kompakt og bidrar til å redusere den totale størrelsen på HTTP forespørsel og responser.

Det tredje kravet går ut på at meldinger skal være selvbeskrivende. En server eller klient skal kunne hente ut all informasjon som trengs for å foreta en operasjon, basert på innholdet til meldingen. Ved å kombinere HTTP headeren `Content-Type` og HTTP statuskoder, kan det forsikres at mottakeren av en melding klarer å vurdere riktig handling basert på meldingen alene. Eksempelvis dersom en server sender en respons til en klient med statuskode `200 OK` og `Content-Type: application/json`, vet klienten at forespørselen ble godkjent og at nyttelasten i responsen kan analyseres som en JSON-representasjon. I prosjektgruppens implementasjon spesifiseres nettopp disse attributtene for samtlige mulige responser som webAPI-et kan generere.

For at klienter skal kunne oppdage og navigere seg gjennom REST-baserte ressurser, trengs det en siste mekanisme. Fielding beskriver det siste kravet som «Hypermedia as the Engine of Application State», eller HATEOAS. Enkelt forklart betyr dette at en server skal bare produsere hypermediabaserte responser. Dette tillater en klient å ut fra serverresponsen bestemme mulige handlinger, samt navigere dypere i et eventuelt ressurshierarki. Dersom en server har ressurser som videre har avhengigheter til andre ressurser kan dette representeres ved hjelp av hypermedia. For eksempel dersom en klient etterspør en studentressurs som har en avhengighet til en skoleressurs, skal responsnyttelasten inkludere en URI til denne avledede ressursen. Klienten har da en naturlig tilgang videre til skoleressursen og tilhørende operasjoner. I serverapplikasjonen er det derimot ingen avhengigheter, så dette kravet er også oppfylt.

4.2.3 Autentisering og autorisering

Dersom en applikasjon eller deler av den skal kunne beskyttes mot uvedkommende, må det implementeres egne sikkerhetsmekanismer for dette. Alle deler av en applikasjon som brukere eller andre applikasjoner kan interagere med, er kandidater for å bli utnyttet av uvedkommende. I konteksten av serverapplikasjonen vil webAPI-et være et slikt interaksjonspunkt. Autentisering og autorisering er grunnleggende sikkerhetsmekanismer som kan implementeres i en applikasjon og kan bidra til sikre blant annet webAPI-er. Begrepene har følgende betydninger:

- Autentisering: Å verifisere identiteten til en bruker/klient

- Autorisering: Å verifisere tilgangsrettighet til den spesifikke ressursen som blir forespurt av en bruker/klient

Det finnes mange forskjellige sikkerhetsprotokoller og -standarder som oppfyller disse kravene. Oppdragsgiver har uttrykt ønske om å implementere disse kravene ved hjelp av identitetsplattformen Auth0 og JSON Web Token (JWT) som øvrig mekanisme for utveksling av sensitiv informasjon. Den kanskje mest åpenbare fordelen ved å velge en identitetsplattform er sikkerhetsaspektet. Det å implementere egne sikkerhetsprotokoller medfører høy risiko (OWASP, 2017), og ofte er det bedre å overlate dette til eksperter.

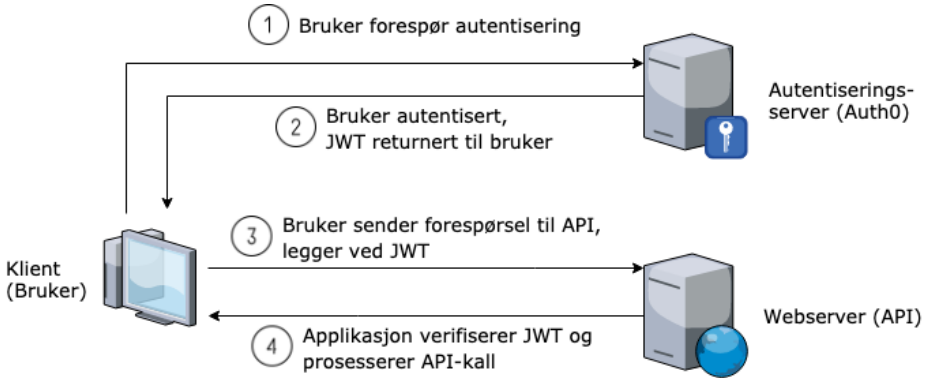
JWT er en standard for oppretting av tilgangstokener som inneholder et sett med påstander som en entitet kan inneha (Bradley, Jones, & Sakimura, 2015). Et eksempel på en slik påstand kan være «entitet X innehar rolle Y som gir tilgang til ressurs Z». Ved hjelp av asymmetrisk kryptering (Datatilsynet, 2017) kan en JWT signeres og verifiseres av andre aktører. En JWT består av tre separate deler som hver blir omsatt til en kodet tekstrepresentasjon:

- Header: inneholder tokentype og algoritmen som ble brukt for å generere tokenet sin digitale signatur
- Payload: informasjon om entiteten som JWT-en tilhører og dens rettigheter
- Signature: digital signatur generert ved å anvende signaturalgoritmen på `header + payload + privatnøkkel`

Auth0 er som beskrevet en identitetsplattform som tilbyr blant annet autentisering og autorisering gjennom bruk av JWT. Auth0 kan integreres med alle applikasjoner som har støtte for eller implementerer JWT-standarden. Ved bruk av Auth0 kan klient- og serverapplikasjoner utvikles isolert fra hverandre, og på et senere tidspunkt kan sømløs autentisering og autorisering konfigureres mellom disse. Auth0 fungerer her som en mellommann; klienter kan hente JWTer fra Auth0 og videre bruke disse til å for eksempel få tilgang til en serverapplikasjon sitt API. En slik flyt er enkelt beskrevet i Figur 4.5. Auth0 tilbyr en administratorkonsoll hvor utviklere kan opprette identitetsinstanser for sine respektive server- og klientapplikasjoner. Her kan det også registreres brukere og tilganger og rettigheter som disse brukerne har kan håndteres.

ASP.NET Core 2.2 har støtte for JWT-basert autentisering og autorisering. I Figur 4.6 er serverapplikasjonen konfigurert til å integrere med Auth0 som identitetsplattform. De første tre linjene forteller applikasjonen at det ønskes tilgang til JWT som autentiserings- og autoriseringsmekanisme. De resterende linjene integrerer med en spesifikk Auth0-instans som er konfigurert for prosjektgruppens serverapplikasjon direkte i Auth0 sin administratorkonsoll. `options.Authority` er URL-en til selve Auth0-instansen, mens `options.Audience` spesifiserer at JWTen bare kan brukes i den spesifikke applikasjonen og den spesifikke ressursen som spesifisert i URL-en.

Ved hjelp av ASP.NET Core sin innebygde attributt `Authorize` kan det videre spesifiseres hvilke av endepunktene i API-et som krever autentisering. I tillegg kan attributtet gis parametere som avgrenser tilgangen ytterligere basert på et sett med rettigheter. I serverapplikasjonen har det blitt valgt en rollebasert autoriseringsmodell, da Auth0 støtter tildeling av roller til brukere. Figur 4.7 viser `GetStudents`-endepunktet med autentisering og rollebasert autorisering aktivert hvor rolle som «admin» er påkrevd.



Figur 4.5 - Auth0 autentisering-/autoriseringssflyt

```
services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(options =>
{
    options.Authority = "https://dev-yx7nujom.eu.auth0.com";
    options.Audience = "http://localhost:5000/api/Students";
});
```

Figur 4.6 - Konfigurasjon av autentisering og autorisering

```
[Authorize(Roles = "admin")]
[HttpGet]
public ActionResult<…> GetStudents() {…}
```

Figur 4.7 - GetStudents-endepunkt med autentisering og autorisering aktivert

4.2.4 Validering

Enhver applikasjon som har mulighet til å håndtere input, enten fra en annen applikasjon eller en bruker, bør ha en form for valideringsmekanisme av inputdata. Formålet med en slik validering er å unngå interne feil som følger av at applikasjonen prøver å prosessere invalide data. WebAPI-et som prosjektgruppen har utformet lar andre applikasjoner utføre operasjoner på et sett med ressurser, som for eksempel å opprette en ny ressurs. Som beskrevet i 4.2.2 må en klientapplikasjon spesifisere en URI og fullstendige data i nyttelasten til forespørselen for å kunne opprette en ny ressurs. Serverapplikasjonen vår vil hente ut og prosessere disse nyttelastdataene, og de må derfor valideres.

ASP.NET Core forenkler arbeidet med nyttelastdata ved å automatisk binde disse dataene til en objektrepresentasjon som kan jobbes med programmatisk

(Microsoft, 2018). I tillegg er det støtte for datavalidering gjennom bruk av dataannotasjoner direkte i modellklassene (Microsoft, 2019). Ulempene med dette er at disse klassene ofte får unødvendig mange slike annotasjoner, noe som reduserer lesbarheten til kildekoden. Det kan også være vanskelig å kombinere slike dataannotasjoner for å lage mer komplekse valideringsregler. Dersom applikasjonen skal følge prinsipper for god utforming, vil dette også være i strid med prinsippet om å adskille kode som har ulike formål (Microsoft, 2019).

Basert på disse vurderingene valgte prosjektgruppen å bruke det eksterne valideringsbiblioteket FluentValidation til dette formålet. Det opprettes en egen valideringsklasse for hver modell som det skal utføres valideringer på. I Figur 4.8 er det opprettet en valideringsklasse for `Student`-objekter. Syntaksen er noe annerledes enn ved dataannotasjoner, men fortsatt lesbar og det er tydelig hva koden utfører. Valideringsreglene gruppen har valgt å implementere var ikke et krav i seg selv, så reglene er derfor av enkel art. En annen fordel med biblioteket er direkte kompatibilitet med ASP.NET Core. Blant annet blir det automatisk generert utfyllende feilmeldinger i serverresponser på ugyldige klientforespørsler. Figur 4.9 viser en feilmelding hvor feltet `FirstName` manglet i en klientforespørsel sendt til `POST`-endepunktet i API-et.

```
public class StudentValidator : AbstractValidator<Student>
{
    public StudentValidator()
    {
        RuleFor(student => student.StudentId).NotNull();
        RuleFor(student => student.FirstName).NotNull();
        RuleFor(student => student.FirstName).MinimumLength(2);
        RuleFor(student => student.LastName).NotNull();
        RuleFor(student => student.LastName).MinimumLength(2);
    }
}
```

Figur 4.8 - Validering med FluentValidation

```
{
    "errors": {
        "FirstName": ["'First Name' must not be empty."]
    },
    "title": "One or more validation errors occurred.",
    "status": 400,
    "traceId": "|e58e3f4a281ed646af716a18415e0923.29bdbd9b_"
}
```

Figur 4.9 - Feilmelding automatisk generert av FluentValidation

4.3 Databasehåndtering og -vedlikehold

«*The organizational structure of data, the data model, is architecturally significant. The technologies and systems that move data on and off a rotating magnetic surface are not. Relational database systems that force data to be organized into tables and accessed with SQL have much more to do with the latter than the former. The data is significant. The Database is a detail*.» (Martin, Clean Architecture, 2018, s. 260).

