NTNU
Det skapende universitet

# Datateknisk infrastruktur for planlegging, gjennomføring og evaluering av undervisning

**Thomas Haugland Rudfoss**

Master i datateknikk
Oppgaven levert: Juni 2010
Hovedveileder: Mads Nygård, IDI
Biveileder(e): Kjell Bratbergsengen, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

# Oppgavetekst

Planlegging og gjennomføring av fag er en omfattende prosess som gjennomføres jevnlig ved instituttene på NTNU. Prosessen krever informasjon om mange ulike aspekter ved undervisningen fra studentantallet, antallet fag, antallet ansatte, budsjetterte kostnader og til evalueringsrapporter fra tidligere fag. Fram til nå har det vært opp til de enkelte ansvarlige på hvert institutt og komme fram til sine egne rutiner på hvordan denne prosessen skal gjennomføres. Noen har da benyttet seg av intern kunnskap til å lage egne databaser som inneholder mye av informasjonen, mens andre har benyttet en mer ad-hoc løsning ved å delegere arbeidet videre til de enkelte foreleserne i hvert fag. Resultatet av dette er at mye informasjon går tapt av ulike årsaker for eksempel fordi forelesere slutter og tar med seg all dokumentasjon om fag han eller hun har holdt, eller at det rett og slett glemmes. Høsten 2009 ble det gjennomført en oppgave med fokus på å forenkle og forbedre planleggingsprosessen. Oppgaven het Læringsadministrative systemer i norsk høyere utdanning og gikk ut på å konstruere en ny database med grunnlag i eksisterende læringsplattformer der man kan lagre det meste av informasjon som omfatter både planlegging og gjennomføring av fag. Fordi det allerede eksisterer et stort antall systemer og databaser som inneholder informasjon om alt fra studentantall til rombestilling er det urealistisk, og sannsynligvis upraktisk at ett system skal holde på all informasjonen. Resultatet ble en omfattende databasemodell med fokus på fleksibilitet for å støtte de ulike rutinene for fagplanlegging som hvert institutt har utarbeidet og ideer til videre arbeid om hvordan databasen skulle bli benyttet.
Denne oppgaven tar resultatet fra høsten 2009 videre og består i hovedsak av tre punkter:

Finne fram ulike løsninger til hvordan man kan benytte databasemodellen fra den tidligere oppgaven i forbindelse med fagplanlegging og gjennomføring.

Evaluere databasemodellen og utbedre den om nødvendig.

Konstruere en prototype av dette systemet som kan benyttes til å demonstrere løsningen.

Målet med oppgaven er å få fram et felles datalager for informasjonen som behøves i forbindelse med fagplanlegging som både legger til rette for effektiv dokumentasjon av ulike hendelser og aktiviteter og samtidig er fleksibelt og fremtidsrettet til effektivt å kunne benyttes av ulike institutter. Det må også være mulig å benytte dette datalageret i samkjøring med eksisterende systemer og til konstruksjon av nye framtidige løsninger.

Oppgaven gitt: 14. januar 2010
Hovedveileder: Mads Nygård, IDI

I oppgaven "Læringsadministrative systemer i norsk høyere utdanning" skrevet høsten 2009 ble det foreslått en ny databasestruktur for fagdata ved institutter på NTNU. Databasemodellen løser flere av dagens problemer ved å muliggjøre sentral lagring av informasjon om fag og deres utførelse samt relaterte emner som ressursbruk (rom, utstyr og elektroniske dokumenter). Denne oppgaven er en videreføring av den forslåtte løsningen fra 2009 med fokus på bearbeiding av databasestrukturen og utvikling av et abstraksjonslag som skal fasilitere arbeid mot databasen. Resultatet har blitt en utbedret databasemodell og en prototype av en modulær applikasjon med støtte for ulike kommunikasjonsprotokoller, dokumentformater og variasjoner i databasestrukturen. Dette systemet kan benyttes til å lagre informasjon om de aller fleste aspekter ved gjennomføringen av et fag. I tillegg kan det utvides underveis til å støtte uforutsette forandringer eller aspekter ved gjennomføringen som ikke har kommet fram tidligere.

# INNHOLDSLISTE


**1 PROBLEMSTILLING ..... 1**

**2 MULIGE LØSNINGER ..... 3**

2.1 HVORDAN HAR INSTITUTTENE ORGANISERT INFORMASJONEN I DAG? ..... 3
2.2 HVILKE OPPGAVER SKAL SYSTEMET LØSE ..... 4
2.3 HVORDAN SKAL DATABASEN IMPLEMENTERES ..... 6
2.4 HVORDAN SKAL KLIENTENE ARBEIDE MOT DATABASEN ..... 8
2.5 HVORDAN SKAL KLIENTENE KOMMUNISERE ..... 12
2.6 HVORDAN SKAL OPPGAVER UTFØRES ..... 13
2.7 KONKLUSJON ..... 14

**3 UTBEDRINGER AV DATABASEN ..... 17**

3.1 TILGANGSKONTROLL OG SIKKERHET ..... 17
3.2 FAG ..... 20
3.3 RESSURSER ..... 23

**4 ABSTRAKSJONSLAGET ..... 25**

4.1 OVERORDNET STRUKTUR ..... 26
*4.1.1 Kjernen ..... 28*
*4.1.2 Kommunikasjonsmoduler ..... 30*
*4.1.3 Arbeidermoduler ..... 32*
*4.1.4 Oversettelsesmoduler ..... 34*
4.2 ANATOMIEN FOR EN OPPGAVEGJENNOMFØRING I PROTOTYPEN ..... 36

**5 EVALUERING ..... 41**

5.1 UTVIKLINGSMETODE OG PROSESS ..... 41
5.2 INFORMASJONSINNSAMLING ..... 43
5.3 PROTOTYPEN ..... 45
5.4 OPPSUMMERING ..... 46

**6 VIDERE ARBEID ..... 49**

**7 BIBLIOGRAFI......51**

**APPENDIKS A TEGNFORKLARING FOR DATABASEMODELLEN......55**

**APPENDIKS B INSTALLASJONS-VEILEDNING OG KRAV......57**

B.1 INITIALISERE DATABASE......57
STARTE PROTOTYPE OG RMI KLIENT AUTOMATISK......58
B.2 FEILSØKING......59

**APPENDIKS C BESKRIVELSE AV KONFIGURASJONSINNSTILLINGER......61**

CORE......61
LOGGING......63
PLUGINS......64

**APPENDIKS D TESTING MED RMI KLIENTEN OG FERDIGE FORESPØRSELSDOKUMENTER......65**

**APPENDIKS E XML STANDARD OG STØTTEDE OPERASJONER......69**

E.1 REQUEST......69
*E.1.1 Eksempel på request dokument på XML og JSON form......71*
E.2 RESPONSE......72
*E.2.1 Eksempel på response dokument på XML og JSON form......73*
E.3 STØTTEDE OPERASJONER......74

**APPENDIKS F GRENSESNITT FOR KJERNEN......77**

**APPENDIKS G KOMMUNIKASJONS-MODULER......81**

G.1 FORMÅL OG BESKRIVELSE......81
G.2 GRENSESNITT......81

**APPENDIKS H ARBEIDERMODULER......83**

H.1 FORMÅL OG BESKRIVELSE......83
H.2 GRENSESNITT......84

**APPENDIKS I OVERSETTELSESMODULER......85**

I.1 FORMÅL OG BESKRIVELSE......85
I.2 GRENSESNITT......86

**APPENDIKS J JOB-KLASSEN......87**

J.1 FELTER......87

**APPENDIKS K HJELPEKLASSER......91**

K.1 BRUKERDATA (USERDATA)......91
K.2 KONFIGURASJONSINNSTILLINGER (CONFNODE)......93

K.3 DATABASETILKOBLING (DBCONNECTION)......94
K.4 ERROROBJECT......95

**APPENDIKS L STATUSKODER FOR RESULTATER......97**

# FIGURLISTE

FIGUR 3-1: OVERSIKT OVER NY STRUKTUR FOR TILGANGSKONTROLL....18
FIGUR 3-2: NY STRUKTUR FOR FAGDELEN AV DATABASEN....22
FIGUR 4-1: SYSTEMSTRUKTUREN MED INSTANSER AV ARBEIDER- OG OVERSETTERMODUL I EN TRÅD....26
FIGUR 4-2: KOMMUNIKASJON MELLOM ABSTRAKSJONSLAGET OG EN KLIENT....28
FIGUR 4-3: JOBBFLYT GJENNOM ABSTRAKSJONSLAGET....29
FIGUR 4-4: DE ULIKE SKRITTENE I KOMMUNIKASJONSPROSESSEN MELLOM KLIENT OG ABSTRAKSJONSLAG....30
FIGUR 4-5: SKRITT I PROSESSEN TIL EN ARBEIDERMODUL....32
FIGUR 4-6: SKRITT I PROSESSEN TIL EN OVERSETTELSESMODUL....35
FIGUR 4-7: SEKVENSDIAGRAM FOR EN OPPGAVEGJENNOMFØRING MED RMICLIENT OG RMICOMMUNICATOR....40
FIGUR A-1: TEGNFORKLARING FOR DATABASEMODELLEN....55
FIGUR D-2: KJERNEN OG RMI KLIENTEN KJØRER....66
FIGUR D-3: RMI KLIENTEN MED EN FORESPØRSEL KLAR FOR SENDING....67
FIGUR D-4: KJERNEN OG RMI KLIENTEN ETTER AT EN FORESPØRSEL ER SENDT....68

# TABELLISTE

TABELL 2-1: SAMMENLIGNING AV DE VIKTIGSTE ASPEKTENE MELLOM MYSQL OG POSTGRESQL (8)............8
TABELL 2-2: SAMMENLIGNING AV ULIKE LØSNINGER FOR KLIENTKOMMUNIKASJON MED DATABASE............11
TABELL A-1: DEFINISJONER PÅ ELEMENTENE I TEGNFORKLARINGEN. ....................................................55
TABELL C-2: BESKRIVELSE AV KONFIGURASJONSELEMENTENE FOR KJERNEN. ..........................................62
TABELL C-3: BESKRIVELSE AV KONFIGURASJONSELEMENTENE FOR LOGGING..........................................63
TABELL C-4: BESKRIVELSE AV KONFIGURASJONSELEMENTENE FOR MODULER (PLUGINS). ..........................64
TABELL E-5: BESKRIVELSE AV ELEMENTER I "OPSPEC" TAGEN..............................................................70
TABELL E-6: BESKRIVELSE AV ELEMENTENE I "RESPONSESPEC" TAGEN. .................................................72
TABELL E-7: LISTE OVER ALLE STØTTEDE OPERASJONER MED BESKRIVELSE.............................................75
TABELL F-8: KONSTANTER BENYTTET I KJERNEN FOR Å DEFINERE TILSTANDEN. ........................................78
TABELL F-9: UNNTAK SOM KAN FOREKOMME VED KALL TIL QUEUEJOB I CORE KLASSEN. ...........................79
TABELL J-10: TILSTANDSKONSTANTER FOR JOBBER. ..........................................................................88
TABELL J-11: VANLIGE FELTER I JOB-KLASSEN. ..................................................................................89
TABELL L-12: STATUSKODER SOM KAN RETURNERES FRA KJERNEN.......................................................97

# 1 PROBLEMSTILLING

Planlegging og gjennomføring av fag er en omfattende prosess som gjennomføres jevnlig ved instituttene på NTNU. Prosessen krever informasjon om mange ulike aspekter ved undervisningen fra studentantallet, antallet fag, antallet ansatte, budsjetterte kostnader og til evalueringsrapporter fra tidligere fag. Fram til nå har det vært opp til de enkelte ansvarlige på hvert institutt og komme fram til sine egne rutiner på hvordan denne prosessen skal gjennomføres. Noen har da benyttet seg av intern kunnskap til å lage egne databaser som inneholder mye av informasjonen, mens andre har benyttet en mer ad-hoc løsning ved å delegere arbeidet videre til de enkelte foreleserne i hvert fag. Resultatet av dette er at mye informasjon går tapt av ulike årsaker for eksempel fordi forelesere slutter og tar med seg all dokumentasjon om fag han eller hun har holdt, eller at det rett og slett glemmes. Høsten 2009 ble det gjennomført en oppgave med fokus på å forenkle og forbedre planleggingsprosessen. Oppgaven het "Læringsadministrative systemer i norsk høyere utdanning" (1) og gikk ut på å konstruere en ny database med grunnlag i eksisterende læringsplattformer der man kan lagre det meste av informasjon som omfatter både planlegging og gjennomføring av fag. Fordi det allerede eksisterer et stort antall systemer og databaser som inneholder informasjon om alt fra studentantall til rombestilling er det urealistisk, og sannsynligvis upraktisk at ett system skal holde på all informasjonen. Resultatet ble en omfattende databasemodell med fokus på fleksibilitet for å støtte de ulike rutinene for fagplanlegging som hvert institutt har utarbeidet og ideer til videre arbeid om hvordan databasen skulle bli benyttet.

Denne oppgaven tar resultatet fra høsten 2009 videre og består i hovedsak av tre punkter:

- Finne fram ulike løsninger til hvordan man kan benytte databasemodellen fra den tidligere oppgaven i forbindelse med fagplanlegging og gjennomføring.

- Evaluere databasemodellen og utbedre den om nødvendig.
- Konstruere en prototype av dette systemet som kan benyttes til å demonstrere løsningen.

Målet med oppgaven er altså å få fram et felles datalager for informasjonen som behøves i forbindelse med fagplanlegging som både legger til rette for effektiv dokumentasjon av ulike hendelser og aktiviteter og samtidig er fleksibelt og fremtidsrettet til effektivt å kunne benyttes av ulike institutter. Det må også være mulig å benytte dette datalageret i samkjøring med eksisterende systemer og til konstruksjon av nye framtidige løsninger.

# 2 MULIGE LØSNINGER

Denne seksjonen tar for seg noen mulige løsninger for hvordan et system basert på (1) kan konstrueres. Her vil jeg først presentere raskt hvordan noen av instituttene har løst problemet i dag. Så vil jeg gå gjennom konkret hvilke oppgaver en applikasjon som arbeider med informasjonsdatabasen må kunne utføre og til slutt sammenligne ulike fremgangsmåter for å implementere disse oppgavene og velge en konkret løsning.

## 2.1 HVORDAN HAR INSTITUTTENE ORGANISERT INFORMASJONEN I DAG?

Per dags dato finnes det få felles systemer for lagring av faginformasjon ved instituttene som også tar hensyn til planleggingsfasen og elementer som for eksempel utregning av økonomiske ressurser (2). De få systemene som eksisterer er som oftest laget for å løse et konkret problem ved et bestemt institutt. Dette har medført at flere institutter enten har lagret informasjonen i enkeltdokumenter på digital eller analog form eller en kombinasjon av disse, utviklet egne løsninger eller rett og slett fordelt oppgavene til de ulike foreleserne i fagene og latt dem håndtere fagdokumentasjon på sin egen måte. Fordi det ikke har eksistert en felles sentral løsning har dette forårsaket problematikk i forbindelse med planlegging av nye fag fordi informasjon om hvordan faget har blitt driftet tidligere er mangelfull eller borte. Samtidig har det også medført vanskeligheter med å endre på eksiterende fag fordi det sjelden finnes dokumentasjon om endringer som har blitt gjort og der det finnes er den mangelfull. Det har også vært vanskeligere å holde oversikt over hvem som har hatt ansvar for fag på ulike tidspunkt. Fordi instituttene innad har sine egne rutiner kan det også bli vanskelig å utveksle informasjon om fag mellom de ulike instituttene. Det er tydelig at det trengs en bedre måte å organisere, lagre og distribuere fagdokumentasjon.

Et system som adresserer disse problemene må også ta hensyn til store variasjoner innad i de enkelte instituttene på fremgangsmåte for både planlegging og gjennomføring av fag. Ved Instituttet for fysikk her på NTNU har de utviklet en egen databaseløsning for lagring av informasjon om studentassistenter og benytter denne i kombinasjon med budsjetter til å finne ut hvor mange de må ansette hvert semester. Samtidig kjøres fagene omtrent likt fra år til år. Derimot ved Instituttet for byggekunst, form og farge er det opp til den enkelte foreleser å planlegge, ansette studentassistenter og gjennomføre faget på basis av noen spesifikke læremål. Fordi metodene er så forskjellige fra sted til sted er det derfor viktig at den løsningen man velger må være fleksibel nok til å kunne benyttes både på institutter der planleggingen gjøres strukturert (fysikk) og der hvor den gjøres flytende og varierende fra år til år (byggekunst).

En applikasjon må til syvende og sist kunne gi en forbedring av struktur og tilgang i forhold til dagens løsninger. Men for at den skal kunne bli tatt i bruk på mange ulike institutter som har radikalt forksjellige måter å planlegge og gjennomføre fag på er den også nødt til å kunne tilpasses eventuelle spesialtilfeller som kan oppstå. Fagplanlegging og gjennomføring er en meget kompleks prosess som benytter informasjon om alt fra antallet studenter som tar faget til hvor mye penger instituttet (og faget) får tildelt. Det er urealistisk å anta at løsningen skal kunne holde på all denne informasjonen på et sentralisert punkt og derfor bør det også kunne benytte data hentet fra andre systemer.