Som nevnt i kravspesifikasjonen er det viktig for oppdragsgiver at referanseprosjektet skal være gjenbrukbart. Nye prosjekter kan være veldig ulike i både i omfang og domenet som modelleres. Det betyr i praksis at applikasjonen må være tilpasningsdyktig og åpent for endringer av implementasjoner i alle lag, inkludert databaseleverandør. Kravet til implementasjon av databasehåndtering var derfor at det ikke kan være tett knyttet til en spesifikk databaseleverandør, men heller at det skal være lett å gjøre endringer. Som Robert C. Martin beskriver skal databasen bare være en detalj mens dataene skal være i fokus. For å oppnå dette valgte prosjektgruppen å ta i bruk rammeverket Entity Framework.

4.3.1 Entity Framework

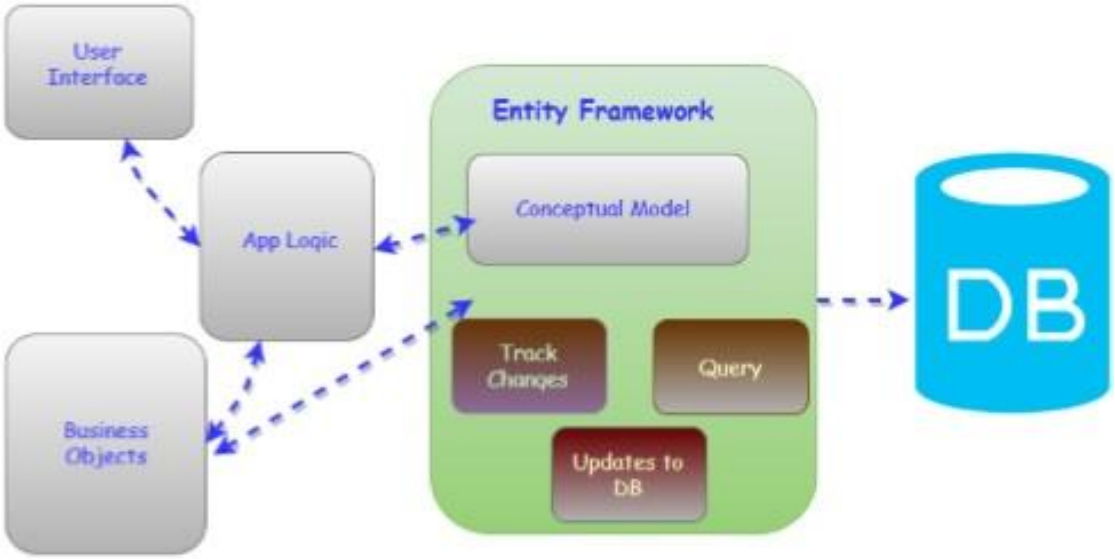
I 3.3.3 forklares EF fra et overordnet perspektiv, men en del mer tekniske detaljer rundt EF og hvordan prosjektgruppen har tatt det i bruk for å oppnå ønsket abstraksjon for henting og persistering av data, må forklares. EF er en ORM som er en type verktøy som forenkler kartlegging mellom objekter i programvaren til tabeller og kolonner i en relasjonsdatabase. Rammeverket tar hånd om oppretting av databaseoppkobling og eksekvering av kommandoer, i tillegg til å håndtere resultater av spørringer og automatisk omforme disse resultatene til objekter i applikasjonen. Rammeverket kontrollerer også endringer som er gjort på objekter og vil persistere disse endringene til databasen.

EF vil generere nødvendige databasekommandoer for lese- og skriveoperasjoner og eksekvere disse kommandoene automatisk. (Tutorialspoint, 2019). I Figur 4.10 blir samhandlingen mellom applikasjonen, EF og databasen illustrert på et konseptuelt nivå. Rammeverket har et bredt spekter av funksjonalitet og er åpent for å konfigureres slik at det dekker ulike applikasjoner sine spesielle behov. Prosjektgruppen har ikke utforsket alle muligheter knyttet til EF, men heller implementert det som regnes som kjernen av rammeverket. Når rammeverket tas i bruk er det hovedsakelig én ting som alltid skal inkluderes: modellen. En modell i EF-sammenheng består av entitetklasser og et `Context`-objekt som representerer en sesjon med databasen, som tillater applikasjonen å utføre spørringer og persistering av data (Microsoft, 2016).

Det finnes flere fremgangsmåter for å ta i bruk EF, men dette beskrives ikke i detalj her. I stedet gis en kort beskrivelse av de vanligste fremgangsmetodene:

- Code First*:* Modellen defineres ved hjelp av C# klasser. Rammeverket vil inspisere disse klassene, og bruke et sett med regler og konvensjoner for å bestemme hvordan disse klassene og relasjonene beskriver modellen, og hvordan modellen bør representeres i databasen.
- Model First*:* Modellen defineres grafisk ved hjelp av Entity Framework Designer, som deretter vil generere og eksekvere SQL for å lage et databaseskjema som reflekterer modellen.

- Database First: Et alternativ til «Code First» og «Model First» hvor entiteter, `properties` og `context` blir generert basert på et eksisterende databaseskjema.



Figur 4.10 - Konseptuell modell for Entity Framework fra https://www.tutorialspoint.com/entity_framework/entity_framework_overview.html

Av de alternative fremgangsmåtene har prosjektgruppen valgt Code First. Valget er basert på ønsket om å skifte fokuset bort fra databasen, i tillegg til at prosjektdeltakerne ikke ønsker å bruke tidsressurser på å sette seg inn i et grafisk verktøy. Denne tilnærmingen tilrettelegger for at utviklere kan gjøre det de kan best, å jobbe med programkode. På tross av at beskrivelsen av EF kan gi inntrykk av at det er relativt komplekst, så er en av fordelene med rammeverket at det er veldig lett å komme i gang med. Videre demonstreres hvordan gruppen har valgt å ta i bruk EF. I Figur 4.11 blir modellen definert ved hjelp av `DbContext`, hvor klassen har en property `DbSet` av typen `Student`. En `DbContext` har én eller flere slike `DbSet`, hvor hvert `DbSet` kan bli sett på som én tabell i databasen. Som tidligere nevnt vil EF håndtere generering database basert på entitetene som blir brukt i disse settene.

I Figur 4.12 illustreres et godt eksempel på hvordan SQL spørringer blir abstrahert vekk for utvikleren. Her initialiseres en ny instans av `ReferenceDbContext` som er vist i Figur 4.11 for å åpne en kobling mot databasen. Videre kan dataene som ligger i databasen behandles som en samling av objekter for å hente ut listen av studenter fra databasen. I bakgrunnen vil EF generere spørringen automatisk, slik at det ikke trengs å formuleres egne spørringer for å hente ut data fra databasen. På denne måten slipper utviklere også å tenke på hvordan dataen fra databasen skal konverteres til datastrukturer i applikasjonen.

Ettersom prosjektgruppen har et enkelt domene å forholde seg til så kommer kanskje ikke kraften av EF så godt frem i akkurat dette tilfellet. Når oppdragsgiver skal ta i bruk prosjektet så vil domenet og relasjonene mellom de forskjellige

entitetene være av mer kompleks natur enn her, og effekten av EF vil være av høyere betydning.

```
public class ReferenceDbContext : DbContext, IReferenceDbContext
{
    public ReferenceDbContext(DbContextOptions<ReferenceDbContext> options)
        : base(options)
        {…}

    public DbSet<Student> Students { get; set; }
}
```

Figur 4.11 - DbContext klassen

```
public IList<Student> GetStudents()
{
    var options = new DbContextOptionsBuilder<ReferenceDbContext>().
        UseSqlServer(_connectionString);

    using (var context = new ReferenceDbContext(options.Options))
    {
        return context.Students.ToList();
    }
}
```

Figur 4.12 - Bruk av DbContext i Repository

4.4 Enhetstesting

Programvaretesting er et viktig verktøy i arbeidet med å oppdage, korrigere og forhindre feil i programkode. Et spesielt viktig aspekt dersom det er fokus på å produsere applikasjoner av høy kvalitet og med god utforming, er enhetstesting (Microsoft, 2018). En enhetstest tester en isolert del av programkoden med det formål å verifisere at eksekveringen av koden gitt et forhåndsdefinert sett med forutsetninger, medfører et gitt forventet resultat. For å kunne gi forutsigbare resultater er det vesentlig at den delen av koden som testes ikke har eksterne avhengigheter. Eksterne avhengigheter er av natur ikke-statiske, noe som gjør det praktisk umulig å få forutsigbare resultater. Formålet med en enhetstest er som sagt å teste en del av programkoden i isolasjon, noe som ikke lar seg gjøre dersom utenforliggende faktorer som ikke kan styres påvirker resultatet. I noen tilfeller vil det likevel eksistere slike avhengigheter. Disse må i så tilfelle substitueres ut med en etterligning av avhengigheten (Microsoft, 2018). Denne etterligningen instansieres som en statisk avhengighet, og vil med andre ord aldri endre sin tilstand fra det som blir spesifisert. På denne måten forsikres det at avhengigheten er forutsigbar.

Microsoft viser til flere fordeler ved bruk av enhetstester. I konteksten av prosjektgruppens oppgave er det to fordeler som er spesielt verdifulle:

- Det blir lavere grad av kobling mellom programenheter (ISO/IEC/IEEE, 2010)

- Ved introduksjon av ny funksjonalitet kan eventuelle feil som dette medfører i allerede eksiterende funksjonalitet, også kalt regresjonsfeil, oppdages tidligere

Å minimere graden av kobling vil si at programvaremoduler gjøres mindre avhengige av hverandre. Dette vil i sin tur gjøre det lettere å gjenbruke kildekoden i prosjektet og tilhørende moduler, noe som henger tett sammen med et av oppdragsgivers hovedmål. Tidlig oppdagelse av regresjonsfeil er også verdifullt da dette direkte påvirker tidsbruk og ressurser som medgår til feilsøking. Som beskrevet tidligere i oppgaven er oppdragsgiver også opptatt av å redusere tidsbruk ved oppstart av nye prosjekter. For å få utbytte av enhetstesting og fordelene dette medfører er det derfor viktig at dette gjøres konsekvent og fra et tidlig stadium.

Oppdragsgiver gav anbefalinger om å ta i bruk et automatisert testrammeverk som håndterer instansiering og kjøring av tester. Dette lar prosjektdeltakerne fokusere på gode testrutiner og høy grad av testdekning fremfor selve testkonfigureringen. xUnit, et testrammeverk for C#/.NET Core, ble valgt til dette formålet. Rammeverket brukes blant annet internt hos Microsoft (Microsoft, 2017) og dekker gruppens øvrige behov i forbindelse med testing. Videre ble det også besluttet å bruke et C#-bibliotek kalt NSubstitute som gjør det lettere å jobbe med etterligning av eksterne avhengigheter. Basert på egne erfaringer har prosjektdeltakerne også besluttet at terskelen er lavere for å opprettholde en god testrutine dersom det eksisterer et standardoppsett for testmetoder. Deltakerne ønsket derfor å ta i bruk en standard for navngiving av testmetoder samt oppsett av selve testene. Microsoft har utarbeidet et sett med retningslinjer for nettopp dette:

- Navngiving av testmetodene bør være tredelt hvor følgende detaljer beskrives: navnet på metoden/programkodeenheten som testes, tilstanden under testscenariet og forventet resultat når scenariet er utført. Eksempelvis `Addisjon_PositiveLedd_GirPositivSum`.
- Oppsettet av testmetodene bør følge en modell bestående av tre steg:
  I. Ordne: Oppsett av objekter som skal vurderes
  II. Handle: Utfør nødvendige handlinger på objektene
  III. Vurdere: Vurder om utfallet er som forventet

Prosjektgruppen har fokusert på å implementere enhetstester for webAPI-et, da dette er en av modulene i prosjektet hvor det er mest programlogikk. I Figur 4.13 har det blitt implementert en fullstendig enhetstest for `GetStudent`-endepunktet fra `StudentsController`-klassen. Enhetstesten følger de overnevnte retningslinjene. Her illustreres også hvordan etterligning av eksterne avhengigheter kan håndteres. `StudentsController`-klassen har behov for å kalle `GetStudents`-metoden på et objekt av typen `IStudentsRepository`. En etterligning opprettes ved hjelp av `Substitute.For<IStudentsRepository>`-metoden fra NSubstitute. Videre kan returverdien til `GetStudents`-metoden spesifiseres på denne etterligningen ved hjelp av `studentsRepositorySubstitute.GetStudents().Returns(…)`. På denne

måten kontrolleres tilstanden til denne avhengigheten fullstendig under testscenariet.

```
[Fact]
public void GetStudents_NonEmptyRepository_ReturnsStatusCode200AndNonEmptyCollection()
{
    // Arrange
    var studentsRepositorySubstitute = Substitute.For<IStudentsRepository>();
    var student = new Student { StudentId = 1, FirstName = "Ola",
        LastName = "Nordmann" };
    var studentsList = new List<Student> { student };
    var studentsController = new StudentsController(studentsRepositorySubstitute);
    studentsRepositorySubstitute.GetStudents().Returns(studentsList);

    // Act
    var response = studentsController.GetStudents().Result as OkObjectResult;
    var statusCode = response.StatusCode;
    var studentsCollection = (System.Collections.IEnumerable)response.Value;

    // Assert
    Assert.NotEmpty(studentsCollection);
    Assert.Equal((int)HttpStatusCode.OK, statusCode);
}
```

Figur 4.13 - Enhetstest for GetStudents-endepunkt

4.5 Pakking og utrulling

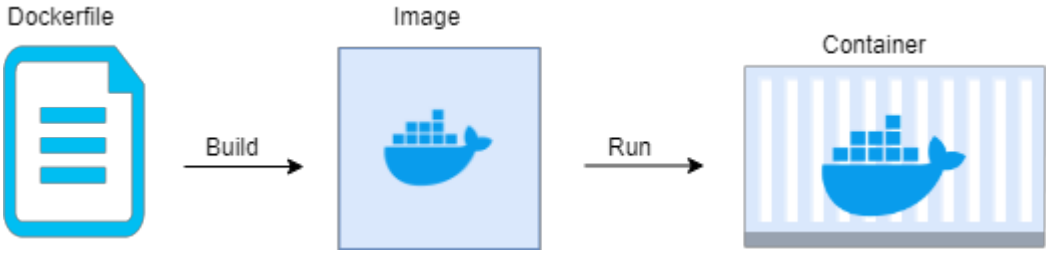
4.5.1 Kontainerisering

Dagens standard i IT-bransjen er at kunden forventer hyppige leveranser og forbedringer. Tiden brukt fra utvikling til utrulling blir mindre og mindre, samtidig som kunder forventer at kvaliteten på programvaren skal være god. For å holde følge med denne trenden forventes det at både utviklere og systemadministratorer skal følge en smidig arbeidsflyt.

For at systemadministratorer skal kunne utføre hyppige utrullinger av programvare, må det kunne forsikres at programvaren som blir rullet ut vil oppføre seg som forventet på miljøet den blir rullet ut til. For å sikre at applikasjonen som blir rullet ut fungerer uavhengig av miljø, har prosjektgruppen valgt å bruke kontaineriseringsteknologi. Kontainerisering er i korte trekk å samle en applikasjon, alle dens avhengigheter og biblioteker inn i en kontainer med dens eget operativsystem. Kontainerisering sammenlignes ofte med virtuelle maskiner (VM), som er et lignende konsept. Det er derimot flere forskjeller mellom kontainerisering og VM, hvor hovedforskjellen er at flere kontainere deler samme operativsystemkjerne som vertsmaskinen, mens et VM krever en separat instans av operativsystem. Dette betyr at skalering av kontaineriserte applikasjoner krever mindre ressurser sammenlignet med ved bruk av virtuelle maskiner.

Som beskrevet i 3.3.7 ble Docker valgt som verktøy for kontainerisering. Docker er det ledende alternativet for kontainerhåndtering i bransjen (Sysdig, 2018), og dokumentasjonen til verktøyet er veldig god. Det er en tre-stegs prosess for å lage kontainere ved hjelp av Docker. Første utarbeides en «Dockerfile», en fil som beskriver kontaineren. Filen inneholder en liste med instruksjoner som «Docker

Engine» kjører for å lage et «Docker-image» av kontaineren. Når en slik Dockerfile bygges vil Docker Engine utføre disse instruksjonene for å konstruere et Docker-image. Et Docker-image er en avbildning som består av flere lag. Denne avbildningen er en komplett og eksekverbar versjon av applikasjonen som inkluderer systembiblioteker, verktøy og andre avhengigheter for den kjørbare koden. Denne avbildningen trenger kun kjernen til vertsoperativsystemet for å kunne kjøre. Ved hjelp av Docker Engine kan et Docker-image kjøres som en Docker-container. En Docker-container representerer med andre ord en instans av et Docker-image. Det kan opprettes én eller flere instanser av samme Docker-image, og flere kontainere kan kjøre på samme vertsoperativsystem.



Figur 4.14 tre-stegs prosess for kontainerisering

4.5.2 Orkestrering

En kontainer kan være avhengig av flere eksterne tjenester og applikasjoner. For prosjektgruppens del vil klientapplikasjonen være avhengig av at serverapplikasjonen kjører, samtidig som serverapplikasjonen er avhengig av at databasetjeneren kjører. For å håndtere kjøring av disse tjenestene som avhengigheter har gruppen valgt å ta i bruk «Docker Compose» som et orkestreringsverktøy. Docker Compose fungerer ved at alle tjenester som kreves av en gitt applikasjon defineres i en YAML-fil, slik at de kan kjøres sammen i et isolert miljø. Deretter brukes en enkel kommando for å instansiere og starte opp alle tjenester basert på konfigurasjonen (Docker inc., 2019).

4.6 Klientapplikasjonen

Prosjektgruppens opprinnelige tolkning av oppgaveformuleringen ga rom for at prosjektet kunne utvides med en klientapplikasjon. Dette var delvis basert på gruppen vurdering av hvordan omfanget av prosjektet ville bli i sin ordinære form. Samtidig hadde gruppen et ønske om å trekke inn mer klientteknologi, for å kunne dekke prosjektdeltakernes spesialiseringer i større grad. Gjennom utviklingsperioden har derimot prioriteten hele tiden vært på serverapplikasjonen, da dette var oppdragsgivers opprinnelige ønske. Bare i perioder hvor det har vært overskudd av tid, har en klientapplikasjon blitt tildelt utviklingsressurser.

Totalt sett har dette resultert i at klientapplikasjonen er av en mer enkel art enn det som opprinnelig ble lagt til grunn i 3.2.2. Prosjektgruppen valgte å fokusere på følgende forenklede sett av krav for klientapplikasjonen: utforming av en enkel prosjektstruktur, integrasjon med Auth0 og implementasjon av tilgang til

serverapplikasjonen sitt webAPI, og konfigurasjon av pakking og utrulling av applikasjonen.

4.6.1 Prosjektstruktur

Tradisjonelle webapplikasjoner sammenlignet med separert klientapplikasjon

I tradisjonelle modeller for webapplikasjoner er det ofte en direkte kobling mellom en klient som viser et brukergrensesnitt og serveren som har produsert dette grensesnittet. Generelt sett vil serveren ha fullstendig ansvar for samtlige oppgaver for applikasjonen. Så godt som all logikk for brukergrensesnittet vil med andre ord bli utført på serversiden. Klientens ansvar vil kun omhandle prosessering av servergenererte dokumenter og utveksling av forespørsler. Ved hver forespørsel og tilhørende respons fra serveren, vil klienten måtte oppdatere hele siden. I et slikt tilfelle vil arkitektoniske valg relatert til brukergrensesnittet og datavisning være en naturlig del av den øvrige arkitekturen for resten av webapplikasjonen.

I mange tilfeller er det derimot et ønske om utvidet funksjonalitet i klienten og kanskje også en separat klientapplikasjon. All logikk for brukergrensesnittet vil da bli flyttet til klienten, og kommunikasjon med serveren vil i stor grad være i forbindelse med datautveksling. En klientapplikasjon som implementerer denne typen modell blir ofte kalt en Single Page Application (SPA) (Microsoft, 2019). Ved hjelp av nyere teknologier som AJAX kan i tillegg enkelte deler av brukergrensesnittet i klienten oppdateres uten å gjennomføre en komplett sideoppdatering. I følge Microsoft kan dette blant annet resultere i en bedre opplevelse for sluttbrukeren, da brukeren slipper å forholde seg kontinuerlige sideoppdateringer.

Arkitektur og prosjektstruktur i konteksten av React

Resultatet av å flytte brukergrensesnittlogikk over i en klientapplikasjon er naturlig nok økt kompleksitet i klienten. For å kunne håndtere dette på en god måte er det også i klientapplikasjonen viktig å ha fokus på god arkitektur og prosjektstruktur. Prosjektgruppen valgte å benytte React som hovedverktøy for utvikling av klientapplikasjonen, og prosjektstrukturen vil derfor preges av dette. React legger til rette for at brukergrensesnittet, det vil si klientapplikasjonen, separeres i komponenter som utvikles hver for seg. Utover dette legger ikke React direkte føringer på hvordan applikasjonen og de tilhørende komponentene struktureres (Facebook Inc., u.d.). Samtidig vil valg av tilnærming i stor grad avhenge av egenskapene til applikasjonen og hvilke funksjonaliteter den skal tilby.

Det kan være nyttig å skissere hvordan applikasjonen og dens brukergrensesnitt skal bli seende ut, og deretter trekke slutninger om en fornuftig arkitektur. Å hente inspirasjon fra de arkitektoniske vurderingene fra serverapplikasjonen kan være en tenkt fremgangsmåte. Det er likevel viktig å være klar over at klientapplikasjonen er av vesentlig lavere total kompleksitet i prosjektgruppens tilfelle. Gruppen har derfor besluttet at fokusere på en generell retningslinje innen programvareutvikling som går på å ikke overkomplisere problemstillinger (Techopedia, u.d.).