## 2.2 HVILKE OPPGAVER SKAL SYSTEMET LØSE

Før en kan begynne å lete fram og evaluere ulike løsninger er det viktig å få oversikt over hvilke oppgaver systemet skal kunne utføre. Et av de aller viktigste kriteriene er fleksibilitet. Hvis systemet skal bli tatt i bruk må det være mulig får instituttene å kunne benytte det på sin måte. Systemet må kunne utføre en rekke ulike operasjoner mot databasen, både relativt trivielle ting som uthenting av studentinformasjon for et fag og mer komplekse oppgaver som også kan innebære oppslag i andre systemer. Fordi det er omtrent umulig å spekulere i hvilke andre systemer som også kan ha behov for å sette inn eller hente ut informasjon fra denne løsningen og måten de vil gjøre dette på bør systemet være såpass tilpasningsdyktig at man relativt enkelt kan implementere støtte flere ulike kommunikasjonsprotokoller og dataformater også etter at systemet er tatt i bruk. I tillegg til dette må systemet kunne

håndheve sikkerhetskrav og begrense tilgang til ulike deler av databasen basert på en brukers identifikasjonsinformasjon. Det er i hovedsak fem oppgaver systemet må gjøre:

- Utføre arbeid mot databasen.
- Muliggjøre kommunikasjon med klienter på ulike former og med ulike formater.
- Fasilitere arbeid mot data i databasen.
- Håndheve sikkerhetsbegrensninger og skjule data for brukere basert på tilgangskriterier.
- Være fleksibel nok til relativt enkelt å tilpasse seg små endringer i databasestrukturen.

I tillegg til disse hovedpunktene bør systemet være relativt enkelt å installere og bruke. Det er også en fordel at det kan utføre en form for ressurshåndtering slik at systemet ikke blir overbelastet.

**Utføre arbeid mot databasen**
Dette er hovedoppgaven til systemet. Den innebærer at en klient sender en forespørsel om uthenting eller innsetting av data og systemet utfører denne mot databasen. Resultatet av oppgaven må så returneres til klienten.

**Kommunisere med klienter**
For at systemet skal kunne utføre arbeid trenger det å motta oppgaver. Dette gjøres ved kommunikasjon med klienten. Slik kommunikasjon kan foregå ved hjelp av et stort antall ulike protokoller som blant annet RPC eller http. For at systemet skal være så fleksibelt som mulig bør det også være mulig å benytte flere ulike kommunikasjonsmetoder.

**Fasilitere arbeid mot data i databasen**
Noen av operasjonene kan kreve omfattende kunnskap om strukturen i databasen. Selv om operasjonen i seg selv kanskje er relativt liten er det ofte nødvendig å arbeide med flere ulike tabeller. Denne kompleksiteten børe skjules for klientene slik at de enkelt kan sette inn og hente ut relevant informasjon uten å måtte vite hvilke tabeller man må benytte.

**Håndheve sikkerhetsbegrensninger**
Databasen kan inneholde store mengder informasjon og mye av det kan være konfidensielt eller unødvendig for bestemte brukere å få tilgang til. Det er derfor en fordel at systemet kan begrense tilgang der dette er nødvendig. I tillegg til begrensing av hva som kan hentes ut og settes inn bør det også være

mulig å begrense hvilke operasjoner som kan utføres slik at for eksempel en student ikke har mulighet til å endre detaljer for et fag.

**Fleksibilitet**
Det er naturlig å anta at det allerede finnes flere store systemer i drift på universitetet og instituttene (3). Disse kan være integrer mot hverandre i ulik grad og har sannsynligvis definerte protokoller og kommunikasjonsmetoder som er vanskelig å endre. Systemet bør derfor kunne tilpasses disse eksisterende løsningene slik at integrasjon går raskt og smertefritt.

## 2.3 HVORDAN SKAL DATABASEN IMPLEMENTERES

Databasen er beskrevet som et sett med tabeller der hver tabell har ulike kolonner (felter) med data. I tillegg er det definert flere relasjoner mellom kolonner fra ulike tabeller for å illustrere forhold mellom dataene. Det er tydelig tenkt at databasemodellen skal kjøre på en relasjonsdatabase som PostgreSQL eller MySQL i motsetning til en objekt-orientert database (4). Jeg har begrenset valget mellom relasjonsdatabasesystemer ned til de to fornevnte fordi begge er gratis og relativt enkle å bruke. I tillegg har jeg erfaring med bruk av begge løsningene. Oracle er utelukket for denne oppgaven på grunn av pris og kompleksitet samt fordi jeg har lite erfaring med denne.

**MySQL**
MySQL er det mest velkjente databasesystemet benyttet i dag (5). De aller fleste små-middels store websider benytter denne og det er i hovedsak MySQL som brukes under opplæring på NTNU. I tillegg er det MySQL jeg har mest erfaring med, det har åpen kildekode (OpenSource) og er gratis.

**PostgreSQL**
Dette databasesystemet er en direkte konkurrent til MySQL og er ofte kalt "The world's most advanced open source database" (6). Den har en rekke funksjoner som ikke finnes i MySQL blant annet mer avansert tilgangskontroll og autentiseringsfunksjonalitet for klienter. Det gjør at den alene utfører en større del av oppgavene som det endelige systemet skal utføre, men den er også noe mer kompleks å sette opp.

For databasesystemet er valget avhengig av måten man løser resten av systemet på. Om man ønsker å benytte en egen applikasjon for å tolke forespørsler fra klientene vil dette programmet selv utføre oppgaver som sikkerhetsbegrensinger og kommunikasjonshåndterning og dermed forsvinner

flere av fordelene ved å benytte PostgreSQL framfor MySQL. Om derimot den beste løsningen viser seg og være at klientene skal kommunisere direkte med databasesystemet vil PostgreSQL med sine mer omfattende autentiserings og sikkerhetsfunksjoner være det beste valget.

Om man kun fokuserer på databasesystemets hastighet ser man at denne stort sett er avhengig av hva slags type operasjoner som kjøres. I noen tilfeller er MySQL raskest mens i andre er PostgreSQL raskest. MySQL har i tillegg flere ulike lagringsmotorer og hastigheten avhenger naturlig nok også av hvilken av disse en velger.

En siste relevant forskjell mellom de to systemene er lisensen. Fordi oppgaven ikke vil innebære direkte endringer på kildekoden til verken MySQL eller PostgreSQL er lisensen på disse systemene ikke relevant med mindre den begrenser hvilke typer applikasjoner som kan bruke systemet, noe de ikke gjør. Der derimot lisensen har en følge er ved bruk av klientkode for eventuelle programmer som skal kommunisere med systemet (for eksempel JDBC drivere). MySQL har en lisens som definerer at alle programmer som linker til eller benytter direkte MySQL klientkode må selv være åpen kildekode programmer (etter GPL lisensen (7)). Det er mulig å benytte MySQL klientkoden i lukket-kildekode programmer, men dette krever en betalt lisens fra SUN. PostgreSQL har ikke denne begrensningen og er derfor bedre egnet om man ønsker å holde kildekoden for resten av programmet lukket.

Valget mellom PostgreSQL og MySQL ender i hovedsak opp til et valg mellom preferanser fra utviklerens side. En stor fordel vet at valget står mellom to SQL databaser er at det er relativt enkelt å bytte dem om hverandre. Begge databasesystemene støtter samme syntaks og omtrent samme funksjoner og for en Java programmerer som benytter seg av det forhåndsdefinerte grensesnittet gitt av JDBC drivere blir problemet enda mindre. I beste fall er det faktisk så enkelt som å endre tilkoblingsstrengen som benyttes og Java vil selv ta seg av å benytte den riktige driveren. Dette forutsetter selvfølgelig at driveren følger med og blir lastet inn korrekt.

**Tabell 2-1: Sammenligning av de viktigste aspektene mellom MySQL og PostgreSQL (8).**

| MySQL | PostgreSQL |
|---|---|
| Mulighet for flere ulike lagringsmotorer (MyISAM, InnoDB, CSV++). Kan derfor tilpasses etter spørringene. | En fast lagringsmotor. |
| Hastighet avhenger av valgt lagringsmotor og spørringsformer (enkle, avanserte, samtidighet). | Har flere optimaliseringsalgoritmer implementer. |
| Optimalisering for flere kjerner er under arbeid. | Raskest på systemer med flere kjerner. |
| ACID krav oppfylt in noen lagringsmotorer (bl.a. InnoDB). | Oppfyller ACID (9) krav. |
| Få begrensninger på navnelengder. | Få begrensninger på navnelengder. |
| Støtter de aller fleste nødvendige datatyper. | Støtter flere datatyper (boolean, bruker-definerte). |
| Indekseringsstøtte avhenger av lagringsmotor | Støtter de aller fleste typer indekser. |
| Åpen lisens, men krever at program som benytter klient biblioteket (bl.a. Java driver) er lisensiert GPL | Åpen lisens og åpen kildekode. |
| Offisiell supporttjeneste. | Flere profesjonelle supporttjenester. |
| ”The worlds most popular open source database”. | ”The worlds most powerful open source database”. |

## 2.4 HVORDAN SKAL KLIENTENE ARBEIDE MOT DATABASEN

Det er i hovedsak tre måter klientene kan arbeide mot databasen på:

- Direkte
- Direkte via views
- Via en ekstern applikasjon

**Direkte**

Direkte kommunikasjon vil si at klientene kobler seg direkte mot databasesystemet (MySQL, PostgreSQL, eller tilsvarende). I dette tilfellet vil ingenting bli tilrettelagt for klientene og de må selv vite hvilke tabeller de skal benytte for å sette inn og hente ut informasjon. For direkte kommunikasjon er det også nødvendig at klienten kan arbeide med SQL spørringer og resultater.

Man vil også måtte benytte sikkerhetsfunksjonene i databasesystemet til å begrense tilgang.

Ved direkte kommunikasjon vil PostgreSQL være den mest naturlige løsningen da denne har mer omfattende sikkerhetsfunksjoner og lisensen tillater at eksterne applikasjoner ikke er registrert under GPL lisensen.

**Direkte via views**
Denne kommunikasjonsmetoden er omtrent den samme som direkte kommunikasjon. Klientene er fortsatt nødt til å koble seg rett inn i databasesystemet, men de arbeider ikke mot selve tabellene. Et view (10) er en sammensetning av kolonner (felter) fra flere tabeller konstruert på grunnlag av en spørring. De kan derfor inneholde bearbeidet informasjon fra tabellene noe som gjør det enklere for klientene å jobbe mot databasen. Et view kan også skjule kompleksiteten i databasen ved å slå sammen data fra et stort antall underliggende tabeller. Av samme grunner som ved direkte kommunikasjon vil det også her være fordelaktig å benytte PostgreSQL.

**Via en ekstern applikasjon**
Med en ekstern applikasjon menes et mellomledd som mottar forespørsler fra klientene og utfører disse på databasen. I dette tilfellet behøver ikke klientene vite noe om de underliggende tabellene og viewene noe som gjør uthenting og innsetting mye enklere sett fra klientens side. På andre siden krever dette at den eksterne applikasjonen eller abstraksjonslaget forstår hva klienten ønsker og den må også vite om strukturen på databasen. I dette tilfellet er valget mellom MySQL og PostgreSQL ikke fult så klart med mindre den eksterne applikasjonen skal være lukket kildekode.

**Sammenligning**
Den direkte løsningen er den som krever minst arbeid for utvikleren. Det eneste som behøves er en installasjon av databasesystemet og oppretting av tabeller. Når det er gjort er det opp til klientene å koble seg til og jobbe mot databasen. Av de tre løsningene er denne også den minst fleksible da enhver endring i strukturen på tabeller eller hele databasen vil kreve at klientene også tilpasses. Den største fordelen med denne løsningen er at man får mye gratis. Det er allerede implementer sikkerhetssystemer og ressurshåndtering som er godt testet.

Direkte via views har omtrent de samme fordelene og ulempene som den direkte løsningen. Den største forskjellen ligger i at mye av kompleksiteten kan skjules bak views og dermed behøver ikke klientene vite like mye om den

underliggende strukturen for å kunne benytte databasen. Om strukturen forandres kan man også oppdatere disse viewene slik at klientene fortsatt kan arbeide mot databasen uten å måtte endres.

Det største problemet ved å benytte seg av den direkte eller view løsningen er fleksibilitet. Fordi klientene kobler seg rett til databasesystemet er de nødt til å kjenne kommunikasjonsprotokollen og kunne konstruere SQL spørringer. Dette er et urealistisk krav da mange av de eksisterende systemene kan være store og komplekse og dermed kreve mye tid og penger for å kunne tilpasses til en ny direkte database. Et annet problem er at systemene kan være produsert av tredjeparter og det er dermed lite aktuelt og forventet at produsenten skal endre programmet slik at det passer inn med en ny løsning. Det er heller ikke mulig for klienten for eksempel å sende en forespørsel til systemet i form av et XML dokument eller JSON data. Mange av de eksisterende systemene har allerede definerte grensesnitt som benytter slike dokumenter, men om man kobler seg direkte til databasen vil det ikke være mulig å benytte seg av disse uten omfattende endringer på databasesystemet.

**Tabell 2-2: Sammenligning av ulike løsninger for klientkommunikasjon med database.**

| Løsning | Fordeler | Ulemper |
|---|---|---|
| **Direkte** | • Full kontroll<br>• Kan få ut all informasjon<br>• Tilgang på alle funksjoner støttet av databasesystemet (også SQL funksjoner) | • Krever omfattende kunnskap om databasens struktur<br>• Ved endringer i databasen må også klienter forandres<br>• Klientene må forstå SQL og resultater<br>• Sikkerhetsmekanismer i databasesystemet kan være begrenset |
| **Direkte m/views** | • Komplekse tabeller kan skjules bak views<br>• Omfattende kontroll<br>• Tilgang på det meste av informasjon | • Krever noe kunnskap om databasens struktur<br>• Endringer i databasen medfører i noen tilfeller forandringer i klienten<br>• Sikkerhetsmekanismer i databasesystemet kan være begrenset |
| **Ekstern applikasjon** | • Kan tilpasses klientene istedenfor at klientene må tilpasses databasen<br>• Kan aggregere og bearbeide informasjon fra databasen<br>• Kan returnere resultat i mange ulike formater<br>• Skjuler databasen for klientene<br>• Skjuler forandringer i databasen for klientene<br>• Kan benytte mer omfattende sikkerhetsrutiner<br>• Kan gjøre oppslag i andre systemet for å utfylle data | • Krever utviklingsarbeid (tid og penger)<br>• Introduserer en ny feilkilde i prosessen<br>• Krever mer prosesseringskraft og kanskje dedikert hardware |

Det siste forslaget er en ekstern applikasjon. Et slikt program vil kunne skjule kompleksiteten i databasen og også tilpasses endringer uten at klientene må forandres. Det må kunne håndheve sikkerhetsbegrensninger og sørge for en form for ressurshåndtering da dette ikke lenger kan gjøres kun på databasenivå, og vil også kreve tid i form av planlegging, utviklingsarbeid og testing. Det endelige resultatet derimot vil bli en mye bedre løsning enn de to overnevnte. Et slikt abstraksjonslagt vil kunne støtte flere ulike kommunikasjonsmetoder og dokumentformater slik at flere ulike systemer kan arbeide mot databasen uten å måtte endres. Det vil kunne sørge for detaljert sikkerhetsbegrensning ved hjelp av roller eller tilsvarende og det vil skjule hele databasestrukturen og eventuelle endringer som gjøres underveis fra klientene. Av disse grunnene har jeg derfor valgt denne løsningen.

## 2.5 Hvordan skal klientene kommunisere

I det forrige avsnittet ble det fastslått at den beste løsningen er å benytte et abstraksjonslag for å fasilitere kommunikasjon mellom klienter og data i databasen. Dette innebærer å motta forespørsler fra klienten, utføre disse mot databasen og returnere et resultat. Fordi det allerede finnes flere ulike systemer i drift er det sannsynligvis en fordel å tillate flere ulike former for kommunikasjon[1]. Etter samtale med ITEA (3) som drifter IT systemene ved NTNU har det kommet fram at de aller fleste systemene kommuniserer ved hjelp av en definert XML standard kalt IMS Enterprise (11). Protokollen for overføring av disse ble ikke diskutert, men det skjer sannsynligvis ved at en part kobler seg mot en tjener via HTTP protokollen og avleverer XML dokumentet til den andre parten eller ved at XML dokumentet plasseres i en bestemt mappe som så sjekkes jevnlig for nye data. Min løsning bør derfor minst kunne støtte disse metodene.

Det er vanskelig å vite på forhånd hvilke systemer som vil komme inn i framtiden og hvilke formater disse vil benytte ved utveksling av data. Det er en stor fordel om abstraksjonslaget kan benyttes i lang tid uten å kreve store endringer for å støtte nye løsninger. I tillegg er det en fordel om klientene selv kan spesifisere hvilket format de ønsker på de returnerte dataene og at de kan være forskjellig fra formatet på forespørselen.