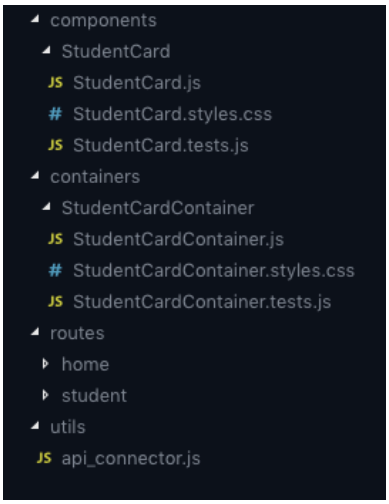
Fra en grovskisse kan det deretter skilles ut et omriss som gir en oversikt over aktuelle brukergrensesnittkomponenter, samt en liste over funksjonalitet som er nødvendig for å oppnå ønsket brukeropplevelse. I 0 er det blitt utarbeidet en enkel grovskisse og tilhørende omriss for klientapplikasjonen. Her blir det skilt mellom vanlige komponenter som `Button`, og beholderkomponenter som `StudentCardContainer`. Vanlige komponenter har som eneste funksjon å gjengi en avgrenset del av brukergrensesnittet, som for eksempel en knapp. Beholderkomponenter er komponenter som utfører mer logikk, som for eksempel å hente og prosessere data eller styre hvilke komponenter som skal eksistere i beholderkomponenten sin kontekst.

Som forklart vil en SPA ikke ha behov for å oppdatere hele nettleservinduet dersom en bruker navigerer til nytt innhold. I konteksten av en SPA er det vanlig å skille mellom ulike stier som representerer ulike typer innhold som det kan navigeres til. I en tradisjonell webapplikasjon ville dette vært representert med ulike nettsider. Skissen i 0 illustrerer én av disse stiene: `http://ClientApp/api`. I tillegg har applikasjonen stien `http://ClientApp/home` som ikke er representert i skissen.

De separate konseptene kan settes sammen i følgende prosjektstruktur:

- components: enkle komponenter som bare skal gjengi en del av brukergrensesnittet (`Button`, `StudentCard`, `UserInfo`, ...)
- containers: smarte komponenter som kan hente og prosessere data, og styre hvilke komponenter som lever i kontaineren sin kontekst
- routes: komponenter som representerer en sti til en separat enhet med unikt innhold
- utils: all funksjonalitet som ikke passer i de øvrige kategoriene (validatorklasser, api-tilkoblingsklasser, med mer

I tillegg til denne inndelingen har hver av de ulike komponenttypene sin egen interne mappestruktur. Mappen for en gitt komponent inneholder kildekoden som representerer komponenten samt tilhørende styling- og testfiler for den gitte komponenten. Fordelen med denne inndelingen er at nye utviklere umiddelbart får et overblikk over hvilke konsepter som eksisterer i applikasjonen. Videre er komponentene og alle eventuelle tilleggsfiler og -klasser plassert i én mappe, noe som resulterer i at all informasjon om en gitt komponent ligger på ett og samme sted. Figur 4.15 viser klientapplikasjonens overordnede prosjektstruktur.



Figur 4.15 - Prosjektstruktur klientapplikasjon

4.6.2 Autentiseringsflyt

I 4.2.3 ble implementasjonen av en typisk autentiseringsflyt forklart i detalj. Klientapplikasjonen vil være en naturlig deltaker i denne modellen, gitt at den integreres med samme identitetsplattform. Applikasjonen kan hente en tilgangstoken fra den valgte identitetsplattformen, Auth0, og kan videre bruke denne for å få tilgang til webAPI-et til serverapplikasjonen. Auth0 tilbyr utvidelsesbiblioteket «auth0-js» for JavaScript som tar seg av oppkobling til Auth0 sitt API for autentisering, noe som forenkler prosessen. Klientapplikasjonen samt hvilke tilganger den skal ha registreres i administratorkonsollen til Auth0. I 4.2.3 ble selve webAPI-et representert ved serverapplikasjonen registrert hos Auth0. Videre spesifiseres det at klientapplikasjonen skal få mulighet til å spørre om tilgangstokener for dette spesifikke webAPI-et. Dette er derimot ikke tilstrekkelig for å kunne fullstendig autorisere seg mot webAPI-et. Det må implementeres en innloggingsmekanisme i klientapplikasjonen, og et gitt sett med brukere må registreres.

Dersom det er ønskelig har Auth0 støtte for å opprette brukere og en tilhørende bruker- og passorddatabase som kan hostes direkte hos Auth0. Alternativt kan en av de mange støttede identitetstilbyderne som Facebook, Google eller Microsoft, benyttes. Dersom en person har registrert en bruker hos en av disse aktørene, kan denne brukeren registreres og benyttes for innlogging og autentisering for webAPI-et. Prosjektgruppen valgte å ta i bruk Facebook-brukere for innlogging, men dette kunne like gjerne vært en av de andre tilbyderne. Fordelen med Facebook er den høye dekningsgraden blant befolkningen, da nesten samtlige innbyggere har registrert en konto her. Videre kan det spesifiseres applikasjonen ønsker ekstra tokener basert på hva Facebook tilbyr. Facebook følger OIDC-standarden (OpenID Foundation, u.d.) som betyr at Auth0 kan konfigureres til å returnere en egen identifikasjonstoken. Et slikt token kan være nyttig da det kan brukes til å hente ut brukerinformasjon som navn, brukernavn og profilbilder, som videre kan brukes for å berike funksjonaliteten til klientapplikasjonen.

Brukere som skal få tilgang til webAPI-et kan registreres med sin tilhørende Facebook-konto i Auth0 sin administratorkonsoll. Videre tildeles roller, og eventuelle andre attributter, til disse brukerne for å kunne utføre tilgangskontroll. I klientapplikasjonen benyttes auth0-js-biblioteket for å la brukere logge seg inn med en Facebook-konto. Dersom innloggingen er suksessfull vil et tilgangstoken sendes tilbake til klientapplikasjonen og lagres i klientens internminne. Videre implementeres webAPI-spørring gjennom JavaScript sitt Fetch-API. Hver gang klientapplikasjonen sender en forespørsel til webAPI-et til serverapplikasjonen, vil tilgangstokenet legges ved. Responsen fra API-et vil variere, basert på hvilke roller som er tildelt brukeren og tokenet.

4.6.3 Pakking og utrulling

De samme problemstillingene og konseptene fra 4.5 gjelder også for klientapplikasjonen. Det vil derimot være forskjell på programvareavhengigheter, -biblioteker og runtimes som benyttes, og dette må skreddersys spesifikt for klientapplikasjonen. React er som sagt et utvidelsesbibliotek for JavaScript som oppdateres hyppig (Facebook Inc., u.d.). I noen tilfeller innfører oppdateringer endringer som resulterer i at applikasjoner som benyttet tidligere versjoner slutter å fungere. Det er med andre ord viktig at det til enhver tid opprettholdes riktig versjon av React og andre avhengigheter. Dette gjelder både i utviklingsperioden og i pakking- og utrullingsperioden.

Når en applikasjon er klar for å benyttes av en eller flere sluttbrukere, benevnes denne ofte som produksjonsklar. Flere av verktøyene og avhengighetene som benyttes under utvikling er ikke nødvendig å inkludere i en produksjonsklar versjon av applikasjonen. I konteksten av pakking og utrulling av en applikasjon er det derfor ofte vanlig å skille mellom én versjon for utviklingsmodus og én for produksjonsmodus. Dette resulterer i to separate Docker-filer: `Dockerfile` (produksjonsmodus) og `Dockerfile.dev` (utviklingsmodus) som vil konstrueres som separate Docker-images.

I Figur 4.16 vil Docker generere et image for utviklingsmodus basert på et JavaScript runtime kalt node. Videre vil alle nødvendige avhengigheter bli installert som spesifisert i tilhørende konfigurasjonsfiler med `package`-prefiks. I noen tilfeller brukes utviklingsverktøy også til å generere produksjonsklare filer. Applikasjoner utviklet med React buker ofte verktøyet create-react-app til både utvikling og bygging. For at Docker skal kunne bygge produksjonsklare filer ved hjelp av slike verktøyer, må et utviklingsmiljø og tilhørende verktøy være tilgjengelig. Prosjektgruppen har valgt å bruke en mekanisme for Docker kalt «multistage-builds» for å spesifisere at det først skal settes opp et miljø tilsvarende utviklingsmodus. Deretter brukes dette til å generere produksjonsklare filer. Utviklingsmiljøet blir til slutt forkastet siden dette ikke lenger er behov for. Figur 4.17 viser et slikt oppsett, hvor produksjonsklare filer til slutt blir hostet i en kontainer basert på en nginx-server.

```
# Add the base image
FROM node:10.15.3-alpine

# Create and set the working directory
WORKDIR /usr/src/reference-project-react

# Install dependencies
COPY package*.json ./
RUN npm ci

# Set default startup command
CMD ["npm", "start"]
```