For å støtte alle disse scenarioene bør abstraksjonslaget konstrueres modulært. Dette forenkler implementasjon av støtte for nye formater og

[1] Med dette menes former som: HTTP, RPC, direkte filer eller tilsvarende.

kommunikasjonsprotokoller samtidig som det er mulig å fjerne eldre moduler som ikke lenger benyttes. Ved å ha egne moduler som mottar forespørsler og oversetter disse til et internt format kan man skille kommunikasjonsdelen av systemet fra den delen som utfører arbeidet mot databasen. Dette forenkler også utviklingen av nye moduler siden de ikke behøver å tenke på hvordan oppgaven skal utføres.

## 2.6 HVORDAN SKAL OPPGAVER UTFØRES

Det er flere ulike fremgangsmåter for hvordan en oppgave kan utføres i abstraksjonslaget. Den enkleste løsningen er at klientene selv skriver SQL spørringer som bare sendes gjennom til databasen, men det vil også medføre at hele poenget med abstraksjonslaget forsvinner og er derfor ikke realistisk. Den aktuelle løsningen vil innebære at klientene sender sin forespørsel i form av et XML dokument eller liknende ved hjelp av en eller flere ulike kommunikasjonsmetoder.

En annen metode er at klientene kobler seg til abstraksjonslaget ved å benytte standard grensesnitt i klientene. Det vil da eksistere egne moduler som tar seg av kommunikasjonen med de bestemte protokollene. Klienten sender så et XML dokument formatert etter en spesifikk standard som abstraksjonslaget så tolker og utfører oppgaven. Dette er en mer fleksibel løsning fordi den tillater at klientene bruker en tilkoblingsmetode som de selv har støtte for og de trenger derfor ikke utvides for å støtte nye protokoller. Men klientene er nødt til å benytte en standard for XML dokumenter som abstraksjonslaget forstår og dette kan bety at man er nødt til å lage egne oversettere for noen klienter.

En tredje løsning er at man, i tillegg til å støtte flere ulike tilkoblingsmetoder også støtter ulike formater på dokumentet som sendes. Når en forespørsel mottas er modulen ansvarlig for å oversette den til et format som kan benyttes internt i abstraksjonslaget. Når så resultatet er hentet ut av databasen kan det oversettes tilbake til et format som klienten kan forstå. Dette gir deg det beste av to verdener. Man får et internt format som er uavhengig av klientene mens klientene selv kan sende forespørsler på det formatet de helst ønsker.

## 2.7 KONKLUSJON

Etter å ha veid de ulike løsningene mot hverandre har jeg kommet fram til at et modulært abstraksjonslag bygget over en MySQL database er den beste løsningen. MySQL databasesystemet ble valgt fordi jeg har mest erfaring med bruk av dette. Fordi begge alternativene benytter SQL syntaks for spørringene er det også relativt enkelt å bytte til PostgreSQL også ved en senere anledning.

Abstraksjonslagsløsningen er valgt fordi den gir en god kombinasjon av fleksibilitet og brukbarhet. Systemet vil støtte moduler både for kommunikasjon, operasjonsutførelse og oversetting av resultater og dette gjør det mulig raskt å implementere støtte for nye oppgaver og endringer i databasen. Selve kjernen i abstraksjonslaget vil ta seg av ressurshåndtering og formidling av jobber mellom de ulike modulene og behøver derfor ikke vite noe om hvordan databasen ser ut.

Forslaget er basert på et designmønster kalt "Adapter". En adapter oversetter grensesnittet for et system slik at det kan forstås av et annet (12). For abstraksjonslaget er denne definisjonen tatt enda lenger. Ikke bare kan den oversette mange av de vanlige SQL kommandoene som ville kunne utføres direkte mot databasen, den vil også legge til rette for og forenkle ulike oppgaver som skal utføres mot systemet. Istedenfor at klienten selv må vite hvilke SQL spørringer som skal kjøres for å opprette et fag forteller den bare abstraksjonslaget at det er denne oppgaven den ønsker å utføre sammen med informasjonen som skal legges inn. Abstraksjonslaget "tilpasser" denne informasjonen til databasen og putter den inn.

Fordi universitetet har flere ulike systemer allerede på plass som hver holder på ulike typer data kan det i mange tilfeller bli nødvendig å slå opp i disse for å hente informasjon. Dette kan ikke gjøres om klienten er direkte koblet til databasen. Men med et abstraksjonslag i mellom er det mulig ikke bare å gjøre et slikt oppslag, men også å skjule at oppslaget blir gjort. På den måten kan man hente ut informasjon fra flere ulike systemer og samle dette i et felles resultat før det returneres til klienten alt uten at klienten må vite noe om hvor dataene ble hentet fra.

Jeg har valgt å utvikle applikasjonen i Java av flere viktige grunner. I forhold til språk som python, .NET eller C/C++/C# er det java jeg har mest erfaring med og det språket jeg kjenner best. Jeg har også tidligere utviklet applikasjoner i java som benytter nettopp MySQL JDBC driveren og dette gir Java en klar fordel framfor de andre alternativene. Det er også operativsystemuavhengig noe som

fjerner avhengigheten av et bestemt operativsystem og dermed gir større fleksibilitet. Java har en stor brukerbase og det finnes utallige artikler og forum der man kan få assistanse og veiledning om det skulle oppstå utviklingsproblemer. En siste fordel med Java er abstraheringsaspektet JDBC driveren gir. Fordi det benyttes en JDBC driver for kommunikasjon med databasen er resten av programmet relativt uavhengig av hvilken driver som benyttes. Dette gjør det mulig raskt og enkelt å skifte ut driveren om man ønsker å benytte et annet databasesystem enn MySQL. Et krav til databasesystemet er at det må benytte SQL syntaks.

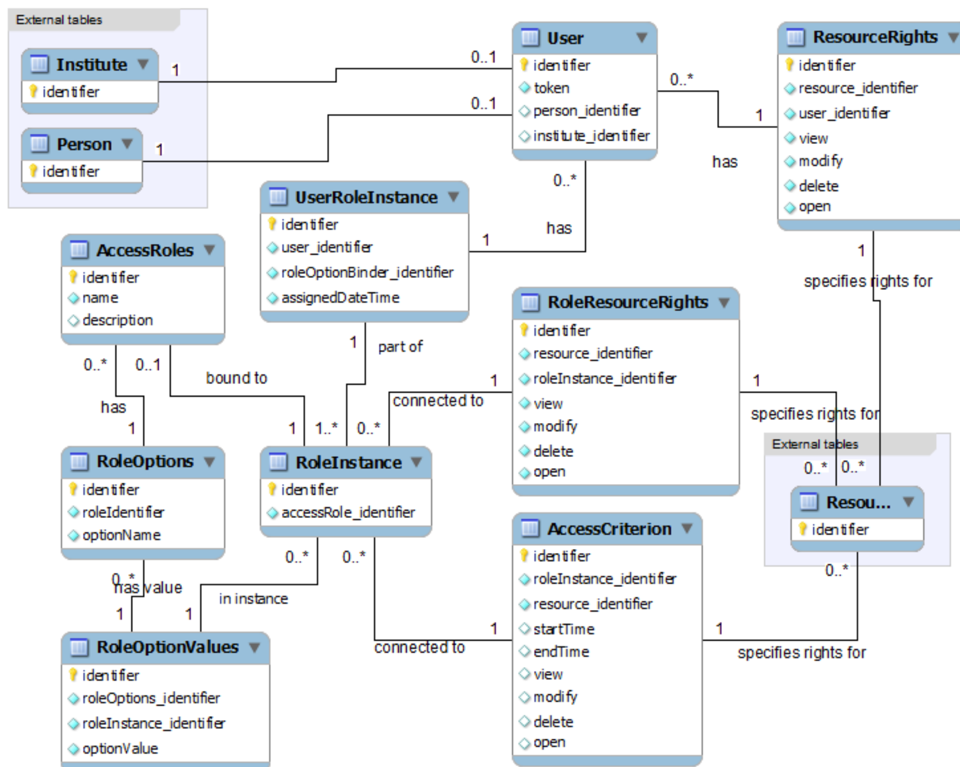
# 3 UTBEDRINGER AV DATABASEN

I forbindelse med videre arbeid med systemet beskrevet i (1) har deler av databasen også blitt endret. En komplett oversikt over databasen finnes i (1). Denne seksjonen vil kun gå gjennom endringer som har blitt gjort i databasestrukturen og hvorfor disse endringene ble gjennomført.

Databasespesifikasjonen slik den er beskrevet i (1) kan ikke benyttes i et ferdig system da få av tabellene inneholder tilstrekkelig kolonner til å holde på nødvendige data. Dette er gjort med vilje for å gjøre det enklere å se strukturen i databasen og relasjoner mellom ulike tabeller. Om databasen skal benyttes i et ferdig system vil det være nødvendig å utvide tabellene med relevante kolonner.

## 3.1 TILGANGSKONTROLL OG SIKKERHET

Tilgangskontrolltabellene beskrevet i den tidligere oppgaven er generelle og meget omfattende og fleksibel. Dette gjør det mulig å foreta endringer på tabeller andre steder i databasen og så kunne definere tilgangsrettigheter uten å endre strukturen for tilgangstabellene. Baksiden ved denne fleksibiliteten er store krav til abstraksjonslaget og dets tolkninger av strukturen. Man må her ta hensyns til et stort antall spesialtilfeller som krever mye ekstra informasjon om handlingen som utføres. Et eksempel er under opprettingen av et fag. Det vil her være en stor fordel om personen som opprettet faget også fikk tilgang til å endre alle aspekter ved det. Slik databasestrukturen er definert i den tidligere oppgaven ville man måtte opprette mange rader i både *ElementRights (* står for User og Group), *TableRights og *ColumnRights. Dette vil medføre mye ekstraarbeid for systemet og det blir vanskeligere for administratorer og raskt finne ut hvem som har tilgang til hva.




**Figur 3-1: Oversikt over ny struktur for tilgangskontroll.**

For å forenkle hele strukturen å gjøre abstraksjonslaget mindre komplekst har jeg gjort store endringer på denne delen av databasen. Istedenfor å operere med rettighetstildeling helt ned på individuelle rader i tabellen i systemet har jeg implementert støtte for rollebasert tilgangskontroll (RBAC (13)). Den største fordelen med et rollebasert system er at det ikke blir nødvendig å definere rettigheter på elementer spesifikt for hver bruker. Isteden definerer man rettighetene for hver rolle og ser om brukeren er tildelt den gitte rollen. Rollebasert tilgangskontroll gjør det også mye enklere å implementere rettighetsbegrensninger i abstraksjonslaget. Istedenfor å måtte sjekke individuelle rettigheter for hvert element som returneres kan en rolle implisitt definere hvilke elementer man skal ha tilgang til. Om for eksempel en studentbruker har rollen "ParticipateInCourse" for et gitt fag vil denne personen automatisk få tilgang til alle meldinger og relevant informasjon om faget. Spesifikke rettigheter kan fortsett gjelde for ressurser og disse må det tas hensyn til ved uthenting av informasjon. Et annet spesifikt eksempel er ved

opprettelse av nye ressurser. I det gamle systemet var det her nødvendig å definere tillatelser for innsetting av elementer i tabellen **Resource** og så tilgang til hver og en av kolonnene i **Resource**. Med det nye systemet holder det at en bruker får rollen som ressursoppretter (eller tilsvarende) og abstraksjonslaget kan vite med et raskt oppslag om personen har tillatelse til å opprette ressurser eller ikke.

I et rollebasert tilgangssystem deles det ut privilegier basert på en eller flere roller en person har fått tildelt. Disse rollene er uavhengig av og må ikke forveksles med rollene definert for et Person element (roller som Student, Foreleser og liknende). Rettighetsrollene vil ofte gjenspeile personrollene, men dette behøver ikke være tilfellet. En person som har rollen student kan også ha tilgang til oppretting av fag eller ressurser mens en foreleser i et fag kan være en student i et annet.

Tabellen User har fått en litt annen funksjon i den nye modellen. En "User" er nå en person eller applikasjon som har tilgang til systemet. Feltet "token" er ment å holde på en unik identifikator for brukeren. Fordi det ikke er kjent hvilken type autentisering som skal benyttes (se (1)) er dette feltet ikke fullstendig definert med datatype (tekst, heltall eller liknende). En mulig løsning kan være å benytte seg av en hashverdi basert på et brukernavn, passord og en såkalt saltverdi (14). Poenget med en slik verdi er å forhindre en hacker eller tilsvarende som har fått tak i tokenen fra å reversere hashverdien og komme fram til brukernavnet og passordet som ble benyttet ved bruk av teknikker som rainbow tables (15).

I (1) var det også definert en mulighet for å binde en bruker til enten en person eller et institutt som allerede var lagret i systemet. Dette er fortsatt mulig ved å knytte instituttelementer eller personelementer til et brukerelement i tabellen **User**. Det er ikke et krav at brukerkontoen knyttes slik, men det gjør det mulig for abstraksjonslaget å benytte eierrettigheter for ressurser.

Rollene en bruker kan ha er definert i tabellen **AccessRoles**. Den inneholder navn og beskrivelse samt en identifikator som brukes i relasjoner. En rolle kan ha et sett med attributter som brukes til ytterligere å spesifisere rollen. Navnet på disse attributtene er lagret i tabellen **RoleOptions** mens verdien ligger i **RoleOptionValues**. Grunnen at disse er adskilt er for enklere å kunne definere hvilke attributter en rolle skal kunne ha uten å måtte slå opp alle verdiene den kan ha definert. Tabellen som binder roller og rolleattributter sammen heter **RoleInstance**. En *rolleinstans* er da en rolle og et sett med attributtverdier som

sammen utgjør et unikt element.  Ved å benytte denne strukturen er det mulig å binde flere brukere til for eksempel "fagdeltaker" for et gitt fag uten å måtte opprette flere like roller med samme attributtverdier. Den gjør det også mulig å spesifisere tilgangsrettigheter til ulike ressurser direkte for en rolleinstans slik at brukere som er medlem automatisk får disse rettighetene. Datamengden som databasen må holde på minskes også fordi systemet ikke lenger trenger å holde på rettighetene for hver ressurs for hver bruker. Det blir også enklere for abstraksjonslaget å definere rettigheter for en gitt bruker som skal meldes inn i en rolle siden det kun er nødvendig å koble brukeren rett til instansen og rollene er ferdig definert.

Et av kravene definert i (1) var også at systemet automatisk skulle kunne begrense tilgang til ressurser både permanent og innenfor gitte tidsrom. Dette gjøres mulig av tabellene **RoleResourceRights**, **ResourceRights**, og **AccessCriterion**. **RoleResourceRights** og **AccessCriterion** er koblet til en instans av en rolle mens **ResourceRights** er koblet rett til brukeren. Rettigheter tilegnes ressurser basert på disse reglene:

- Rettigheter definert i **ResourceRight** overskriver alle andre rettigheter.
- Rettigheter definert i **AccessCriterion** overskriver rettigheter definert i **RoleResourceRights** så sant de er innenfor det spesifiserte tidsrommet.
- Rettigheter definert i **RoleResourceRights** overskriver standardrettigheter.
- Standardrettigheter trer i kraft når ingen andre rettigheter er definert og gir ingen tilgang til ressursen.
- Eieren av en ressurs vil alltid ha full tilgang til ressursen uansett andre rettigheter.

Ved å benytte rollebasert tilgangskontroll gjøres abstraksjonslagets jobb mye enklere. For å utføre en operasjon behøves det kun noen få databasekall for å finne ut om klienten har de tilstrekkelige rettighetene. Hvis ja vil operasjonen kunne fortsette, hvis nei vil den bli avbrutt.