Figur 4.16 - Dockerfile for utvikling

```
# This dockerfile utilizes multistage builds

# 1. First build stage: Build a temporary image for generating production-ready files
## Add the base image
FROM node:10.15.3-alpine as builder

## Create and set the working directory
WORKDIR /usr/src/reference-project-react

## Install dependencies
COPY package*.json ./
RUN npm ci

## Generate production build of React app
COPY . ./
RUN npm run build

# 2. Second build stage: Build the production image
## Add the base image
FROM nginx:1.16.0-alpine

## Extract files from first build stage
COPY --from=builder /usr/src/reference-project-react/build /usr/share/nginx/html
COPY nginx.conf /etc/nginx/nginx.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Figur 4.17 - Dockerfile for produksjonsmodus

5 EVALUERINGER

5.1 Evalueringsmetode

I 3.5 ble det presentert tre hovedmetoder for evaluering av prosjektresultatene: evaluering gjennom direkte tilbakemelding fra oppdragsgiver, kontinuerlig testing av løsningen og intern oppfølging gjennom Kanban. Disse har i varierende grad innfridd forventningene for hva som kan ansees som verdifulle tilbakemeldinger.

5.1.1 Direkte tilbakemelding fra oppdragsgiver

I oppstarten av prosjektet ble det etter samtaler med oppdragsgiver utarbeidet en anbefalt arbeidsflyt. Formålet med arbeidsflyten var i hovedsak å legge opp til at hver funksjonalitetsleveranse av prosjektgruppen ble direkte evaluert av oppdragsgiver. Prosjektets sluttbrukere vil i hovedsak bestå av utviklere og andre med teknologisk bakgrunn. Det var derfor spesielt verdifullt å få kontinuerlige tilbakemeldinger av personer som faller innenfor målgruppen til prosjektet. For å etterfølge arbeidsflyten valgte gruppen å bruke Git, et versjonshåndteringsverktøy, og GitHub for lagring og hosting av kildekoden.

Arbeidsflyten er direkte sammenlignbar med GitHub sin standardiserte arbeidsflyt (GitHub, 2017). En hovedversjon av prosjektet ligger til enhver tid lagret i GitHub. Når en av prosjektdeltakerne starter arbeidet på en ny tilleggsfunksjonalitet blir dette gjort på en isolert kopi av hovedversjonen. Dersom det blir vurdert at tilleggsfunksjonaliteten er klar for integrasjon i hovedversjonen, opprettes det en såkalt «Pull Request» hos GitHub. En Pull Request representerer et ønske om å integrere en funksjonalitet, og alle deltakere i prosjektet blir informert om dette og får mulighet til å vurdere. Dette inkluderer de andre prosjektdeltakerne i tillegg til oppdragsgiver.

Prosjektet i GitHub er konfigurert slik ny funksjonalitet bare kan integreres dersom endringene har blitt godkjent via en Pull Request av minimum én annen deltaker. Dette er et viktig moment siden det her er oppdragsgiver som har fått i oppgave godkjenne eller avslå en gitt Pull Request. Oppdragsgiver har videre mulighet til å spesifisere eventuelle nødvendige endringer som trengs før godkjenning kan gjennomføres. På denne måten forsikrer prosjektgruppen seg om at samtlige leveranser til prosjektet er på nivå med oppdragsgiver sine standarder. I tillegg gir dette gruppen konkrete tilbakemeldinger på hvilke deler av de ulike kildekodefilene og eventuell øvrig utforming som må forbedres.

5.1.2 Kontinuerlig testing av løsningen

Løsningen har ikke hatt noen form for kompleks forretningslogikk og det har heller ikke blitt satt av ressurser til automatisert integrasjons- og funksjonell testing. Prosjektgruppen har derimot sett verdien i å implementere automatiserte enhetstester for de større programvareenhetene i prosjektet. Optimalt sett burde enhetstestene blitt implementert i forkant av implementasjonen av de respektive

enhetene. Dette viste seg å bli vanskelig å gjennomføre i praksis, siden det krever en god forståelse for enhetene og teknologiene som skal testes. Gruppen så seg derfor nødt til å avvente implementasjon av enhetstester til først etter en del iterasjoner av programvareenhetene var gjennomført. Først da kunne implementasjon av testene blir gjort med selvsikkerhet om at gruppen var i besittelse av tilstrekkelig kunnskap.

Første iterasjon av enhetstester for `Web.API` ble implementert omlag midtveis i prosjektet, og utbedret kontinuerlig. Mot slutten av prosjektet ble det også utarbeidet enhetstester for `Web.Infrastructure.Repository`. De automatiserte testene fikk derfor mindre innvirkning på den totale evalueringen av prosjektets resultater enn først planlagt. For å veie opp for mangel på enhetstester ble mye testing gjennomført manuelt. I første omgang gikk dette på å gjennomføre bygging og kompilering av programkoden. Videre ble det også gjennomført manuelle tester mot webAPI-et, spesielt i forbindelse med verifisering av autentiserings- og autoriseringsflyt.

Oppdragsgiver hjalp etterhvert prosjektgruppen med å få satt opp en Jenkins-server for kontinuerlig utrulling av siste hovedversjon av prosjektet. Her ble kildekoden hentet fra GitHub, for så å gjennomføre bygging og kompilering automatisk. Dette viste seg å bli en viktig bidragsyter til testing, da Jenkins gir ytterligere feilmeldinger dersom bygging og kompilering av kildekoden ikke lar seg gjennomføre.

5.1.3 Intern oppfølging gjennom Kanban

Opprinnelig var tanken at utviklingsmetoden som prosjektgruppen la til grunn ved oppstart skulle spille en større rolle i den kontinuerlig evalueringen av resultatene. Det viste seg derimot at det ikke var mer informasjon å hente her enn det gruppen allerede fikk gjennom arbeidsflyten som spesifisert i 5.1.1. GitHub Project Boards ble derfor mer en supplerende oversikt over fullføringen av de ulike funksjonalitetene.

5.2 Evalueringsresultater

Oppdragsgiver har gitt prosjektgruppen utfyllende tilbakemeldinger på hver enkelt enhet som har blitt utviklet, gjennom GitHub sin Pull Request mekanisme. Dette har ført til at andre problemstillinger, løsninger og alternativer enn det opprinnelig lagt til grunn i Pull request-en har blitt oppdaget og vurdert. Pull request-ene har gjennomgått varierende antall iterasjoner, inntil oppdragsgiver til slutt har godkjent disse og latt funksjonaliteten bli integrert i hovedversjonen. Dette har resultert i at samtlige funksjonelle og ikke-funksjonelle egenskaper ved prosjektet har blitt evaluert fra flere hold, noe som i sin tur har styrket kvaliteten. Samtidig har det faktum at gruppen har fulgt oppdragsgivers anbefalte arbeidsflyt, gitt en helt annen oppfatning av hva som forventes i forbindelse med samarbeid på kildekode i et virkelig scenario. Etterhvert i prosjektet har gruppen lagt mer vekt på at disse Pull Request-ene skal dokumenteres godt, slik at oppdragsgiver i større grad kan forstå

hva gruppen har tenkt undervegs i implementasjonene. Dette har medført en mer åpen og verdifull dialog mellom gruppen og oppdragsgiver i forbindelse med utformings- og implementasjonsvalg. Siden oppdragsgiver også faller i målgruppen til produktet, har dette gitt spesielt verdifulle tilbakemeldinger i forbindelse med sluttbrukernes behov.

Ved å utføre manuelle tester på integrasjoner og konfigurasjonstung funksjonalitet har gruppen fått en høyere grad av selvsikkerhet rundt løsningen. Det har vært et viktig fokus å forsikre seg om at funksjonaliteten fungerer på tvers av utviklingsmiljøer og -systemer, noe som har latt seg gjøre blant annet gjennom kontaineriseringsteknologi. En bakside med manuell testing er derimot at mye av tidsressursene medgår til dette fremfor implementasjon. Likevel må det ansees som nyttig da manuell testing i større grad har gjort at gruppen har måttet gå i dybden på teknologiene som har blitt brukt. Dette har gjort gruppen oppmerksom på tidligere ukjente detaljer som videre har bidratt til økt læreutbytte. Slike læreutbytter har vært særdeles nyttig i påfølgende iterasjoner for de ulike programvareenhetene, da problemer og løsninger blir gjenkjent raskere. Etterhvert har det i større grad blitt tatt i bruk automatisert testing, både gjennom automatisk enhetstesting og automatisk bygging og kompilering av kildekode på Jenkins-serveren. Dette kombinert med den manuelle testing har bidratt til at gruppen, med høy sannsynlighet kan være sikre på at det til enhver tid ligger stabil og eksekverbar kode i hovedversjonen av prosjektet.

6 RESULTATER

6.1 Oppdragsgivers vurdering av resultater

I løpet av prosjektet har oppdragsgiver hatt en nøkkelrolle i for prosjektgruppens fremgang, gjennom god oppfølging og kontinuerlige tilbakemeldinger. Det har blitt gitt tilbakemeldinger både direkte i kildekoden gjennom GitHub sin Pull Request mekanisme, og gjennom kommunikasjonskanalen Slack. Tilbakemeldingene undervegs har vært konstruktive, og har bidratt til at gruppens sluttresultater har nådd høyest mulig kvalitet. Ved avslutningen av prosjektet ble oppdragsgiver forespurt om å svare på et sluttevalueringsskjema, vedlagt i Appendiks E, i forbindelse med gjennomføring av prosjektet og tilhørende resultater. Det ble også avholdt et avslutningsmøte i siste fase av prosjektet.

Kort oppsummert føler oppdragsgiver at kravene for oppgaven har blitt innfridd, og at prosjektgruppen har levert til forventningene. Oppdragsgiver ser for seg at løsningen vil kunne være en verdifull bidragsyter til å forenkle oppstart av nye prosjekter i samme tech-stack. Samtidig mener oppdragsgiver at det i stor grad vil være lettere for nye utviklere å sette seg inn i tech-stacken som løsningen er basert på. Løsningen er også blitt utformet på en måte som gjør den lett å utvide til å inkludere andre problemstillinger, teknologier og utformingsmønstre. Dette er som resultat av at løsningen blant annet fremhever god praksis for utforming og implementasjon.

Det ble også kommentert at utvidelsen av problemstillingen med klientbasert teknologi ga gruppen en ekstra belastning som kunne vært unngått dersom den opprinnelige kravspesifikasjonen hadde blitt overholdt. Oppdragsgiver mener den opprinnelige kravspesifikasjonen i seg selv var stort nok gitt omfanget til en bacheloroppgave. Det er enighet mellom partene at ressursene som medgikk her sannsynligvis kunne bidratt til ytterligere funksjonalitet for serverapplikasjonen. Til tross for dette har introduksjonen av klientteknologi bidratt til lærerike diskusjoner mellom oppdragsgiver og prosjektgruppen, som kan ansees som verdifullt i et større perspektiv.

Det ble påpekt en generell mangel på supplerende dokumentasjon for løsningen, spesielt i forbindelse med integrasjoner. Partene ble enige om å utarbeide dokumentasjon i etterkant av den formelle sluttdatoen for prosjektet, da prosjektgruppen ser verdien i dette for begge parter. Forøvrig føler oppdragsgiver at kommunikasjonen og arbeidsflyten til prosjektgruppen har vært god.

6.2 Prosjektdeltakernes vurdering av resultater

Prosjektgruppen selv er generelt fornøyd med sluttproduktet som prosjektet resulterte i. Som tidligere nevnt var en del av teknologiene fullstendig ukjent for deltakerne ved oppstart. At det faktisk ble levert noe av verdi på kravene relatert til disse teknologiene, er noe gruppen anser som en suksess i seg selv. Når det gjelder de ikke-funksjonelle kravene som arkitektur og utformingsmønstre er gruppen også stort sett fornøyd. Deltakerne mener spesielt at de arkitektoniske valgene var godt begrunnet og i stor grad bidro til en gjenbrukbar løsning. På den andre siden skulle prosjektgruppen ønske at de ble utviklet en mer omfattende og virkelighetsnær domenemodell. Dette er noe som kunne bidratt til en høyere grad av tilfredsstillelse for prosjektgruppen i seg selv, men dog ikke nødvendigvis i forbindelse med oppdragsgiver sine krav.

Mot slutten av prosjektet valgte gruppen å fokusere på å fullføre funksjonelle krav fremfor å utarbeide supplerende dokumentasjon for løsningen innen den offisielle sluttdatoen. Her tas det selvkritikk da samtlige deltakere mener en slik dokumentasjon ville bidratt til et bedre helhetsinntrykket av løsningen. Samtidig ville dette bidratt til at det i større grad ble enklere å forstå, ta i bruk og utvide løsningen.

7 DISKUSJON

7.1 Planlegging

Som nevnt innledningsvis i rapporten hadde prosjektgruppen ikke helt klart for seg hva sluttproduktet skulle være, og heller ikke hvordan veien til et sluttprodukt skulle se ut. For å sikre at prosjektet ble utviklet i henhold til oppdragsgivers ønsker, brukte prosjektgruppen mye tid i planleggingsfasen. Gruppen analyserte oppgavebeskrivelsen og referanseprosjekter i andre teknologier som var gjort tilgjengelig av oppdragsgiver. Dette ble gjort for å trekke ut potensielle målsetninger og kravspesifikasjoner. Etter hvert ble det produsert noen prototyper for hvordan utformingen av prosjektet kunne være, som ble brukt som utgangspunkt i videre diskusjoner med oppdragsgiver.

På tross av at arkitektonisk utforming av løsningen ikke var fremhevet i kravspesifikasjonen, valgte gruppen å bruke betydelig med tid på akkurat dette. Omtrent halvannen uke ble dedikert til utforskning og diskusjon av forskjellige kandidat-arkitekturer, tiden brukt på dette var mer enn opprinnelig antatt. Siden gruppen brukte såpass mye tid på arkitektur, burde dette kanskje kommet tydeligere frem i kravspesifikasjonen. Arkitektur hadde så mye å si for det videre arbeidet at det måtte prioriteres å gjøres ordentlig.

I planleggingsfasen brukte gruppen også mye tid på å utforske dokumentasjon og forskjellige implementasjoner av teknologiene beskrevet i kravspesifikasjonen.

7.2 Gjennomføring

Før prosjektgruppen startet implementasjonen av kravspesifikasjonen, valgte gruppen å innføre en veldefinert arbeidsflyt som skulle følges underveis i gjennomføringen av prosjektet. Denne arbeidsflyten er tett knyttet til bruk av Github, «feature branching» og pull requests som er beskrevet i mer detalj under punkt 5.1.1.

I starten av implementasjonsfasen jobbet gruppen sammen for å implementere et skjelett for serverapplikasjonen. I denne tidsperioden var fokuset å implementere en prosjektstruktur i henhold til planlagt prosjektarkitektur og grunnleggende funksjonalitet i form av endepunkter, datahåndtering og dokumentasjon. Dette skjelettet ble brukt som utgangspunkt når gruppen videre gikk over til selvstendig parallelt arbeid i form av feature branching.

Prosjektplanen kan gi inntrykk av at implementasjonsfasen var en sammenhengende periode med definert start og slutt. Dette er delvis sant, men prosjektgruppen har i stor grad fulgt en iterativ tilnærming til utviklingen. For hver tilleggsfunksjonalitet som ble lagt til gikk gruppen gjennom fem delfaser:

1. Analyse av kravspesifikasjon.
2. Innhenting av dokumentasjon og utforming av løsning.
3. Implementasjon.

4. Testing.
5. Vurdering og verifisering.

På tross av diverse utfordringer underveis i utviklingsfasen gikk utførelsen av prosjektet stort sett bra, men noen punkter i kravspesifikasjonen var mer ukjent og krevende enn andre. Spesielt oppgaver knyttet til CI/CD og kontainerisering viste seg å være en utfordring, ettersom ingen av gruppemedlemmene hadde nevneverdig erfaring eller kunnskap rundt dette.

7.3 Konsekvenser av valgt fremgangsmåte

Prosjektet

Med bakgrunn i prosjektdeltakernes spesialiseringer valgte prosjektgruppen å utvide oppgavens omfang til å inkludere et prosjekt i React. I etterkant har gruppen konkludert med at denne utvidelsen førte til at oppgavens omfang ble større enn antatt og dette gikk utover arbeidet med serverapplikasjonen, som var hovedmålet til prosjektet. Hadde gruppen valgt å begrense omfanget av oppgaven til å ha fullt fokus på serverapplikasjonen kunne de heller anvendt tiden brukt på klientapplikasjonen til å gjøre funksjonaliteten i serverapplikasjonen fyldigere. I avsluttende møte med oppdragsgiver fikk gruppen støtte under denne konklusjonen, samtidig som de mente at de selv burde frarådet gruppen fra å utvide oppgavens omfang.

Planlegging

Som nevnt brukte prosjektgruppen mye tid i planleggingsfasen for å forstå oppgaven, valg av utforming og utforskning av de forskjellige teknologiene i kravspesifikasjonen. Gruppen mener at dette var nødvendig for å unngå misforståelser, identifisere delmål og lage tidsestimater for de identifiserte delmålene. Gruppen hadde hele tiden godt definerte mål og en enighet om hva som skulle bli gjort. Gruppen mener også at de fant en god balanse mellom for mye og for lite planlegging. Endringer underveis er uunngåelig og derfor ønsket gruppen å planlegge nok til å vite hva som måtte bli gjort, samtidig som de var åpne for endringer.

Det ble også brukt mye tid på valg av arkitektur og prosjektstruktur. På tross av at dette ikke ble fremhevet i kravspesifikasjonen mener prosjektgruppen at dette var god bruk av ressurser i planleggingsfasen. Det kan være vanskelig å evaluere mindre deler av prosjektet som prosjektstruktur og arkitektur. Det er ikke nødvendigvis et fasitsvar på en optimal arkitektur og det finnes forskjellige utformingsmønstre som søker å løse de samme problemstillingene. Prosjektgruppens erfaring med den utarbeidede arkitekturen er derimot entydig: den utarbeidede arkitekturen førte til at gruppen startet med et solid grunnlag i form av prosjektstruktur og arkitektur som videre tilrettelagte for lettere utvidelse av funksjonalitet i gjennomføringsfasen. En god programvarearkitektur er også et viktig aspekt som støtter opp mot målet til prosjektet ved å tilrettelegge for gjenbruk, skalering, videreutvikling og utbytting av moduler.

Gjennomføring

Valgt arbeidsflyt er definitivt noe prosjektgruppen er fornøyde med. Bruken av Github, feature branching og pull requests førte til at arbeidsflyten underveis var god, både prosjektdeltakere og oppdragsgiver hadde til enhver tid oversikt over arbeidet som var blitt gjort og hva som gjensto. Denne måten å arbeide på sikret også kontinuerlig kvalitetssikring og kommunikasjon mellom gruppen og oppdragsgiver.