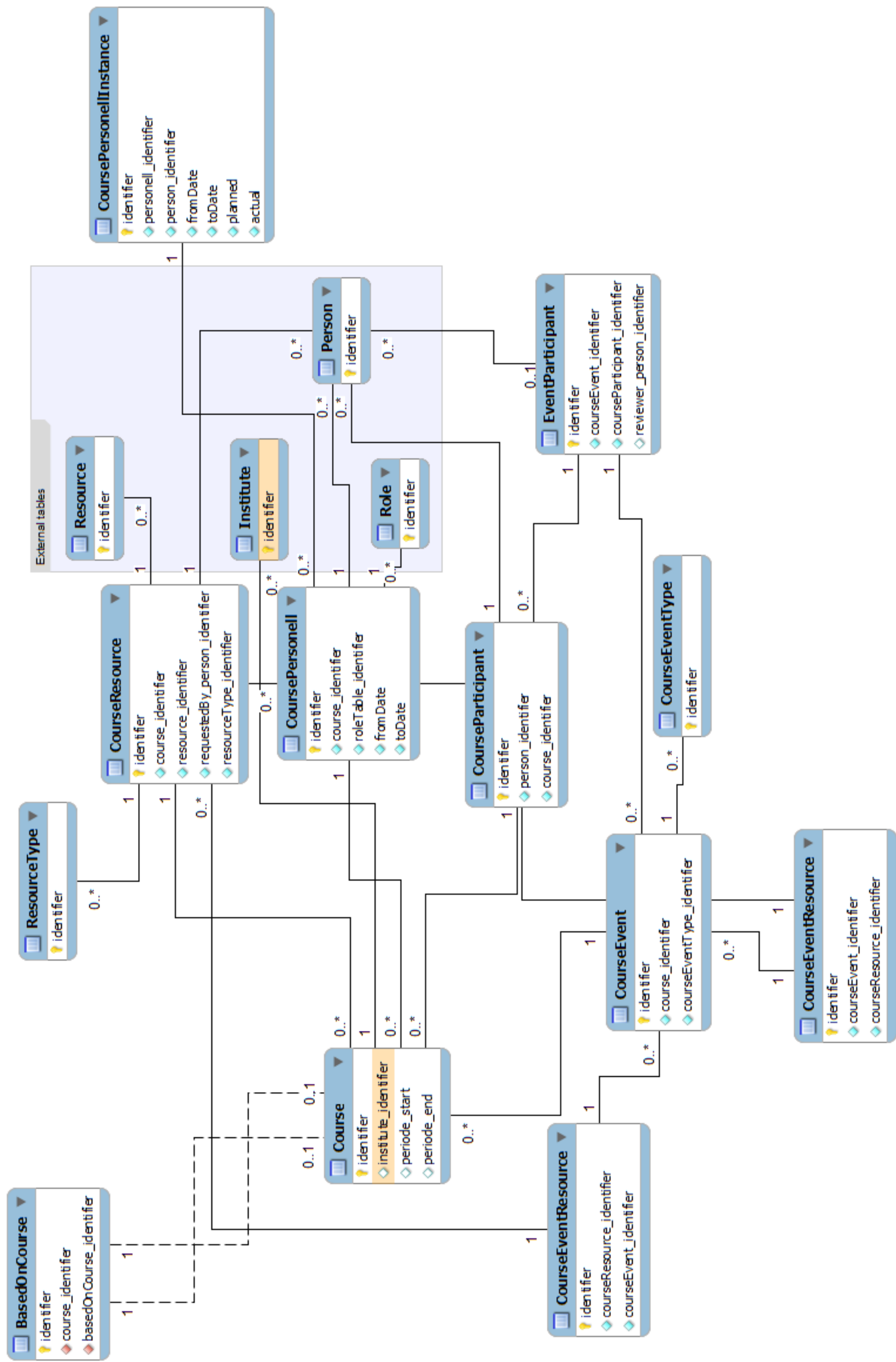
## 3.2 FAG

Etter samtale med ansatte på instituttet (16) har det kommet fram at fagdelen av databasen var noe mangelfull. Det største problemet lå i lagring av informasjon om roller knyttet til et fag. For å få optimalt utbytte av

informasjonen er det nødvendig for instituttet å vite ikke bare hvem som har innehatt en rolle under gjennomføring, men også hvilke roller som må fylles for et fag og hvem som fylte disse for gitte tidsrom. I noen tilfeller vil det også være mulig for en person å avbryte sin rolle underveis for så å bli erstattet av en annen. Her kan man da ende opp med et fag som krever **en** foreleser under planlegging, men som i praksis endte opp med å ha **tre** på ulike tidspunkt gjennom utføringen av faget. For å kunne lagre denne informasjonen har det blitt gjort forandringer på tabellen **CoursePersonell**. Det har også blitt lagt til en ny tabell kalt **CoursePersonellInstance**.

Endringen i CoursePersonell er relativt enkel. Det er lagt til et felt kalt *roleTable_identifier*. Dette feltet peker på identifikatoren til en rad i tabellen **Role** fra persondelen. På den måten vil CoursePersonell definere en rolle som man ønsker skal fylles for et fag. I tillegg har feltet *person_identifier* blitt fjernet fordi det ikke lenger kan knyttes personer direkte til rollen. Isteden blir dette gjort i den nye tabellen **CoursePersonellInstance**. Tidligere var man også nødt til å se gjennom alle rolletabeller etter roller med den ønskede identifikatoren fra **CoursePersonell**, men nå vet man i tillegg navnet på tabellen og dermed spares abstraksjonslaget for en god del arbeid.

**CoursePersonellInstance** tabellen holder på informasjon om en "instans" av en rolle for et fag. Med dette menes en tilknytning mellom en person og en ønsket fagrolle. Tabellen vil også inneholde informasjon om når personen skal eller skulle tiltråd rollen og når han eller hun skulle avstå (*fromDate* og *toDate* feltene respektivt). I tillegg til dette inneholder tabellen to felt kalt *planned* og *actual*. Meningen med disse er at de skal definere rolleinstansen som en planlagt instans eller som en faktisk gjennomført instans. Det kan for eksempel planlegges at en person skal fylle rollen som foreleser for et fag gjennom hele fagets løpetid (si et semester), men i praksis kan det hende personen ikke har mulighet til dette og det derfor settes inn en ekstra foreleser for å ta over rollen innenfor gitte perioder. Dette scenarioet lagres i systemet ved at det opprettes en rad for den planlagte gjennomføringen der personen er satt opp fra fagets start til slutt. I denne raden vil feltet *planned* ha verdien 1 (tolkes som boolsk "sann" eller "true") for å definere at dette var en planlagt instans. Senere vil det opprettes en eller flere instanser der denne personen er satt som foreleser for et tidsrom, mens det opprettes andre instanser der en annen person er satt som foreleser. I disse nye radene vil feltet *actual* ha verdien 1 for å indikere at det var dette som faktisk skjedde. Dermed har systemet lagret både planlagt (planned) og gjennomført (actual) informasjon slik at dette kan benyttes ved videre planlegging og eventuelt til å kalkulere lønnsutbetalinger.




**Figur 3-2: Ny struktur for fagdelen av databasen.**

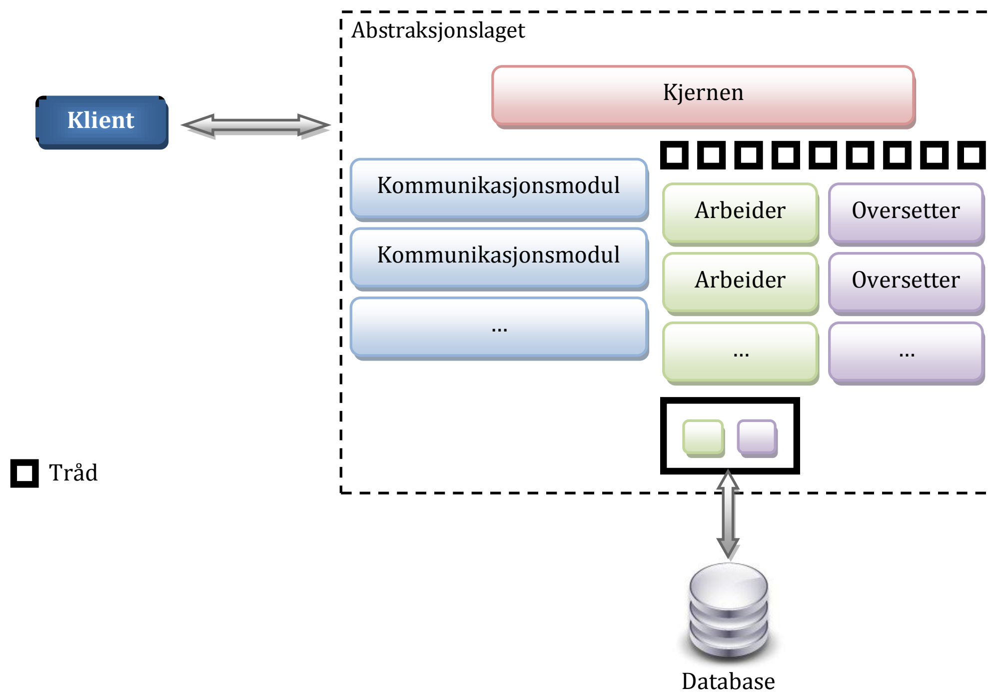
## 3.3 RESSURSER

For ressurser var modellen tidligere definert slik at man måtte slå opp i **Resource** tabellen for å finne ressurstypetabellen og elementet i denne som beskrev ressursen. Dette er endret for å kunne raskere slå opp ressurser basert på en bestemt type. Ved å inkludere et felt i hver ressurstypetabell (bilde, presentasjon, bok osv…) som unikt identifiserer ressurselementet fra tabellen **Resource** blir det mulig å gå direkte fra en slik tabell og tilbake til **Resource** tabellen. Dette gjør det mye enklere å gjøre oppslag som for eksempel ”finn alle bilderessurser” fordi man kan gå begge veier. Tidligere var man nødt til å lete gjennom alle ressursene og kompilere en liste med ressurser som tilfredstilte denne typen.

# 4 ABSTRAKSJONSLAGET

Abstraksjonslaget er programmet som skjuler databasen fra eksterne klienter som ønsker å hente eller sette inn informasjon. Det er utviklet en prototype av dette programmet skrevet i Java med noen tilhørende moduler som demonstrerer hvordan det er tenkt systemet skal virke. I tillegg er det laget et klientprogram som kan kommunisere med systemet både via XML og JSON dokumenter. Denne seksjonen vil først gå detaljert gjennom strukturen på abstraksjonslaget og prototypen i detalj for så å gå gjennom den anatomiske strukturen på en oppgavegjennomføring med metodenavn og kall til de ulike grensesnittene og klassene i systemet. Mer informasjon om grensesnittene for kjernen og de enkelte modulene finnes i appendiksene H-J.

## 4.1 OVERORDNET STRUKTUR



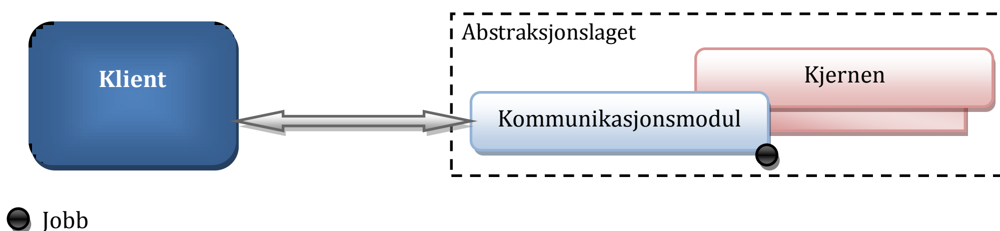

**Figur 4-1: Systemstrukturen med instanser av arbeider- og oversettermodul i en tråd.**

Abstraksjonslaget er basert på en liten fast kjerne ansvarlig for ressurs- og modulhåndtering og et sett med moduler som implementerer ulik funksjonalitet rundt denne (Figur 4-1). Modulene er ansvarlig for å hente inn forespørsler fra klienter, utføre disse mot databasen og returnere et resultat klienten kan forstå. Kjernen sørger for å tilby modulene tilgangspunkter der de kan kommunisere med hverandre og administrerer ressurser for disse. Abstraksjonslaget kan derfor deles inn i fire hoveddeler:

- Kjernen
  Ansvarlig for oppstart og avslutning, lasting av moduler og administrasjon av oppgaver
- Kommunikasjonsmoduler
  Kommuniserer med klientene på ulike måter
- Arbeidermoduler
  Utfører de faktiske oppgavene mot databasen
- Oversettelsesmoduler
  Oversetter resultatet fra arbeidermodulene til formatet klienten har bedt om (om nødvendig).

Figur 4-2 viser en oversikt over hvordan en klient vil kommunisere med abstraksjonslaget. Når en klient ønsker å hente ut eller sette inn informasjon i databasen kontakter den abstraksjonslaget via en av kommunikasjonsmodulene. Så sender den forespørselen sin i et format kommunikasjonsmodulen kan gjenkjenne (XML, JSON eller tilsvarende). Kommunikasjonsmodulen konstruerer da et objekt for å holde på forespørselen kalt en ”jobb” (eller ”Job”). Jobben sendes gjennom systemet og utføres hvis mulig[1]. Til slutt oversettes resultatet til formatet klienten ønsker og det returneres.

[1] Fordi det er arbeidermodulene som utfører oppgaven kan det hende en klient ønsker å utføre en jobb som ingen arbeider kan gjøre. I slike tilfeller returneres en feilmelding til klienten.




**Figur 4-2: Kommunikasjon mellom abstraksjonslaget og en klient.**

De følgende seksjonene vil beskrive de ulike delene av systemet i større detalj.

### 4.1.1 KJERNEN

Kjernen har følgende oppgaver:

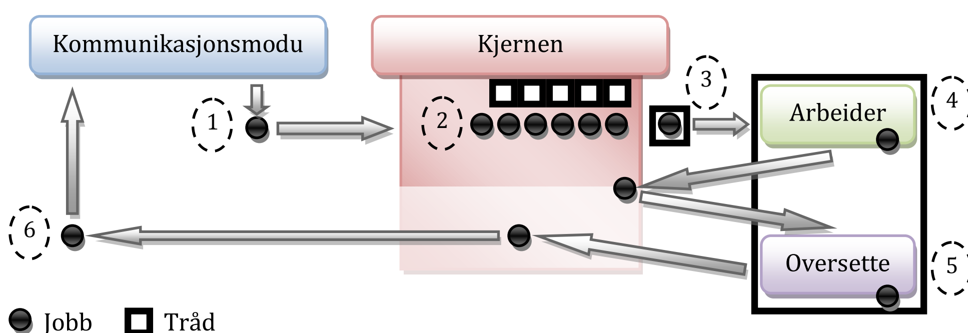
- Innlasting av konfigurasjonsinnstillinger
- Initialisering av systemet (ymse oppgaver i forbindelse med oppstart)
- Innlasting av moduler
- Avslutning av systemet
  Kjernen informerer alle modulene om at systemet skal avsluttes og modulene må selv avslutte sine interne prosesser.

Kjernen er rammeverket som binder modulene sammen. Den er ansvarlig for å laste inn modulene ved oppstart samt fasilitere kommunikasjon mellom modulene og mellom modulene og kjernen. For å konfigurere kjernen brukes det en XML fil kalt config.xml. Denne må ligge i samme mappe som applikasjonen for å kunne leses og hentes inn når programmet startes opp. En detaljert beskrivelse av konfigurasjonsfilen finnes i Appendiks C.

Når kjernen startes opp lastes modulene inn en etter en og registreres. Modulene behøver ingen kunnskap om hverandre for å kunne kjøre fordi kjernen tar seg av all intermodulær kommunikasjon. Hver modul arbeider med et kjent grensesnitt mot kjernen og kan arbeide til en viss grad som om de var alene. Denne måten å strukturere systemet på kalles ofte en bro (17) . Fordelen med denne strukturen er at kompleksiteten internt i kjernen er skjult for modulene og kan variere uavhengig av disse. Om det oppdages feil eller om kjernen skal utbedres vil de eksisterende modulene fremdeles kunne arbeide. Tilsvarende vil også kjernen fortsatt virke selv om modulene blir forandret fordi de også har definert en bestemt bro (grensesnitt) som kan benyttes. En

beskrivelse av grensesnittet for kjernen finnes i Appendiks F mens grensesnittet for de ulike modulene finnes i appendiks H-J.

I tillegg til oppstart og avslutting av moduler tar kjernen seg også av ressurshåndtering. For at systemet ikke skal løpe løpsk med ressurser kan man definere hvor mange tråder systemet skal kunne benytte til utføring av oppgaver. Jo flere tråder jo flere oppgaver kan utføres samtidig[2]. I tillegg til dette kan man også begrense antallet samtidige tilkoblinger til databasen. Disse to innstillingene gjør det mulig å spesifisere hvor stor del av ressursene abstraksjonslaget skal benytte.