Gruppedynamikken innad i gruppen har vært god gjennom hele prosjektet. Prosjektdeltakerne har jobbet konsekvent og vært flinke til å jobbe selvstendig på oppgaver gitt, samtidig som prosjektdeltakerne har hjulpet hverandre med problemer som har dukket opp underveis. Prosjektgruppen er også fornøyde med å ha opprettholdt kommunikasjonen med prosjekteierne underveis. Statusmøter, kommunikasjon gjennom Slack og GitHub har gjort at nødvendige endringer ble oppdaget tidlig og prosjektet til enhver tid beveget seg i riktig retning i henhold til oppdragsgivers ønsker.

8 KONKLUSJON

De fleste utviklere må forvente å møte ukjente teknologier og problemstillinger gjennom sin karriere. På lik linje må bedrifter som tilbyr utviklingstjenester forvente at kundebehov endres, nye teknologier blir etterspurt og ukjente problemstillinger dukker opp. Å navigere i kompleks eller mangelfull dokumentasjon er en tidkrevende prosess i begge tilfeller, som ofte bidrar til økt tidsbruk på utviklingsprosjekter. Dette er tidsressurser som i stedet kunne blitt brukt på å levere ytterligere verdi til kundene. Som en løsning på dette utvikler EVRY referanseprosjekter. Teknologier og typiske problemstillinger blir valgt ut basert på dagens behov, og satt sammen i applikasjoner og systemer. Disse fungerer som en grunnmur for intern opplæring og forenklet oppstart av nye prosjekter.

Dette var konteksten for oppgaven som prosjektgruppen tok fatt på, på vegne av EVRY Bergen. Det var behov for å utvikle et referanseprosjekt som skulle løse typiske problemstillinger i forbindelse med programvareutvikling i et gitt sett av nye teknologier. Prosjektgruppen så dette som en mulighet for å utvide sin kompetanse i flere aktuelle områder innenfor systemutvikling og arkitektur. De overordnede målene for prosjektet hadde sterk tilknytning til konteksten. Ønsket var å utvikle et referanseprosjekt som ga en enkel og klar oversikt over teknologiene og problemstillingene. Løsningen skulle videre være lett å utvide og integrere inn i konkrete utviklingsprosjekter. Generelt sett var det også ønsker om at løsningen skulle kunne tas i bruk både som en læringsressurs for utviklere, men også som et grunnlag for prosjekter som benytter samme type teknologi.

Innledningsvis besluttet prosjektgruppen å legge et vesentlig fokus på valg av arkitektur. Vurderingene gruppen gjorde i oppstartsfasen ledet til konklusjonen om at flere av målene kunne oppnås bare dersom prosjektet som helhet ble bygget på en solid arkitektur. Flere av egenskapene som den endelige arkitekturen potensielt sett kunne bidra med ble gjenkjent i egenskapene som de overordnede målene etterspurte, noe som bidro til konklusjonen. De arkitektoniske vurderingene som ble gjort førte også med seg verdifull lærdom om de ulike komponentene som prosjektet kom til å bestå av, og hvordan disse på beste måte kunne kobles sammen.

Videre tok prosjektgruppen i bruk en spesifikk arbeidsflyt på oppfordring fra oppdragsgiver. Arbeidsflyten la opp til at funksjonalitetsleveransene til enhver tid holdt seg innenfor kvalitetsrammene til oppdragsgiver. Dette ble en verdifull bidragsyter til måloppnåelsen, da oppdragsgiver fikk fullstendig innsyn og mulighet til å bidra gjennom konstruktive tilbakemeldinger direkte i kildekoden. Som resultat av dette ble også problemer oppdaget på et tidligere stadium. Et tett samarbeid med oppdragsgiver kombinert med et grundig forarbeid førte derfor til at prosjektgruppen klarte å levere på nesten samtlige krav i kravspesifikasjonen.

Prosjektgruppen sitter igjen med et referanseprosjekt som tilfredsstiller kravspesifikasjonen og målene til oppdragsgiver, innenfor rammene som ble gitt.

Oppdragsgiver mener sluttresultatet er et godt utgangspunkt for et optimalt referanseprosjekt. Dette betyr med andre ord at referanseprosjektet ikke er fullstendig, og både oppdragsgiver og prosjektgruppen har sett rom for forbedringer og potensielle tilleggsfunksjonaliteter. For det første finnes det mange aktuelle problemstillinger i forbindelse med programvareutvikling som ikke har vært et tema i denne oppgaven. Samtidig er det flere av komponentene i referanseprosjektet som er av relativt enkel art. Som beskrevet i rapporten er det enkelte spesifikasjoner og standarder hvor prosjektgruppen ikke har hatt kapasitet til å implementere disse i sin fullstendige form.

Prosjektgruppen og oppdragsgiver er begge enige om at sluttresultatet har høyt potensiale for fremtidig videreutvikling. Gitt mer tid ville prosjektgruppen fokusert på noen hovedområder. Å utvide domenemodellen og introdusere realistiske brukstilfeller og tilhørende forretningslogikk anser prosjektgruppen som av stor verdi. Dette vil gi sannsynligvis medføre endringer i resten av prosjektet som i sin tur vil fremtvinge utformingsmessige revurderinger som kan bidra til økt kvalitet totalt sett. Videre bør det i større grad implementeres fullstendige spesifikasjoner og standarder som for eksempel retningslinjene fra REST og OIDC standarden for autentisering. Som resultat av en eventuell utvidet domenemodell vil også databasemodellen måtte oppdateres. Dette vil være gunstig siden Entity Framework er et kraftig rammeverk med mye ubrukt funksjonalitet som ikke har blitt fremhevet. I forbindelse med kontinuerlig integrasjon og utrulling er det flere aspekter som ennå ikke har blitt undersøkt.

9 LITTERATURLISTE

Avram, A. (2016, 6 21). *Microsoft REST API Guidelines Are Not RESTful*. Hentet fra Webområde for InfoQ: https://www.infoq.com/news/2016/07/microsoft-rest-api/

Berners-Lee, T., Fielding, R. T., Frystyk, H., Gettys, J., Leach, P., Masinter, L., & Mogul, J. (1999). *Hyper Text Transfer Protocol -- HTTP/1.1.* Internet Engineering Task Force.

Bradley, J., Jones, M., & Sakimura, N. (2015). *JSON Web Token (JWT).* Internet Engineering Task Force.

Datatilsynet. (2017, 3 7). *Kryptering | Datatilsynet*. Hentet fra Webområde for Datatilsynet - Informasjonssikkerhet: https://www.datatilsynet.no/rettigheter-og-plikter/virksomhetenes-plikter/informasjonssikkerhet/kryptering/

Docker inc. (2019, Juni 1). *Overview of Docker Compose*. Hentet fra Docker docs: https://docs.docker.com/compose/overview/

EVRY NORGE AS. (2019, Mai 7). *Om EVRY*. Hentet fra Webområde for EVRY: https://www.evry.com/no/selskapet/om-oss2/om-evry/

Facebook Inc. (u.d.). *File Structure*. Hentet fra Webområde for React Docs: https://reactjs.org/docs/faq-structure.html

Facebook Inc. (u.d.). *Releases - facebook/react*. Hentet fra Webområde for Github: facebook/react: https://github.com/facebook/react/releases

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* Doktoravhandling, University of California, Irvine.

GitHub. (2017, 11 30). *Understanding the GitHub flow*. Hentet fra Webområde for Github - Guides: https://guides.github.com/introduction/flow/

ISO/IEC/IEEE. (2010). *Systems and software engineering — Vocabulary.* ISO/IEC/IEEE.

Lanciaux, R. (2017, 8 24). *A feature based approach to React development*. Hentet fra Webområde for Ryan Lanciaux blogg: https://ryanlanciaux.com/blog/2017/08/20/a-feature-based-approach-to-react-development/

Martin, R. C. (2002). *Agile Software Development, Principles, Patterns, and Practices.* Pearson Education, Inc.

Martin, R. C. (2018). *Clean Architecture.* Pearson Education, Inc.

Microsoft. (2013, Oktober 16). *Entity Framework: Microsoft Developer Network*. Hentet fra Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/gg696172(v=vs.103).aspx

Microsoft. (2016, Oktober 27). *Entity Framework Core: Microsoft Docs*. Hentet fra Microsoft Docs: https://docs.microsoft.com/en-us/ef/core/

Microsoft. (2017, 11 29). *Unit testing C# in .NET Core using dotnet test and xUnit*. Hentet fra Webområde for Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-dotnet-test

Microsoft. (2018, Februar 23). *Database Providers*. Hentet fra Microsoft Docs: https://docs.microsoft.com/en-us/ef/core/providers/

Microsoft. (2018, 11 13). *Model Binding in ASP.NET Core*. Hentet fra Webområde for Microsoft Docs: https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-2.2

Microsoft. (2018, 7 28). *Unit testing best practices with .NET Core and .NET Standard*. Hentet fra Webområde for Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices

Microsoft. (2019, 2 16). *Architectural principles*. Hentet fra Webområde for Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/architectural-principles#separation-of-concerns

Microsoft. (2019, 1 30). *Choose Between Traditional Web Apps and Single Page Apps (SPAs)*. Hentet fra Webområde for Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/choose-between-traditional-web-and-single-page-apps

Microsoft. (2019, April 7). *Introduction to ASP.NET Core*. Hentet fra Microsoft Docs: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.2

Microsoft. (2019, 4 6). *Model validation in ASP.NET Core MVC and Razor Pages*. Hentet fra Webområde for Microsoft Docs: https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation?view=aspnetcore-2.2#validation-attributes

MIT. (2011, 3 10). *IdP (Identity Provider) - Glossary - Hermes*. Hentet fra Webområde for MIT - The Knowledge Base: http://kb.mit.edu/confluence/display/glossary/IdP+(Identity+Provider)