**Figur 4-3: Jobbflyt gjennom abstraksjonslaget.**

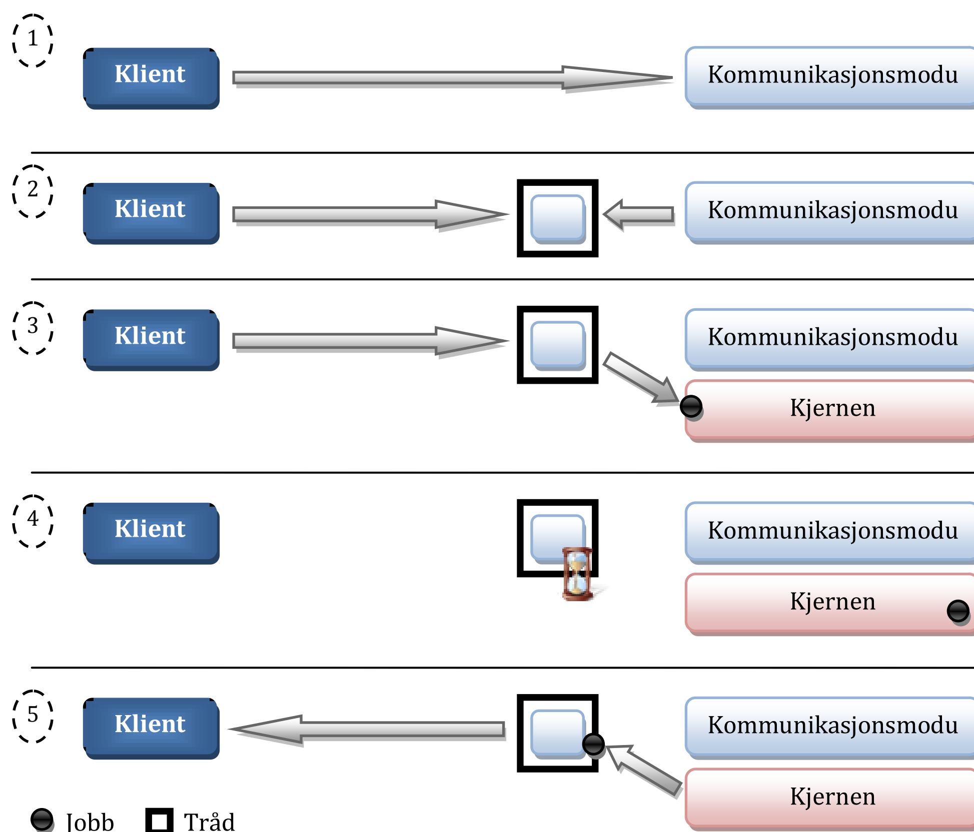
Som beskrevet i seksjon 4.1 opprettes det et jobb-objekt som beskriver oppgaven en klient ønsker utført. Figur 4-3 viser hvordan en jobb beveger seg gjennom systemet. Kjernen har under oppstart opprettet et visst antall arbeidertråder som venter på å få utføre oppgavene som komme rinn. For å fordele oppgaver utover de aktuelle trådene er det implementert en relativt enkel FIFO[3] køstruktur. Når en oppgave kommer inn i systemet via en kommunikasjonsmodul (skritt 1) plasseres den i en kø i påvente av en ledig arbeidstråd (skritt 2). Når en tråd blir ledig får den tildelt en oppgave før den går videre til neste skritt der den går inn i en ny FIFO kø for å vente på en ledig databasetilkobling (skritt 3, databasetilkoblingskøen vises ikke). Når en databasetilkobling blir ledig sendes jobben til en arbeidermodul som utfører oppgaven i den tildelte tråden (skritt 4) før den returneres til kjernen. Det siste

[2] Begrensninger i trådhåndteringen i det underliggende operativsystemet kan medføre at jobber som har fått tildelt hver sin tråd ikke nødvendigvis kjører samtidig.
[3] FIFO: First In First Out er en rettferdig køstruktur der den som kom først inn I køen slippes først ut på samme måte som enhver vanlig kø virker.

som skjer før resultatet sendes tilbake til klienten (skritt 6) er en oversetting (skritt 5) til formatet klienten ba om hvis mulig. Når oppgaven er gjort returneres tråden til kjernen og venter til den får en ny oppgave.

### *4.1.2 Kommunikasjonsmoduler*

Kommunikasjonsmoduler har ansvaret for å implementere kommunikasjonsprotokoller mellom kjernen og klientene. I vanlige tilfeller har en slik modul ansvaret for å kommunisere via en bestemt protokoll. Man kan for eksempel ha moduler som støtter http protokollen over TCP/IP mens en annen kan støtte RPC (eller Java RMI) protokollen. Ved å skille ut denne funksjonaliteten i egne moduler er det mulig å utvide kjernen til å støtte et stort antall kommunikasjonsformer og dermed tilrettelegge for en mengde ulike klienter. For flere detaljer om hvordan kommunikasjonsmoduler er implementert i prototypen se Appendiks G.




**Figur 4-4: De ulike skrittene i kommunikasjonsprosessen mellom klient og abstraksjonslag.**

Når en klient ønsker å utføre en oppgave mot systemet kontakter den en av kommunikasjonsmodulene (skritt 1). Så snart modulen blir oppmerksom på dette opprettes det en ny tråd (skritt 2). Denne tråden blir tildelt ansvaret for all videre kommunikasjon med klienten. Grunnen til at man oppretter en ny tråd er for å kunne avvente et resultat fra kjernen. Dette resultatet kan ta lang tid å produsere og det kan hende jobben blir satt i kø. For at modulen ikke skal måtte låses mens den venter på dette lages det en egen tråd kun for denne klienten. Denne tråden kan vente mens resten av systemet får fortsette. Det er ikke dermed sagt at alle kommunikasjonsmoduler er nødt til å implementere denne formen for avventing. Om en modul ønsker å benytte en helt annen metode står den fritt til å gjøre det. Når tråden er opprettet mottar den oppgaven fra klienten med alle data som trengs for å utføre den. Det opprettes et jobb-objekt som så sendes til kjernen (skritt 3). Mens kjernen arbeider (eller retter sagt modulene i kjernen) venter tråden (skritt 4). Når jobben er gjort og resultatet er klart eller ved en feil returneres jobb-objektetet til tråden. Den sender så resultatet tilbake til klienten før den avslutter seg selv (skritt 5).

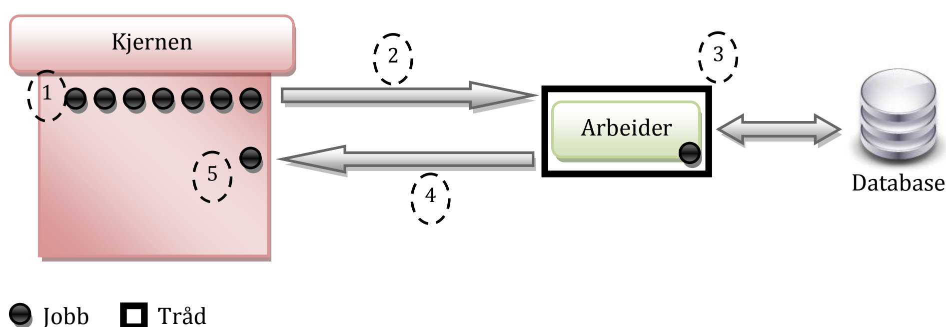
Klientkommunikasjonsmodulene er basert på en form for adapter. En adapter oversetter et grensesnitt til et annet slik at det kan benyttes av ulike klienter (12). Når en klient kobler seg til abstraksjonslaget sørger kommunikasjonsmodulen for at kjernen kan snakke med klienten. Dette gjelder både for kommunikasjonsprotokollen som benyttes og formatet på forespørselen (XML, JSON og liknende). Fordelen med å ha slike adaptere for kommunikasjon er åpenbar. Den tillater at en mengde ulike klienter kan kommunisere med abstraksjonslaget og databasen uten å måtte endres for å støtte en bestemt protokoll. Isteden er det abstraksjonslaget selv som tilpasses til klientene. Dermed behøver man ikke å gjøre forandringer på eksisterende systemer som både kan være store og komplekse.

I prototypeapplikasjonen er det utviklet en ekstern kommunikasjonsmodul. Denne implementerer støtte for Java RMI kommunikasjon via XML og JSON dokumenter. Disse dokumentene følger en enkel standard laget kun for dette systemet som det finnes mer informasjon om i Appendiks E. I tillegg er det utviklet en liten applikasjon som kan sende forespørsler til denne modulen. For flere detaljer om dette se Appendiks D.

### 4.1.3 ARBEIDERMODULER

Kjernen i seg selv er kun en organisator og fasilitator for de ulike modulene. Den kan ikke motta forespørsler direkte fra klienten og den kan heller ikke utføre disse forespørslene mot databasen. Den første oppgaven er kommunikasjonsmodulene ansvarlig for mens den andre faller til arbeidermodulene. En arbeidermodul er en liten bit kode som kan utføre en eller flere oppgaver mot databasen. Når en klient ønsker å hente ut eller sette inn informasjon i databasen sender den en forespørsel til en kommunikasjonsmodul. Denne forespørselen inneholder all informasjon en arbeidermodul trenger for å utføre oppgaven. Arbeidermodulen tar imot forespørselen fra kjernen i form av et jobb-objekt (se seksjon 4.1), utfører den og returnerer et resultat til jobb-objektet som så kan returneres til kjernen for videre prosessering (se oversettelsesmoduler i seksjon 4.1.4).

Når abstraksjonslaget starter opp lastes arbeidermodulene inn. For at kjernen skal kunne vite om hvilke operasjoner som kan utføres mot databasen vil hver arbeidermodul fortelle hvilke konkrete oppgaver den kan utføre. For eksempel vil en arbeidermodul som omhandler fag og fagoperasjoner kunne fortelle kjernen at den kan utføre operasjonene ”getAllCourses”, ”addCourse”, ”removeCourse” osv… Når en klient skal utføre en operasjon mot databasen sendes en forespørsel der operasjonen identifiseres. Alt kjernen trenger å gjøre da er å finne ut om en arbeidermodul har meldt fra om at den kan utføre den gitte operasjonen og videresende jobben til denne (etter avventing av tilgjengelige ressurser).




**Figur 4-5: Skritt i prosessen til en arbeidermodul.**

Når en jobb[4] har kommet inn i systemet plasseres den i en kø i påvente av en ledig prosesseringstråd. Så snart en tråd blir ledig blir jobben allokert til den og arbeidermodulen tar over[5]. Arbeideren har nå full kontroll over jobben. For de aller fleste oppgaver er det ikke nødvendig for arbeideren å gjøre annet enn å sjekke at klienten har tilstrekkelig rettigheter og så utføre oppgaven. Hvis den lykkes legges resultatet i jobb-objektet og arbeideren returnerer. Hvis ikke lagres en feilkode og feilmelding i jobben som så kan returneres til klienten. Figur 4-5 viser skrittene i prosessen fra en jobb plasseres i en kø til den er ferdig utført eller har feilet av en eller annen grunn. I skritt 1 plasseres et jobb-objekt i køen. Dette skjer først etter at kjernen har sjekket at det finnes en arbeidermodul som kan utføre den spesifikke oppgaven. Hvis for eksempel en klient sender en forespørsel om at systemet skal utføre oppgaven "vaskOpp" og det ikke finnes en arbeider som har registrert at den kan gjøre denne oppgaven vil jobben aldri bli plassert i køen. Køen inneholder derfor kun oppgaver som man vet systemet kan utføre.

Når en arbeidertråd blir ledig blir den tildelt en jobb fra køen og arbeidermodulen som kan utføre den gitte oppgaven blir aktivert (skritt 2). Arbeideren kan så benytte seg av en databasetilkobling som følger med jobben til å utføre oppgaven som er definert. Arbeideren er ikke begrenset til kun å jobbe mot den definerte databasen og siden den har en egen tråd og full kontroll kan den om nødvendig koble seg til eksterne kilder eller gjøre andre operasjoner for å få jobben gjort. Det er med andre ord ikke lagt noen spesifikke begrensninger på hva arbeideren kan gjøre og det gjør det mulig å slå opp i andre databaser eller kontakte andre systemer om det kreves. Etter at informasjonen er hentet ut eller satt inn returnerer arbeideren jobb-objektet tilbake til kjernen (skritt 4) hvor resultatet kan bearbeides ytterligere om nødvendig før det returneres til kommunikasjonsmodulen og til slutt klienten.

For å demonstrere hvordan arbeiderne virker er det laget et sett med enkle arbeidermoduler i prototypeapplikasjonen. Disse kan benyttes ved å kjøre det eksterne programmet som kommuniserer med abstraksjonslaget via Java RMI. Mer informasjon om hvordan dette kan testes finnes i Appendiks D.

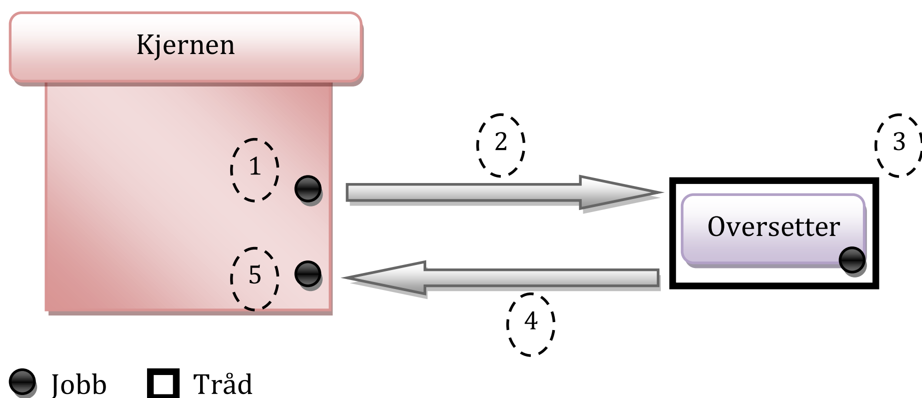
[4] Les jobb-objekt.

[5] Teknisk sett plasseres jobben i en ny kø i påvente av en ledig databasetilkobling, men dette er utelatt fra denne beskrivelsen for enkelhets skyld.

### *4.1.4 OVERSETTELSESMODULER*

Så langt støtter abstraksjonslaget ulike kommunikasjonsprotokoller via kommunikasjonsmodulene og en teoretisk stor mengde ulike operasjoner via arbeidermodulene, men for å sørge for enda større fleksibilitet gis klienten i tillegg til dette også muligheten til å definere returformatet i forespørselen. Internt i systemet benyttes JSON til å lagre resultatet av jobben, men når det skal returneres er det langt fra sikkert at klienten støtter det formatet. En oversettelsesmodul løser dette problemet. Når en klient ønsker å få resultatet på et bestemt format definerer den det i forespørselen. Så lenge det finnes en oversettelsesmodul som kan oversette til dette formatet vil resultatet bli returnert korrekt. Det er også mulig for kommunikasjonsmodulen selv (som er ansvarlig for å bygge jobb-objektet til å begynne med) og definere et annet format dersom klienten ikke kjenner til denne muligheten. Om klienten for eksempel er et eksternt system som kommuniserer via en bestemt form for XML dokument (eksempelvis IMS Enterprise eller liknende) kan kommunikasjonsmodulen oppdage dette og inkludere en oversettelse tilbake til IMS Enterprise formatet når jobben er gjort. En oversettelsesmodul er med andre ord ikke nødt til å oversette mellom to fullstendig ulike formater som JSON og XML, men kan også benyttes til å konvertere et XML format til et annet.

Under oppstart av abstraksjonslaget lastes oversetterne inn og (på samme måte som med arbeidermodulene) de informerer kjernen om hvilket format de kan oversette til. Alle oversetterne må kunne lese internformatet (JSON). Når kjernen mottar et jobb-objekt sjekker den om det finnes en oversetter som kan oversette resultatet til det ønskede formatet. Hvis ikke vil jobb-objektet returneres tilbake til kommunikasjonsmodulen umiddelbart med en feilmelding. Jobben tar dermed ikke opp flere ressurser enn nødvendig om det viser seg at den ikke kan gjennomføres. Dette skrittet foretas rett etter sjekken om det finnes en arbeidermodul som kan utføre jobben (se seksjon 4.1.3)




**Figur 4-6: Skritt i prosessen til en oversettelsesmodul.**

Figur 4-6 viser prosessen til en jobb fra den returneres fra en arbeider til den er klar for avsending til klienten. Etter at en jobb er blitt gjennomført i en arbeidermodul returneres den tilbake til kjernen. Her sjekkes det om jobbens resultat skal oversettes til et annet format. Hvis dette ikke er tilfellet vil resultatet returneres til klienten som en JSON streng(18). Om det derimot er definert et ønsket format for resultatet vil den relevante oversetteren hentes inn og jobb-objektet sendes dit (skritt 1 og 2). Når en oversetter mottar et jobb-objekt ser den på det interne resultatet og utfører oversettelsen (skritt 3). Det endelige resultatet lagres separat i jobb-objektet og det returneres til kjernen (skritt 4 og 5). Som med arbeidermodulene kan også oversetteren gjøre mer komplekse operasjoner enn en direkte oversettelse. Den naive måten å konvertere data på er å gjøre det manuelt, skritt for skritt, men om man vil oversette til et bestemt XML format kan dette gjøres ved å benytte XSLT(19) oversettelsesbiblioteker (java har flere) som kan lastes inn i abstraksjonslaget sammen med oversettelsesmodulen. Fordi det ikke er lagt begrensninger på hva oversetteren kan gjøre kan den til og med kontakte online tjenester og få resultatet oversatt på den måten om nødvendig.

I prototypeapplikasjonen er det inkludert to oversettelsesmoduler som viser hvordan disse kan være implementert. Den ene oversetter til XML mens den andre konverterer til kommadelte verdier (CSV). For flere detaljer se Appendiks I.

## 4.2 ANATOMIEN FOR EN OPPGAVEGJENNOMFØRING I PROTOTYPEN

Denne seksjonen vil forklare nøyaktig hva som skjer fra en klient kobler seg til og fram til den får tilbake et resultat i prototypen. I dette eksempelet er RMI kommunikasjonsmodulen benyttet sammen med RMIClient applikasjonen. Det er sendt en forespørsel som en XML fil og sekvensen er beskrevet i Figur 4-7.

Det første som skjer er at RMIClient programmet slår opp i Javas RMI registry og finner fram til den nødvendige klassen i abstraksjonslaget (prototypen). Når dette er gjort sender den XML dokumentet via metoden xmlRequest i RMICommunicator klassen.

```
public String xmlRequest(String xmlDoc, String token) throws
RemoteException
```

*Utfører en oppgave mot databasen basert på et XML dokument. Token parameteren ignoreres fordi tokenet er en del av XML dokumentet.*

Før "xmlRequest" kan behandle forespørselen kontakter den kjernen via metoden "newConnection" for å sjekke at det er ledige tilkoblinger tilgjengelig. Hvis det ikke er tilfellet returneres en feilkode til klienten og prosesseringen avsluttes. Om det derimot er ledige tilkoblinger oversetter den forespørselen til JSON format ved hjelp av metoder i JSON biblioteket og sender den videre til executeRequest metoden.

```
private Job executeRequest(JSONObject json)
```

*Denne metoden oppretter et nytt jobb-objekt (Job) og sender dette videre til kjernen. Så venter den til en arbeidertråd informerer den om at jobben er utført før den returnerer jobben med det ferdige resultatet. Jobb-objektet returneres sjelden fra metodene i systemet fordi de ulike klassene selv holder på en referanse til objektet. Det er derfor unødvendig å returnere nye referanser da jobb-objektet allikevel er oppdatert.*

Så snart "executeRequest" metoden har opprettet et jobb-objekt (Job) og fylt det med de nødvendige parametrene, samt en referanse tilbake til seg selv så den kan "vekkes" når jobben er gjort, sendes det videre via kjernens "queueJob" metode som så videresender jobben til JobManagerns "queueJob" metode. Når det er gjort går kommunikasjonsmodulen inn i en venteperiode mens oppgaven utføres.

```
protected void queueJob(Job job) throws
JobManagerNotRunningException, QueueIsFullException,
NoJobExecutorException, InternalErrorException
```

*Denne metoden er ansvarlig for å undersøke jobben og sjekke at abstraksjonslaget har de nødvendige komponentene for å utføre den. I jobb-objektet finnes det et felt (jobTask) som inneholder navnet på oppgaven som skal utføres. Dette navnet benyttes i et oppslag mot alle de lastede JobExecutor-modulene for å finne ut om noen av dem kan utføre oppgaven. Om den ikke finner en passende modul kastes unntaket "NoJobExecutorException". Hvis den finner en passende JobExecutor-modul lages det en ny instans av den som så knyttes til jobb-objektet. Så går den videre og sjekker om det finnes en modul som kan oversette resultatet til det ønskede formatet (resultFormat feltet i jobb-objektet). Om systemet mangler en passende modul her endres status på jobb-objektet til "feilet" og et "InternalErrorException" kastes. Hvis jobben passerer alle sjekkene blir den plassert i køen (waitingWorkQueue) i påvente av en ledig arbeidstråd (WorkerThread). Til slutt informerer metoden en av de (potensielt) ventende arbeidstrådene om at en ny jobb er klar.*

For å unngå overhead ved opprettelse og avslutting av tråder hver gang en jobb er gjort lages det et sett med arbeidstråder under oppstart. Disse venter så på og få oppgaver. Når en jobb settes i køen informeres en av disse arbeidstrådene om dette ved hjelp av "notify" metoden og tråden "vekkes opp". Det er ikke garantert at arbeidstrådene blir vekket i samme rekkefølge som de ble konstruert, men det er heller ikke nødvendig da alle er identiske. Fordelen med denne løsningen er at systemets kjerne kan fortsette å prosessere innkommende forespørsler samtidig som eksisterende oppgaver kan utføres.

For å begrense antallet samtidige tilkoblinger til databasen opprettes det også et fast sett med databasetilkoblinger under oppstart. Disse plasseres i en "haug" (freeDBConnections) og kan plukkes av arbeidstrådene etter hvert som de kjører. Når en arbeidstråd er ferdig ryddes og returneres databasetilkoblingen til "haugen". Så snart en arbeidstråd blir vekket henter den ut det første jobb-objektet fra køen. Så går den videre og plukker en databasetilkobling fra haugen (requestDBConForJob). Om det ikke er noen ledige tilkoblinger venter den til en blir ledig. Arbeidstråden er nå klar til å utføre oppgaven. Den har alle elementene som trengs, en egen tråd, databasetilkobling og jobb-objektet og kan nå kalle metoden "doJob" på JobExecutor instansen for jobben.

```java
public void doJob(Job job)
```

*Dette er kjernemetoden i hele systemet. Enhver JobExecutor må implementere denne metoden men hvordan den gjør det er ikke spesifisert. Når denne metoden kalles skal JobExecutoren utføre oppgaven spesifisert i jobb-objektet. Når jobben er gjort skal jobb-objektet oppdateres med det nødvendige interne resultatet (internalResult feltet i jobb-objektet) og metoden kan returnere. Fordi arbeidsmodulen allerede har en referanse til jobb-objektet trenger ikke metoden returnere noe.*

En JobExecutor instans har sin egen tråd, sin egen databasetilkobling og informasjonen den trenger for å utføre oppgaven i jobb-objektet og kan derfor kjøre som om den var alene. Om oppgaven krever informasjon hentet fra andre systemer kan den også gjøre dette, men det er viktig at den ikke returnerer før jobb-objektet er oppdatert med et resultat. Hvis jobben feiler må den benytte klassen ErrorObject (se K.4). Denne klassen har en del hjelpefunksjoner som kan brukes til å lage standardiserte feilrapporter og putte disse i jobb-objektet. Om jobben utføres korrekt plasseres resultatet i internalResult feltet i Job klassen og metoden returnerer.

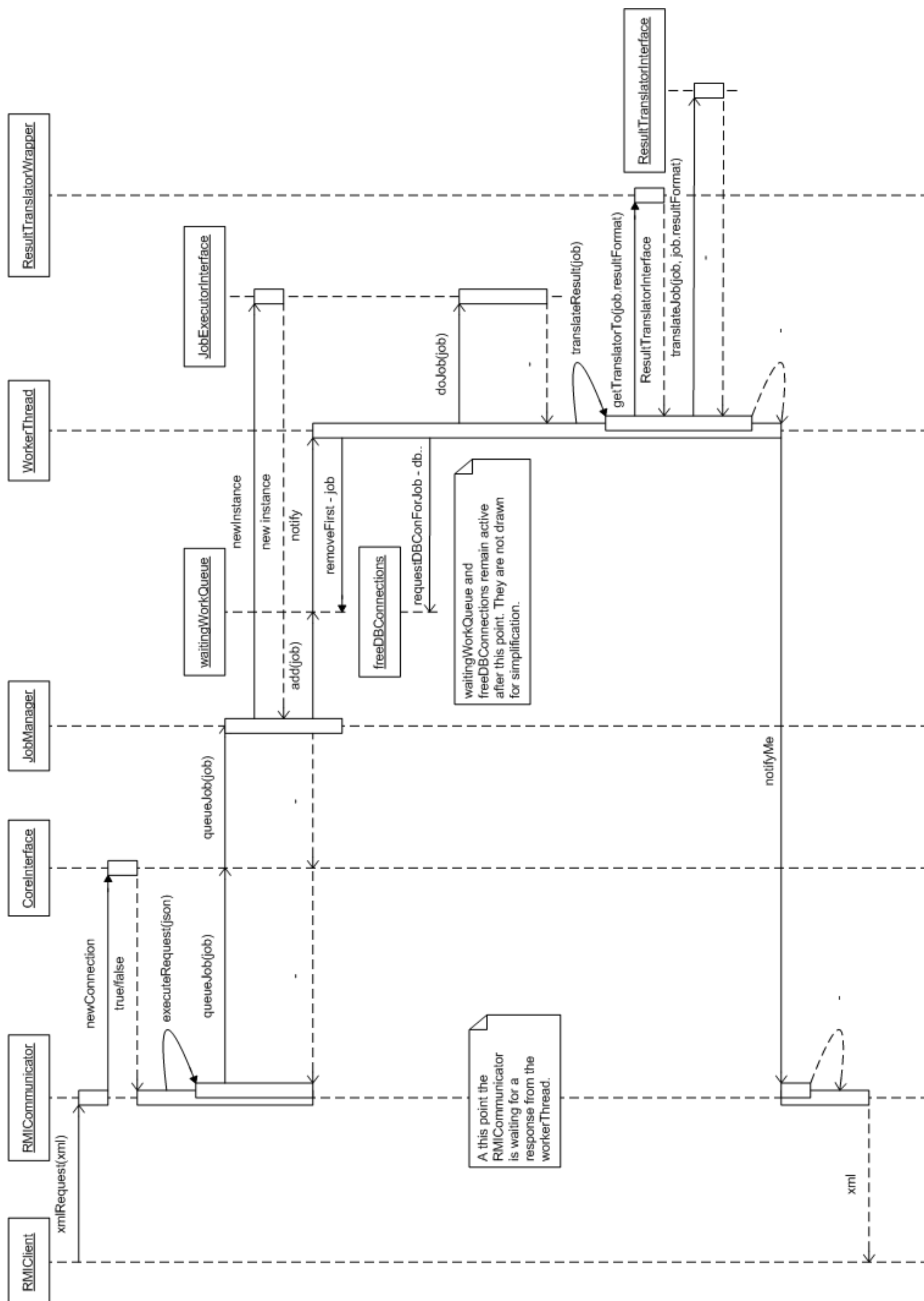
Når jobben er gjort tar arbeidstråden over igjen. Resultatet av jobben ligger nå lagret i jobb-objektet, men for øyeblikket eksisterer det kun som et sett med JSON objekter. For at klienten skal kunne tolke resultatet er det nødt til å oversettes til et tekstformat. Klienten kan selv bestemme hvilket format som ønskes ved å definere dette i forespørselen. For dette eksempelet er det antatt at klienten ønsker resultatet på XML format. Arbeidstråden kaller da en hjelpemetode i JobManager klassen kalt "translateResult". Denne metoden henter fram den nødvendige oversetteren (her XML) ved å se på resultFormat feltet i jobb-objektet og benytter den til å oversette resultatet av jobben.

```java
protected void translateResult(Job theJob)
```

*Før resultatet returneres blir det oversatt ved hjelp av denne metoden. I jobb-objektet finnes et felt kalt "resultFormat" som definerer hvilket format klienten ønsker på resultatet. Dette feltet benyttes i et oppslag i de lastede oversettelsesmodulene og den korrekte klassen returneres. Denne klassen benyttes så til å oversette resultatet og oppdatere jobb-objektet.*

Etter oversettelsen gjenstår det bare opprydning for arbeidsmodulen. Først rydder den opp databasetilkoblingen ved å lukke eventuelle åpne ”statements” og frigjøre disse og så returneres tilkoblingen til ”haugen” så den kan benyttes av andre arbeidstråder. Til slutt kaller den metoden ”notifyMe” på kommunikasjonsmodulen som startet hele operasjonen ved å benytte referansen lagret i jobb-objektets ”communicator” felt.

”executeRequest” metoden som har ventet siden jobb-objektet ble sendt blir nå vekket opp og kan returnere resultatet til klienten. Den vet ingenting om det faktiske resultatet annet enn at det er fylt ut av et av skrittene som har blitt gjennomført. Om det har oppstått en feil eller om jobben ble gjort korrekt betyr ingenting fordi resultatfeltet (”result”) i jobb-objektet alltid har fått en verdi i et av skrittene som er blitt gjennomført. ”executeRequest” modulen returnerer så sitt ferdige jobb-objekt til ”xmlRequest” metoden som så returnerer det endelige resultatet til klienten. Før RMICommunicator-modulen kan fullføre helt må den si ifra til kjernen om at tilkoblingen ikke lenger er i bruk slik at andre kan benytte den. Dette gjøres ved å kalle metoden ”connectionFinished” i kjernen. Klientens forespørsel er nå utført og systemet er tilbake til tilstanden det hadde før klienten koblet seg til.




**Figur 4-7: Sekvensdiagram for en oppgavegjennomføring med RMIClient og RMICommunicator.**

# 5 EVALUERING

## 5.1 UTVIKLINGSMETODE OG PROSESS

Gjennom studiet ved NTNU og arbeid med prosjekter har jeg fått smake på i hovedsak to veldefinerte utviklingsmetoder: Vannfallsmodellen og SCRUM. Vannfallsmodellen er meget restriktiv og egner seg godt for et strukturert prosjekt der man har klare mål og en tydelig vei å gå. SCRUM er mer fleksibel og tillater større endringer uten å kollapse. I tillegg har den mulighet til å forskyve deler av oppgaven frem og tilbake samt legge til og fjerne oppgaver i prosjektet dersom det blir nødvendig.

Utviklingsmetoden jeg benyttet har vært relativt lite strukturert. Den begynte opprinnelig som noe SCRUM liknende fordi jeg ønsket fleksibiliteten denne metoden gav. Jeg hadde laget en "backlog" med et sett med "tasks" som jeg i begynnelsen regnet med jeg måtte gjøre. Det første problemet jeg støtte på med denne prosessen var at den er vanskelig å gjennomføre når man er en person. SCRUM avhenger mye av input fra andre personer i prosjektet og det er slik informasjon som benyttes når det tas avgjørelser om oppgaver skal tilføyes, fjernes eller endres. Disse endringene dokumenteres hele tiden i form av en backlog og hver "sprint" henter de viktigste oppgavene fra denne. Fordi jeg var alene følte jeg at jeg hele tiden hadde oversikt over disse oppgavene og dette resulterte i at jeg ikke oppdaterte backloggen etter hvert som "tasks" ble gjort som igjen resulterte i noe dårlig administrasjon av tid. For det meste følte jeg ikke dette var et stort problem, men i et par tilfeller endte jeg plutselig opp med mye å gjøre og kort tid å gjøre det på.

Når det gjelder utviklingen av prototypen var jeg lenge usikker på hvor jeg skulle begynne. I andre prosjekter har jeg ofte funnet at den beste måten for meg var å begynne programmeringen med noen få ideer, men ikke forvente at det jeg skrev skulle benyttes i det endelige resultatet. Etter å ha definert noen

elementer jeg visst måtte være med i prototypen fulgte jeg denne prosessen. Det er en stor fordel og en stor ulempe ved å utvikle på denne måten. Den store ulempen er at en høyst sannsynlig ender opp med å gjøre en god del arbeid som så må kastes fordi man begynner å programmere før alle delene av programmet er planlagt. Men den store fordelen som etter min mening veier opp for det ekstra arbeidet er at man får muligheten til å se hvordan koden fungerer i langt større grad enn om man kun sitter med teoretiske modeller av systemet. Dette gjør det mulig å komme fram til løsninger som man kanskje ikke tenkte på under første planlegging og så lenge man går inn med forventningen om at man sannsynligvis må gjøre mye på nytt virker det heller ikke spesielt demoraliserende at mye av det man har gjort må forkastes.

Eksempelvis var den første iterasjonen av prototypen kun en kjerne som sendte oppgaver til en arbeidermodul, ventet og så returnerte resultatet. Men etter hvert som jeg jobbet fant jeg blant annet ut at det ville være en stor fordel å kunne begrense ressursbruken til systemet. Den første iterasjonen hadde ingen form for ressurshåndtering. Den kunne heller ikke utføre flere jobber samtidig, noe som er ganske relevant. Jeg begynte derfor å gjøre endringer i koden for å få den til å passe med ideene som etter hvert kom inn. Til slutt endte jeg opp med et meget ustrukturert system der mye av koden var kompleks og i flere tilfeller duplisert, men den store fordelen var at alle ideene var på plass. Jeg endte dermed opp med å begynne omtrent fra bunnen av med mye av koden, men fordi jeg nå hadde en klar oversikt over hvilke funksjoner jeg ville implementere var det mye lettere å strukturere systemet.

Min utviklingsprosess er preget av relativt lite opprinnelig planlegging. Jeg benytter meg heller av flere iterasjoner der de første ideene blir bearbeidet og raffinert. Det er mange feller å gå i med en slik prosess og jeg er fult klar over at det kan medføre at mye arbeid må gjøre som igjen. Jeg mener allikevel jeg har klart å finne en balansegang. Ved heller å fokusere på strukturelle problemer i de første iterasjonene og hoppe over kompleks implementasjon av algoritmer kan jeg få på plass et rammeverk (riktignok uten fungerende kjerne) som jeg så kan benytte til å teste ut ideene mine på. Oppgaven er relativt løst definert med tanke på hva jeg er nødt til å finne ut før jeg kan gå i gang og jeg hadde allerede tilgang på mye informasjon fra den tidligere oppgaven og min veileder som også virket som informasjonskilde for instituttets planleggingsrutiner. Fordi jeg ikke var helt sikker på hvilke funksjoner jeg ønsket i starten var det bedre for meg å begynne med en ting og så gjøre forandringer underveis. Den første iterasjonen måtte forkastes og jeg var nødt til å begynne på nytt, men fordi jeg hadde forventet et slikt scenario hadde jeg ikke brukt mye tid på å få detaljene

på plass. Jeg konstruerte et rammeverk som så gav meg nye ideer. Når jeg etter hvert begynte programmeringen på prototypen som ble det endelige resultatet hadde jeg gjort mye planlegging. Ikke i form av en stor mengde modeller og dokumenter, men i form av et halvferdig prosjekt som jeg kunne studere og arbeide videre med. Resultatet er etter min mening en relativt god arbeidsprosess der jeg både får gjennomført planlegging og samtidig får testet ut ideene.