NSubstitute. (u.d.). *NSubstitute - A friendly substitute for .NET mocking libraries*. Hentet fra Webområde for NSubstitute: https://nsubstitute.github.io/

OpenID Foundation. (u.d.). *OpenID Connect*. Hentet fra Webområde for OpenID Foundation: https://openid.net/connect/

OWASP. (2017). *OWASP Top 10 - 2017.* Hentet fra Webområde for OWASP Wiki: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

Richards, M. (2015). *Software Architecture Patterns.* Sebastopol: O'Reilly Media, Inc.

Spacey, J. (2016, Mai 27). *Simplicable.* Hentet fra Simplicable.com: https://simplicable.com/new/separation-of-concerns

Sysdig. (2018). *2018 Docker Usage Report*. Hentet fra Webområde for Sysdig Blog: https://sysdig.com/blog/2018-docker-usage-report/

Techopedia. (u.d.). *Keep It Simple Stupid Principle*. Hentet fra Webområde for Techopedia Definitions: https://www.techopedia.com/definition/20262/keep-it-simple-stupid-principle-kiss-principle

Tutorialspoint. (2019, Mai 9). *Entity Framework - Overview: Tutorialspoint*. Hentet fra Tutorialspoint: https://www.tutorialspoint.com/entity_framework/entity_framework_overview.htm

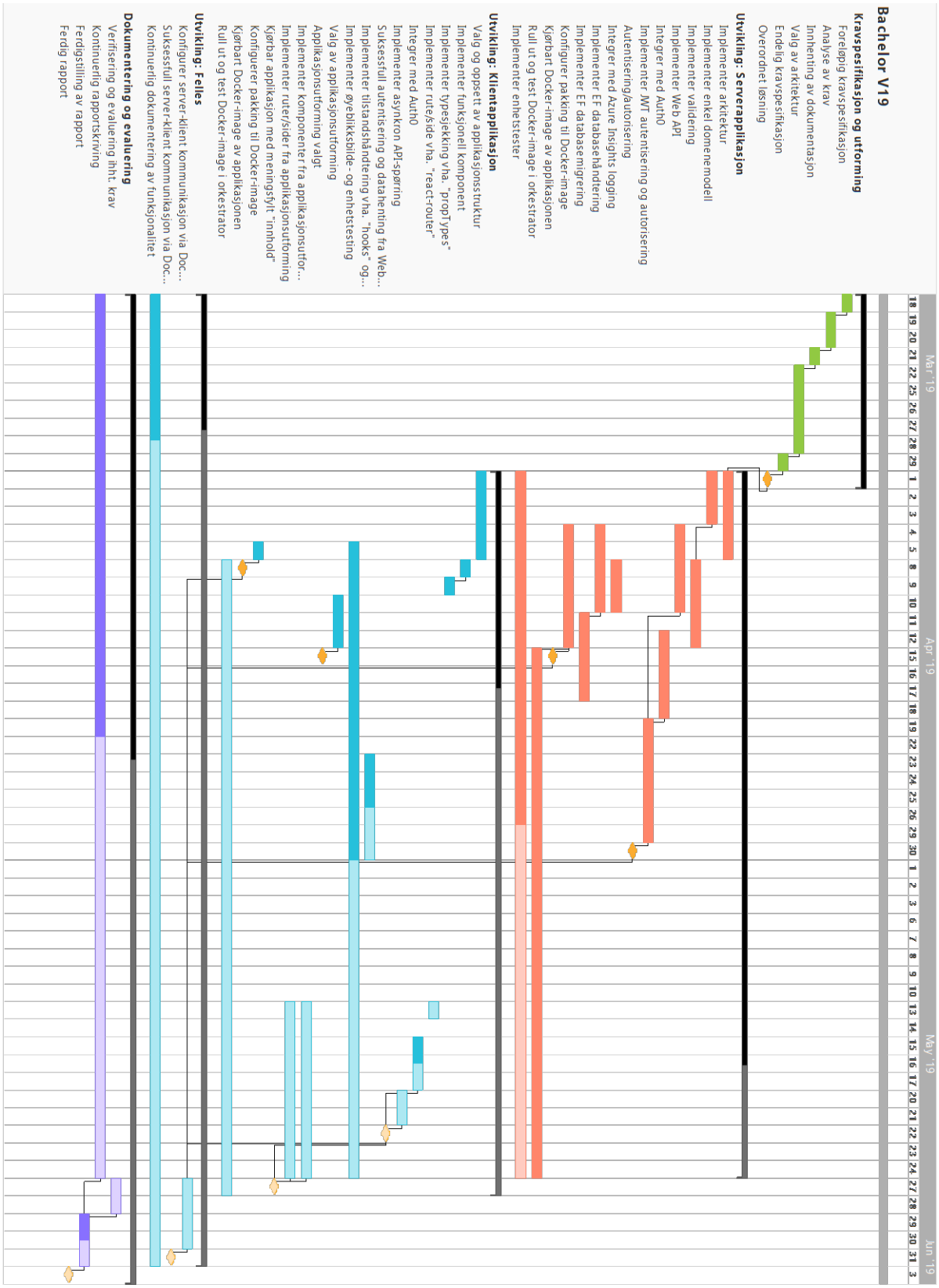
Wikipedia. (2019, Mars 15). *Robert C. Martin : Wikipedia*. Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Robert_C._Martin

10 APPENDIKS

Appendiks A Risikoliste

Risiko	Konsekvens	Sannsynlighet (1-10)	Alvorlighet av konsekvens	Forebyggende tiltak
For lite kompetanse/erfaring med de teknologiene som skal brukes til at vi klarer å gjennomføre oppgaven.	Prosjektgruppen vil ikke kunne gjennomføre prosjektet etter oppdragsgivers ønsker.	3	9	Prosjektdeltakere må sette av tilstrekkelig med tid til å sette seg inn i teknologiene. Dette kan bli gjort ved å lese på dokumentasjon, oppsøke informasjon fra andre kilder og prøve ut teknologiene i praksis.
Manglende dialog med oppdragsgiver.	Kan føre til at vår utforming av prosjektet kan avvike fra oppdragsgiver sin kravspesifikasjon.	3	7	Prosjektgruppen må sørge for å ta i bruk kommunikasjonskanalen som er satt opp for diskusjon rundt oppgaven. Prosjektgruppen kan også sette av tid til å sitte på oppdragsgivers kontor og ha hyppige møter med oppdragsgiver.
For stor arbeidsmengde per deltaker	Kan føre til at prosjektdeltakerene vil opplever arbeidsmengden som uhåndterlig og fremgangen i prosjektet vil ikke være tilstrekkelig.	5	7	Prosjektgruppen må være påpasselig med å ikke undervurdere arbeidsmengden for å gjennomføre oppgaver. Dersom prosjektgruppen opplever at arbeidsmengden går utover planlagt arbeidsmengde så vil det være et alternativ å begrense omfanget.

Appendiks B GANTT diagram



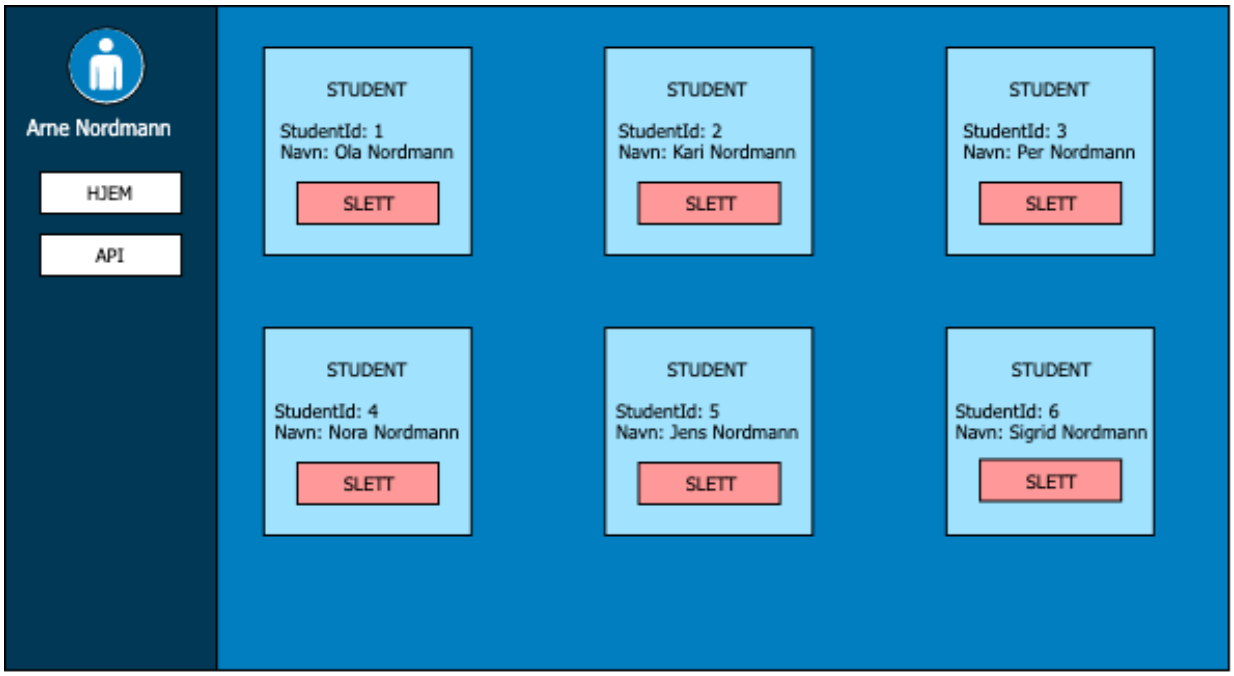
Appendiks C Ordliste

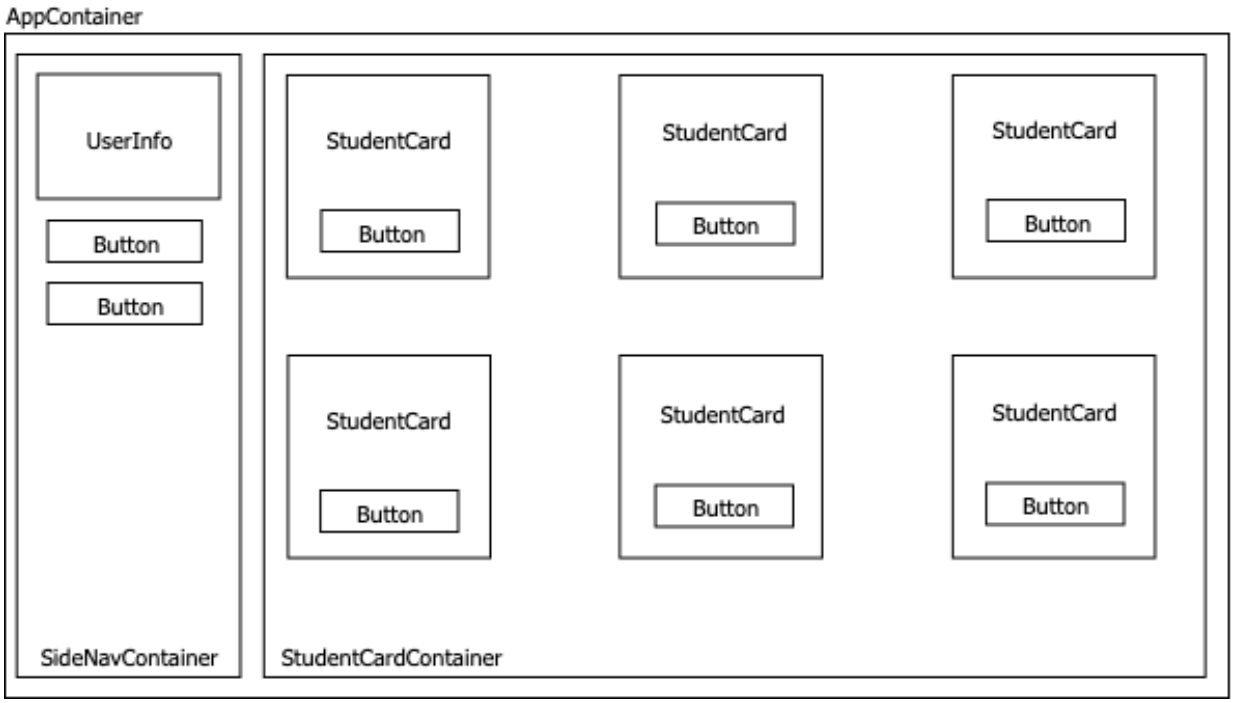
Begrep	Forklaring
AJAX (Asynchronous JavaScript and XML)	En mekanisme for asynkron utveksling av forespørsel og responser i JavaScript.
API	Applikasjonsprogrammeringsgrensesnitt (engelsk: Application Programming Interface).
Backend	Et samlebegrep for teknologi som omhandler utforming og implementasjon av tjenester og systemer, som frontendapplikasjoner kan benytte, ofte i konteksten av vertsmaskiner.
Kobling	I sammenheng av systemutvikling: et mål for graden av avhengighet mellom programvareenheter/moduler.
CRUD	I programmering står CRUD for «create», «read», «update» og «delete». Dette er de fire grunnleggende operasjoner knyttet til persistering.
Deklarativ programmering	Et deklarativt programmeringsspråk er et type språk hvor man beskriver resultatet man ønsker å oppnå, uten å angi hvordan det skal produseres. Eksempler: Prolog og SQL.
Avhengighetsinjeksjon	Engelsk: «Dependency Injection». En teknikk hvor et objekt (eller statisk metode) forsyner et annet objekt med avhengigheter. En avhengighet er et objekt som blir brukt (en tjeneste).
Extreme Programming (XP)	Et rammeverk for programvareutvikling. Extreme programming modellen anbefaler å ta beste praksis fra tidligere programvareutviklingsprosjekter til et ekstremt nivå. Eksempler på dette kan være: testing, «Code Review», inkrementell utvikling m.m.
Feature Branching	Arbeidsflyt i Git hvor hver funksjonalitet som blir utviklet skal ha sin egen dedikerte gren (branch), i stedet for den gjeldende «master-branchen».
Frontend	Et samlebegrep for teknologi som omhandler utforming og implementasjon av brukergrensesnitt, ofte i konteksten av klientmaskiner Frontend er den delen av programvaren som ligger nærmest brukeren. Det er koden som former det du visuelt ser på skjermen, og bestemmer hva som skjer når du interagerer med disse elementene.
Imperativ programmering	Et imperativt programmeringsspråk er et type språk hvor man beskriver hvordan ting skal gjøres, trinn for trinn. Eksempler: C#, Java og C++.

JSON (JavaScript Object Notation)	Et tekstbasert format for formatering av meldinger som brukes i datautveksling. Standarden har sin opprinnelse fra JavaScript sin representasjon av enkle datastrukturer.
JSX (JavaScript Syntax Extension)	En utvidelse for JavaScript, som gir en XML-lignende måte å definere React-komponenter i en trestruktur.
JWT (JSON Web Token)	JSON Web Token er en standard for overføring av informasjon mellom to parter på en sikker måte. Brukes ofte i autentiseringssammenheng.
Kontinuerlig integrasjon	Utviklingspraksis som går ut på hyppig integrasjon av kodeendringer inn i en delt kodebase samt testing av endringer tidlig og ofte.
Objektetterligning	Å lage objekter som gjenskaper oppførselen til ekte objekter på en kontrollert måte. Oftest brukt som en del av programvaretesting. Nyttig dersom man tester metoder med eksterne avhengigheter, og man vil forsikre seg om at metoden ikke påvirkes av eksterne forhold.
ORM (Object Relational Mapper)	Object Relational Mapping er en metode for å konvertere datastrukturer mellom plattformer som i utgangspunktet er inkompatible.
Parprogrammering	To utviklere jobber på samme maskin og alternerer mellom å skrive kildekode Teknikk for programvareutvikling hvor to programmerere arbeider sammen ved samme tastatur.
RUP (Rational Unified Process)	En iterativ og inkrementell programvareutviklingsprosess. RUP deler inn utviklingsprosessen i fire veldefinerte faser. Hver av disse fasene har en spesifikk plan for utførelse.
Runtime	Programvare hvor kildekode for en gitt plattform kan eksekveres i en eksekveringsmodell. Ulike programmeringsspråk/plattformer blir levert med egne runtimes.
SCRUM	Scrum er et agilt rammeverk for optimalisering av produktutvikling – i utgangspunktet programvarebaserte produkter. Rammeverket vektlegger teamwork og iterativ utvikling mot et veldefinert mål. I SCRUM har man definerte roller, eksempelvis «produkteier», «SCRUM master» og utvikler.
SLACK	Programvare laget for kommunikasjon og samarbeid i arbeidssammenheng.

SOLID	Et akronym for fem designprinsipper brukt for å gjøre programvareutforming mer forståelig, fleksibelt og vedlikeholdbart: • Single responsibility principle • Open-closed principle • Liskov substitution principle • Interface segregation principle • Dependency inversion principle
URI (Uniform Resource Identifier)	En karakterstreng som entydig identifiserer en spesifikk ressurs.

Appendiks D Konseptuelt brukergrensesnitt





Appendiks E Skjema for sluttevaluering

I hvor stor grad gir løsningen et bedre overblikk over tech-stacken og dens bruk?

0 1 2 3 **4** 5 6

I hvor stor grad tror du/dere at løsningen vil forenkle oppstart av nye prosjekter basert på tilsvarende tech-stack?

0 1 2 3 4 5 **6**

I hvor stor grad tror du/dere at løsningen vil gjøre det letter for nye utviklere å sette seg inn i tech-stacken?

0 1 2 3 4 **5** 6

I hvor stor grad har løsningen blitt utformet på en måte som fremhever god praksis for utforming av arkitektur og prosjektstruktur?

0 1 2 3 4 **5** 6

I hvor stor grad har løsningen blitt utformet på en måte som fremhever god praksis for bruk av teknologier, integrasjoner og implementasjonen rundt dette?

0 1 2 3 4 **5** 6

I hvor stor grad har løsningen blitt utformet på en måte som gjør den lett å utvide/bygge videre på?

0 1 2 3 4 **5** 6

Har gruppen oppnådd ditt/deres mål/forventninger for oppgaven?

Ja

Hvordan føler du/dere kommunikasjonen og arbeidsflyten til prosjektgruppen har vært?

Dårlig Mindre god Nøytral **God** Veldig god

Er det deler av prosjektet gruppen kunne gjort bedre eller skulle fokusert mer på?

Dere har vært flinke til å selv vurdere hvor fokuset bør ligge, og ved usikkerhet har dere spurt om hjelp/råd. Skulle kanskje ønsket at det var litt mer dokumentasjon i form av Readme etc, men dette har vi diskutert tidligere, og er nok et følge av mangel på tid.

Har du/dere eventuelle andre tilbakemeldinger?

Vi burde nok sett tidligere at både frontend og backend ble et litt stort felt å dekke. Samtidig så har det nok vært lærerikt å ha de diskusjonen vi har hatt, som kom ut av dette. Generelt sett har dere gjort en meget god jobb, og levert meget godt.