## 5.2 INFORMASJONSINNSAMLING

Fordi oppgaven baserte seg på det jeg gjorde høsten 2009 hadde jeg allerede en del informasjon på plass fra samtaler med veiledere og studie av It's Learing og Fronter. Dette medførte at jeg gikk inn i oppgaven med et inntrykk av at jeg hadde en relativt grei oversikt over rutiner benyttet ved planlegging av fag for instituttene. Jeg var klar over at det var store variasjoner på disser rutinene fra sted til sted, men mente allikevel at jeg hadde et godt utgangspunkt. Utviklingsarbeidet ble derfor påbegynt relativt tidlig og dette var en av grunnene til at jeg, etter samtaler med ulike parter fant ut at det var best å starte på nytt. En verdifull lekse om at man skal være veldig forsiktig med å arbeide ut ifra egne antagelser.

Jeg visste hele tiden at jeg burde finne ut mer om hvordan de ulike instituttene gjennomførte fagplanlegging, hvilke systemer som blir benyttet til denne prosessen og hvilke systemer som allerede eksisterer sentralt for administrasjon av slik informasjon. Det jeg ikke var helt forberedt på var omfanget av disse systemene.

Jeg hadde en samtale med IT avdelingen for NTNU (ITEA) i midten av mars hvor jeg fikk en introduksjon til hvilke programmer og databaser som er i drift og hva disse systemene inneholder. Dette inkluderer FS (Felles Studentsystem), Studweb, EpN (Emner på Nett), Syllabus og andre. Disse systemene jobber sammen og utveksler informasjon med hverandre. Det som kom tydelig fram her er hvor urealistisk det egentlig er å forvente at *et* enkelt system skal kunne holde på all informasjonen. Jeg ble mer og mer overbevist om at det heller ikke er den beste løsningen og lagre all data på et sted. Måten det ble presentert på medførte også til at jeg gikk fra møte noe demotivert og med en følelse av at de problemene jeg ønsket å løse allerede var løst ved hjelp av kombinasjoner av de eksisterende systemene. Det virket på meg som om min oppgave rett og slett var unødvendig fordi ITEA allerede hadde de nødvendige systemene på plass

noe som gikk litt imot det jeg hadde fått vite tidligere om planleggingsprosessen for IDI.

For å klare opp i dette valgte jeg å gå til instituttene og prøve å få litt mer detaljert informasjon om hvordan de selv planla og gjennomførte fag. Jeg valgte meg ut de tre instituttene Fysikk, Bygg og Pedagogikk med håp om at de skulle gi meg ulike vinklinger på prosessen. Fordi jeg hadde hatt flere samtaler med veileder som også har drevet med planlegging for IDI hadde jeg allerede fått vite mye om hvordan dette ble gjennomført her. Resultatet ble mye verdifull informasjon. Det viste seg nemlig at instituttene enten hadde laget egne interne systemer for planlegging (fysikk) eller ikke benyttet seg av noen digitale løsninger i det hele tatt[1] (bygg og pedagogikk). Inntrykket jeg fikk fra instituttene var at de fleste mente at systemene levert av ITEA var gode, men at de kun adresserte problemer som en ”outsider” ville fått øye på. De manglet innsikten som kom fra førstehåndserfaring ved gjennomføring av disse oppgavene. Det var også store variasjoner i metodene som ble benyttet. Instituttet for fysikk hadde en meget strukturert framgangsmåte mens bygg var veldig flytende og varierte ofte fra år til år. Felles for alle instituttene var at de benyttet informasjon fra de ferdige systemene som grunnlag og så innførte egne løsninger for selve prosessen. For eksempel kunne de hente ut informasjon om studieretning og studentene fra FS, registrere nye fag i EpN og administrere rom og forelesningssaler via Syallabus, men de hadde intet sentralt system som holdt på informasjon som benyttes direkte i planleggingen. Alle jeg pratet med var også positive til ideen om en løsning som kunne bidra til enklere og mer effektiv håndtering av denne informasjonen.

Det er farlig å gå inn i et prosjekt med ideen om at man allerede vet mye av det som er nødvendig for å fullføre det. Jeg begynte prosjektet med en god del data fra tidligere, noe som medførte at jeg var relativt sikker på at jeg hadde oversikt over det nødvendige. Jeg var fult forberedt på å måtte søke fram mer informasjon fra ulike kilder underveis, men ble overrasket over hvor mange detaljer jeg ikke hadde klart for meg. Om jeg skulle gjort prosjektet igjen ville jeg begynt arbeidet med å kontakte de ulike instituttene for å innhente informasjon om hvordan de arbeidet før jeg gjorde noen form for programmering.

---

[1] Med dette mener jeg spesialtilpassede systemer. Jeg går ut ifra at de alle bruker tekstbehandlingsprogrammer og andre standard kontorprogrammer.

## 5.3 PROTOTYPEN

Det endelige resultatet av prosjektet ble en databasestruktur basert på den beskrevet i oppgaven fra høsten 2009 (1) og en prototype av abstraksjonslaget som håndterer kommunikasjon og interaksjon mellom klienter og databasen, men om systemet skal kunne benyttes i praksis er det mange egenskaper som må evalueres. Ytelse, fleksibilitet, brukervennlighet og kompleksitetsgrad ved utvikling av nye moduler er de faktorene jeg vil fokusere mest på fordi jeg mener disse er de viktigste for om systemet vil bli tatt i bruk eller ikke.

**Ytelse**

Den første faktoren er ytelse. Hvor effektivt systemet kan utføre jobber er mest relevant så systemet har blitt testet med tanke på overhead[2]. Ytelsen ved utføring av en jobb er naturlig nok mest avhengig av hva jobben går ut på og hvor kompleks den er. Fordi det er vanskelig å vite på forhånd hvilke oppgaver systemet til slutt kan måtte utføre på grunn av støtten for utvidelse ved moduler er det tiden jobben bruker internt i kjernen som er mest relevant. Det er denne tiden som kan optimaliseres konkret før alle modulene er på plass. Tallene som ble målt er også avhengig av ressursene tilgjengelig på maskinen. I testøyeblikket kjørte flere andre programmer samtidig og systemet var under relativt høy belastning. Etter å ha kjørt 100 jobber 5 ganger ble gjennomsnittstiden en jobb brukte inne i kjernen (ikke inkludert tiden arbeidet tok) målt til 3,7ms der høyeste målte verdi var 8ms (3 forekomster) og laveste verdi var 1ms (1 forekomst). Det vil si at kjernen alene kan prosessere ca 270 jobber per sekund.

Et annet aspekt ved ytelse er tiden det tar å starte opp og avslutte programmet. Fordi det er modulært er det en del operasjoner som må utføres under oppstart og avslutting for å hente inn og gjøre klar moduler. Moduler kan både være bygget inn i jar filen sammen med kjernen og komme i eksterne jar filer der sistnevnte naturlig nok tar mer tid. Dette punktet er derimot mindre relevant fordi systemet optimalt kun skal startes og stoppes en sjelden gang. Når systemet først kjører behøver man ikke bruke tid på å laste inn nye moduler. Internt i systemet benyttes optimaliserte datastrukturer[3] for å holde på referanser til modulene så tiden det tar å finne fram en modul er minimal.

---

[2] Tidsbruk utenfor faktisk utførelse av oppgavene.

[3] Med dette menes strukturer som Hash tabeller og liknende. Når en klient sender en jobb må man finne ut hvilken modul som skal løse dette. Navnet på jobben brukes som nøkkel i et HashMap og man har dermed O(1) oppslagstid.

**Kodestruktur**
En viktig regel jeg har prøvd å følge gjennom hele programmeringsprosessen er "Compartmentalization". Det vil si at jeg har forsøkt å dele koden opp i logiske deler som hver kan virke omtrent uavhengig. Eksempler på dette er skille mellom kjernen og moduler, skille mellom de ulike modulene og mellom hjelpeklasser som UserData eller ConfNode og kjernen. Ved å holde elementene adskilt blir det enklere å arbeide på hver enkelt bit og lettere å feilsøke dersom det skulle oppstå problemer i en del. I de mest dramatiske tilfellene kan man også relativt enkelt erstatte hele seksjoner med ny kode.

Kodestrukturen og klassene bærer stort preg av at dette tross alt er en prototype og ikke produksjonsklar kode. Fordi jeg ønsket å få fram ideene og funksjonene i koden har jeg ikke brukt veldig mye tid på å optimalisere strukturen. Noen av klassene i koden har også fått implementert funksjoner som bryter delvis med regelen jeg satte. Det tydeligste eksempelet på dette er kommunikasjonsmoduler som også er nødt til å oversette forespørselen til internformatet (JSON). Dette medfører at hver modul må implementere oversettelsesfunksjonalitet for eksempel for XML til JSON noe som betyr duplisering av kode. En bedre løsning ville vært om jeg benyttet oversettelsesmodulene eller noe tilsvarende til å gjøre denne konverteringen. Med en slik løsning vil man minske feilkildene, redusere duplisering og forenkle konstruksjonen av kommunikasjonsmodulene.

I den første iterasjonen av prototypen hadde jeg også separert JobManagern fra resten av kjernen. JobManagern er den delen av koden som sørger for ressurshåndtering og fordeling av jobber på arbeidermoduler. Fordelen ved dette var at det ble mulig og relativt enkelt bytte ut ressurshåndteringsalgoritmen. I senere iterasjoner slo jeg fra meg dette mest fordi jeg ikke ønsket å introdusere mer kompleksitet i prototypen, men en slik mulighet er noe som jeg kan vurdere for fremtidig arbeid.

For en detaljert gjennomgang av strukturen se seksjon 4.1.

## 5.4 OPPSUMMERING

Planlegging av fag er en omfattende prosess om alle instituttene ved NTNU må gjennomføre før hvert semester kan begynne. Planleggingen baserer seg på bruk av informasjon fra et stort antall ulike kilder. Alt fra økonomiavdelingens budsjetter til studentantall og fagomfang er viktige elementer som må

inkluderes i en slik prosess. Fram til i dag har instituttene selv utviklet egne rutiner og i noen tilfeller også egen programvare og databaser for å forenkle denne oppgaven. Dette har ført til en del problemer blant annet med lagring av eldre dokumentasjon for fag og samarbeid på tvers av institutter.

Begynnelsen på en løsning ble foreslått som en del av oppgaven "Læringsadministrative systemer i norsk høyere utdanning" høsten 2009 og videre arbeid med problemet basert på løsninger foreslått der ble utviklet i denne oppgaven. Resultatet har blitt en databasemodell med mulighet for lagring og dokumentasjon av de aller fleste aspekter ved planlegging og gjennomføring av fag. Det ble også utviklet en prototype av en applikasjon kalt "abstraksjonslaget". Formålet med dette programmet var å forenkle interaksjon med databasen ved å fasilitere kommunikasjon fra ulike kilder og med ulike protokoller. Abstraksjonslaget kan også forsterke de eksisterende tilgangsbegrensningene ved å tilby ytterligere sikkerhetsmekanismer oppå de som gis av databasesystemet. Kombinasjonen av databasemodellen og abstraksjonslaget gir en robust løsning som både ivaretar omfattende dokumentasjon, kan benyttes i planleggingsprosessen og samtidig er fleksibel nok til å tillate toveis kommunikasjon med eksisterende systemer. Det er forfatterens håp at denne løsningen skal bidra til enklere og mer effektive rutiner rundt planlegging og gjennomføring av fag og dermed også et bedre fagtilbud ved instituttene.

# 6 VIDERE ARBEID

Resultatet av dette prosjektet ble en prototype av abstraksjonslaget for kommunikasjon og interaksjon med databasen. Men selv om prototypen er kjørbar og kan utføre enkelte operasjoner er det fortsatt mye som gjenstår før programmet kan tas i bruk ved instituttene.

Det viktigste punktet som gjenstår er bearbeiding av tabellene i databasen. Med prototypen følger det et skript som genererer tabeller og relasjoner, men tabellene inneholder som regel kun identifiseringsfelt og felter som benyttes i relasjonene. Videre arbeid vil derfor inkludere og finne ut hvilke felter som er relevant for de ulike tabellene, eksempelvis hvilke felter som trengs for å lagre nødvendig informasjon om fag. Før disse feltene er på plass er det også vanskelig å konstruere nye arbeidermoduler.

En endring som med fordel kan introduseres er muligheten til å benytte moduler til oversetting fra innkommende forespørsler til det interne formatet. Dette kan løses ved å opprette en egen type moduler eller benytte oversettere. Egne moduler gir den beste formen for "compartmentalization", men siden oversetterne allikevel vil måtte konvertere fra internformatet tilbake til forespørselsformatet er det mer logisk å plassere denne funksjonaliteten her. En mulig løsning som vil bedre kommunikasjonen med andre systemer som utveksler XML dokumenter er støtte for XSLT (19). Java har allerede flere biblioteker med omfattende støtte for dette og fordi JSON biblioteket som benyttes i prototypen også enkelt kan konvertere til og fra XML vil det være en relativt billig løsning på problemet med ulike XML standarder. Dette krever også at det lages XSLT dokumenter som beskriver konverteringen mellom formatene.

Prototypen har et lite sett med medfølgende moduler. Det er en kommunikasjonsmodul, fem arbeidere og to oversettere. Disse er ment å demonstrere hvordan modulene kan konstrueres, men i seg selv gjør de lite.

For at systemet skal kunne benyttes til praktisk bruk er det nødvendig å lage moduler som kan utføre et stort antall ulike oppgaver. Etter samtale med ITEA fikk jeg vite at de aller fleste systemene benytter et standardisert format for XML dokumenter kalt IMS Enterprise. For at disse skal kunne kommunisere med mitt system trengs det en modul som kan oversette IMS dokumenter til det interne formatet.

Kommunikasjonsmetodene i prototypen er relativt komplekse og inkonsistente. Det ville gjøre arbeidet med modulutvikling lettere om kjernen ble skrevet om til å benytte event-basert kommunikasjon. Dette er en metode basert på designmønsteret ”observer” (20) der deler av koden kan ”abonnere” på endringsmeldinger fra andre deler av koden. Fordelen ved et slikt mønster er at kjernen ikke behøver å vite om hvilke modulen den må informere når spesifikke handlinger skjer. Med event-basert kommunikasjon vil kjernen melde fra om endringer og operasjoner og modulene kan selv ”melde seg på” kun de hendelsene som er relevant for den selv.

I kjernen er det implementert en enkel algoritme for ressurshåndtering og begrensninger. Den baserer seg på en FIFO kø der jobbene plasseres og utføres i den rekkefølgen de kommer inn. Måten systemet begrenser samtidige jobber på er ved å begrense antallet ”arbeidertråder” som får kjøre. Dette gir en rettferdig ressursfordeling der alle jobbene stilles likt, men det er ikke nødvendigvis alltid den beste metoden. Et eksempel er tilfeller der en av jobbene rett og slett er en backuprutine som sikkerhetskopierer alle data i databasen. Det er ikke nødvendig at den kjører så snart den får mulighet, men det vil skje med det eksisterende systemet. Hadde det vært mulig å sette prioritet på jobbene ville man kunne nedprioritere slike oppgaver, mens viktigere operasjoner fikk kjøre først.

Det kreves omfattende testing i et realistisk miljø for å finne ut hvilke algoritmer som vil egne seg best til rettferdig og effektiv ressursfordeling. Sannsynligvis vil man også oppdage at det ikke finnes en algoritme som alltid er best. En utvidelse av systemet til å inkludere muligheten for rask utbytting av denne delen vil bidra til økt effektivitet og mer vedlikeholdbar kode.

# 7 BIBLIOGRAFI

1. **Rudfoss, Thomas Haugland.** Læringsadministrative systemer i norsk høyere utdanning. 2009.

2. —. Intervju med fagansvarlig på tre ulike institutter. 2010.

3. —. Samtale med ITEA. *Samtale med ITEA.* Trondheim : s.n., 16 Mars 2010.

4. **Wikipedia.** Object database. *Object database.* [Internett] [Sitert: 12 Februar 2010.] http://en.wikipedia.org/wiki/Object_database.

5. **MySQL.** MySQL: The world's most popular open source database. *MySQL: The world's most popular open source database.* [Internett] [Sitert: 3 Mai 2010.] http://www.mysql.com/?bydis_dis_index=1.

6. **PostgreSQL.** PostgreSQL: The worlds most advanced open source database. *PostgreSQL: The worlds most advanced open source database.* [Internett] [Sitert: 3 Mai 2010.] http://www.postgresql.org/.

7. **Free Software Foundation (FSF).** GNU General Public License. *GNU General Public License.* [Internett] [Sitert: 3 Mai 2010.] http://www.gnu.org/licenses/gpl.html.

8. **WikiVS.** MySQL vs PostgreSQL. *MySQL vs PostgreSQL.* [Internett] [Sitert: 3 Mai 2010.] http://www.wikivs.com/wiki/MySQL_vs_PostgreSQL.

9. **Wikipedia.** ACID. *ACID.* [Internett] [Sitert: 12 Februar 2010.] http://en.wikipedia.org/wiki/ACID.

10. —. View (database). *View (database).* [Internett] [Sitert: 3 Mai 2010.] http://en.wikipedia.org/wiki/View_(database).

11. **IMS Global Learning Consortium.** IMS Enterprise Information Model. *IMS Enterprise Information Model.* [Internett] IMS Global Learning Consortium, 01

Juli 2002. [Sitert: 15 April 2010.] http://www.imsglobal.org/enterprise/entv1p1/imsent_infov1p1.html#1425770.

12. **Wikipedia.** Adapter pattern. *Adapter pattern.* [Internett] [Sitert: 13 Mai 2010.] http://en.wikipedia.org/wiki/Adapter_pattern.

13. **David Ferraiolo, Ramaswamy Chandramouli, Richard D. Kuhn.** *Role-based access control.* 2. utgave. Boston : Artech House, 2007.

14. **Wikipedia.** *Salt (cryptography).* [Internett] Mars 2010. http://en.wikipedia.org/wiki/Salt_(cryptography).

15. —. *Rainbow tables.* [Internett] Mars 2010. http://en.wikipedia.org/wiki/Rainbow_table.

16. **Bratbergsengen, Kjell.** Muntlig kommunikasjon. 2010.

17. **Wikipedia.** Bridge pattern. *Bridge pattern.* [Internett] [Sitert: 13 Mai 2010.] http://en.wikipedia.org/wiki/Bridge_pattern.

18. **JSON.** *JSON.* [Internett] Mars 2010. http://www.json.org/.

19. **W3C.** XSL Transformations. *XSL Transformations.* [Internett] [Sitert: 18 Mai 2010.] http://www.w3.org/TR/xslt.

20. **Wikipedia.** Observer pattern. *Observer pattern.* [Internett] [Sitert: 20 Mai 2010.] http://en.wikipedia.org/wiki/Observer_pattern.

21. **Sun.** Level (Java 2 Platform SE v1.4.2). *Level (Java 2 Platform SE v1.4.2).* [Internett] [Sitert: 20 Februar 2010.] http://java.sun.com/j2se/1.4.2/docs/api/java/util/logging/Level.html.

22. —. Logger. *Logger.* [Internett] [Sitert: 20 Februar 2010.] http://java.sun.com/j2se/1.4.2/docs/api/java/util/logging/Logger.html.

23. **MySQL.** Prepared Statements. *Prepared Statements.* [Internett] [Sitert: 30 Januar 2010.] http://dev.mysql.com/tech-resources/articles/4.1/prepared-statements.html.

24. **Wikipedia.** *Thread pool pattern - Wikipedia.* [Internett] Mars 2010. http://en.wikipedia.org/wiki/Thread_pool.

25. **NTNU.** *Syllabus.* [Internett] April 2010. http://www.ntnu.no/sa/sfs/studsys/timeplan.

26. —. *Studentweb.* [Internett] April 2010. http://www.ntnu.no/studier/studentweb.

27. **Sun.** *Statement.* [Internett] Mars 2010. http://java.sun.com/j2se/1.4.2/docs/api/java/sql/Statement.html.

28. **Wikipedia.** *Role-based access control.* [Internett] Mars 2010. http://en.wikipedia.org/wiki/Role-based_access_control.

29. —. *Resource starvation - Wikipedia.* [Internett] Mars 2010. http://en.wikipedia.org/wiki/Resource_starvation.

30. **W3C.** *RDF - Semantic Web Standards.* [Internett] Februar 2010. http://www.w3.org/RDF/#Resource_Description_Framework_.28RDF.29.

31. **It's learning.** *It's learning - læringsplattform.* [Internett] April 2010. http://www.itslearning.no/.

32. —. it's learning - Integrasjoner. *it's learning - Integrasjoner.* [Internett] It's learning. [Sitert: 15 April 2010.] http://www.itslearning.no/integrasjoner.

33. **It's Learning.** *it's learing 3.3.* [Internett] Februar 2009. https://www.itslearning.com/.

34. **Wikipedia.** *Interface (Java) - Wikipedia.* [Internett] Mars 2010. http://en.wikipedia.org/wiki/Interface_(Java).

35. **NTNU.** *Historien om FS.* [Internett] April 2010. http://www.ntnu.no/portal/page/portal/ntnuno/tre-spalter?selectedItemId=27465&rootItemId=25703§ionId=4562&_piref36_794797_36_794780_794780.artSectionId=4562&_piref36_794797_36_794780_794780.articleId=11788&_piref36_794799_36_794780_794780.sectionId=45.

36. **Sun.** *HashMap.* [Internett] Mars 2010. http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html.

37. **Wikipedia.** *FIFO - Wikipedia.* [Internett] Mars 2010. http://en.wikipedia.org/wiki/FIFO.

38. **NTNU.** *Emner på Nett.* [Internett] April 2010. http://www.ntnu.no/sa/sfs/studsys/emner.

39. **Wikipedia.** *Deadlock - Wikipedia.* [Internett] Mars 2010. http://en.wikipedia.org/wiki/Deadlock.

40. —. Remote procedure call. *Remote procedure call.* [Internett] [Sitert: 3 Mai 2010.] http://en.wikipedia.org/wiki/Remote_procedure_call.

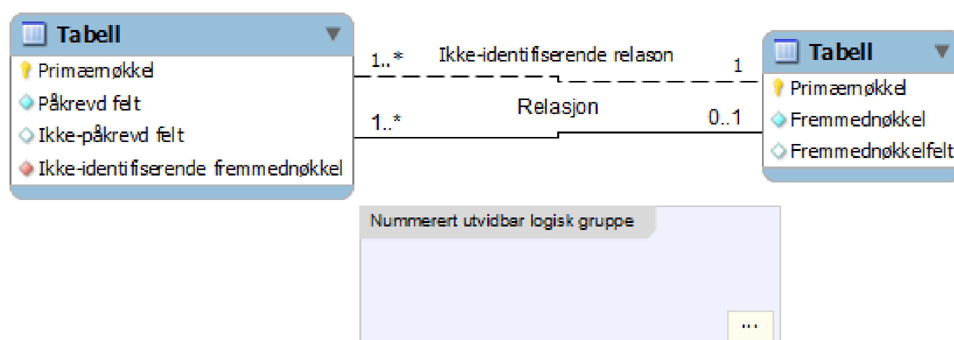
41. —. Facade pattern. *Facade pattern.* [Internett] [Sitert: 13 Mai 2010.] http://en.wikipedia.org/wiki/Facade_pattern.

42. —. Design pattern. *Design pattern.* [Internett] [Sitert: 2 Februar 2010.] http://en.wikipedia.org/wiki/Design_pattern_(computer_science).

43. **Dublin Core Metadata Initiative.** *DCMI Home.* [Internett] Februar 2010. http://dublincore.org/.

44. **Goyvaerts, Jan.** Regular-Expressions.info. *Regular-Expressions.info.* [Internett] [Sitert: 20 Februar 2010.] http://www.regular-expressions.info/.

# Appendiks A Tegnforklaring for databasemodellen




**Figur A-1: Tegnforklaring for databasemodellen.**

**Tabell A-1: Definisjoner på elementene i tegnforklaringen.**

| Forklaringer | |
|---|---|
| Primærnøkkel | Identifiserer et element i tabellen unikt. |
| Påkrevd felt | Må fylles ut for alle elementer i tabellen. |
| Ikke-påkrevd felt | Behøver ikke å fylles ut. |
| Ikke-identifiserende fremmednøkkel | Fremmednøkkel til en annen tabell i systemet. Feltet som det pekes til er ikke primærnøkkel i denne tabellen. |
| Fremmednøkkel | Et påkrevd felt som peker til en primærnøkkel i en annen tabell. |
| Fremmednøkkelfelt | Et ikke-påkrevd felt som (i dette tilfellet) er referert til i en annen tabell. |
| Nummerert utvidbar logisk gruppe | En gruppe med tabeller hentet inn fra en annen oversikt. Behøver ikke nødvendigvis være nummerert. |

# Appendiks B Installasjons-veiledning og krav

For å installere og teste prototypeapplikasjonen trengs det en maskin med Java installert. Det er også nødvendig å ha tilgang på en MySQL database enten på den samme maskinen eller en annen som prototypen har tilgang til. Denne veiledningen går ut ifra at det benyttes en maskin både som MySQL server og til å kjøre prototypen. Maskinen antas å ha både Java Runtime Environment installert og kjøre MySQL Server på Windows.

## B.1 INITIALISERE DATABASE

1. Opprett en ny database på MySQL serveren.
2. Opprett en bruker som har alle rettigheter på den nye databasen.
3. Kjør filen initdatabase.sql på databasen for å opprette tabeller og noen enkle testdata. Skriptet oppretter også en rotbruker (administrator) med en bestemt ”token”. Denne tokenen eller identifikatoren må benyttes når man skal utføre de første operasjonene på databasen. Uten denne vil prototypen nekte tilgang. Tokenen finnes på linje 965 i initdatabase.sql og plasseres i XML forespørselen som skal utføres. Se Appendiks E for en detaljert beskrivelse av de ulike XML dokumentene for operasjonene.
4. Åpne filen config.xml og endre nøkkelen <core><db> slik at den passer med databasen og brukeren som ble opprettet.
   URLen ser slik ut: jdbc:mysql://[servernavn]:[port]/[database]
   Eksempel med databasen ”master” på lokal maskin:
   jdbc:mysql://localhost:3306/master

Du kan nå konfigurere prototypen ved å endre på innstillingene i config.xml For mer informasjon om innholdet in config.xml se Appendiks C.

Hvis du ønsker å benytte RMI modulen for å kommunisere med prototypen er du nødt til å starte Javas RMI registry først. Dette gjøres ved å åpne et kommandovindu og skrive ”rmiregistry”. Prototypen antar at RMI registeret benytter standardporten for kommunikasjon.

### STARTE PROTOTYPE OG RMI KLIENT AUTOMATISK

For enkelthets skyld er det lagt ved en bat fil ”start.bat” som kan benyttes til oppstart av alt som behøves for å teste prototypen. Skriptet starter opp Java RMI Registry, Core.jar og RMIClient.jar i den rekkefølgen.

**Starte prototypen manuelt:**

1. Sørg for at følgende filer ligger i samme mappe:
   Core.jar
   config.xml
   mysql-connector-java-5.1.12-bin.jar
   RMICommunicator.jar
   RMIClient.jar
2. Åpne config.xml, rull ned til <plugins>, <clientCommunicators> og nøkkelen med <name> RMICommunicator. Oppdater <jarPath> til å tilsvare mappen der RMICommunicator.jar ligger.
   De resterende plugins følger med internt i Core.jar og trenger derfor ikke endres.
3. I et kommandovindu skriver du ”java -jar Core.jar”.
4. Etter at kjernen har startet opp og er klar åpner du et nytt kommandovindu og skriver
   ”java -jar RMIClass.jar”.
5. Kommandovindu vil nå gi output fra prototypen og du kan sende forespørsler via klienten. Se i feilsøkingsseksjonen under om det skulle oppstå feil.

**VIKTIG:** Hvis ikke filene ligger i samme mappe må du oppdatere manifestet for Core.jar og RMIClient.jar før du kan kjøre dem. Dette er fordi Core.jar antar at mysql-connector-java-5.1.12-bin.jar filen ligger i samme mappe som den og RMIClient.jar antar at RMICommunicator.jar filen finnes i samme mappe som den.

## B.2 FEILSØKING

**Problem:** Får en advarsel under oppstart: ”java.sql.SQLException: No suitable driver found…”

**Løsning:** Problemet skyldes at filen mysql-connector-java-5.1.12-bin.jar ikke ligger i samme mappe som Core.jar. Om du ønsker å benytte en annen versjon av jdbc driveren eller koble til en annen database må du også endre manifestfilen (META-INF\MANIFEST.MF) i Core.jar til å laste inn den korrekte jdbc driveren under oppstart.

**Problem:** Får en advarsel under oppstart: ”JobManager: Failed to connect to database!...”

**Løsning:** Prototypen kan ikke koble til databasen. Sjekk at du har opprettet bruker og database for prototypen. Sjekk også at <url>, <user> og <password> er korrekt i config.xml filen.

**Problem:** Får en advarsel under oppstart: ”Could not load [modulklasse]…”

**Løsning:** En av modulene definert i config.xml filen kan ikke lastes inn. Dette kan skyldes at jar-filen som det pekes til ikke finnes eller ikke er lesbar eller at klassenavnet er skrevet feil eller ikke finnes inne i selve jar-filen. Sørg for at elementene i config.xml er definert korrekt og at filen finnes.

**Problem:** Får en advarsel under oppstart: ”java.rmi.ConnectException: Connection refused…”

**Løsning:** Denne feilen oppstår hvis prototypen startes opp med RMICommunicator modulen, men uten at rmiregistry er startet på forhånd. Avslutt kjernen, start rmiregistry og start kjernen igjen for å løse problemet.

# Appendiks C Beskrivelse av konfigurasjonsinnstillinger

Denne appendiksen vil gå gjennom de tre seksjonene av filen config.xml og beskrive verdier og effekter for de ulike elementene. Konfigurasjonsfilen er en XML fil som definerer innstillinger for prototypen. Når programmet starter opp lastes denne filen fra samme mappe som den Core.jar ligger i. Innstillingene har fire datatyper og gyldige verdier for de forskjellige er:

- Tall: Heltall større enn 0
- Tekst: En tekst streng
- Boolsk verdi: true / false eller yes / no
- Loggingsnivåverdi: Dette er en tekst som definerer granulariteten av logging. Den benytter de samme konstantene som Java Logging Level klassen (20) med små bokstaver.

## CORE

Core seksjonen inneholder innstillinger som benyttes i kjernen.

**Tabell C-2: Beskrivelse av konfigurasjonselementene for kjernen.**

| Element | Standard | Beskrivelse |
|---|---|---|
| **workerThreadCount** | 20 | Antallet arbeidertråder som opprettes. Jo flere arbeidertråder jo flere jobber kan utføres i parallell. |
| **maxJobQueueSize** | 20 | Maksimalt antall jobber som kan vente i kø før nye jobber blir avvist. |
| **jobTimeout** | 5 | Maksimalt antall sekunder en jobb kan tilbringe i systemet (både i kø og under kjøring). |
| **enableConsoleControls** | true | Muliggjør enkle administrative kommandoer via en konsoll i kommandolinjen mens kjernen kjører. Når kjernen har startet opp skriv "hjelp" og trykk enter for en liste over kommandoer. |
| **adminRoleName** | root | Rollenavnet for administratorrollen. I databaseskriptet opprettes denne rollen og en bruker. Hvis du vil gi rollen et annet navn må du også endre dette for at systemet skal kunne gjenkjenne administratorer. |
| **db** | | Denne delen spesifiserer innstillinger for databasen |
| **workerDBConnectionCount** | 20 | Antallet simultane databasetilkoblinger. For optimal ytelse bør det være like mange databasetilkoblinger som tråder. Det gir ikke mening at dette tallet er større enn antall tråder da en tråd kun benytter en tilkobling. |
| **connectionRetries** | 3 | Hvor mange ganger systemet skal prøve på nytt før tilkoblingen avbrytes. Kjernen vil avslutte umiddelbart om den ikke får kontakt med databasen. |
| **url** | jdbc:mysql://localhost:3306/master | Tilkoblingsadressen som benyttes av JDBC driveren for å opprette kommunikasjon med databasen. |
| **user** | master | Brukernavnet for database. |
| **password** | master | Passordet for database. |

## LOGGING

Logging seksjonen inneholder innstillinger for loggingssystemet i kjernen og modulene.

**Tabell C-3: Beskrivelse av konfigurasjonselementene for logging.**

| Element | Standard | Beskrivelse |
|---|---|---|
| **enableCoreLogger** | yes | Aktiver eller deaktiver logging for kjernen |
| **enablePluginsLogger** | yes | Aktiver eller deaktiver logging for moduler (plugins) |
| **logTimers** | yes | Gjør at kjernen skriver tidsdata til loggen etter at hver jobb har blitt utført. Tidsdata inneholder jobbens kjøretid, overhead og fullstendig tid i systemet. |
| **coreLogLevel** | finest | Bestemmer hvor detaljert kjerneloggen skal være. |
| **pluginsLogLevel** | finest | Bestemmer hvor detaljert modulloggen skal være. |

### PLUGINS

Plugins seksjonen inneholder innstillinger for plugins i tre undergrupper:

- clientCommunicators: Kommunikasjonsmoduler
- jobExecutors: Arbeidermoduler
- resultTranslators: Oversettelsesmoduler

Hver av undergruppene kan inneholde et eller flere <plugin> elementer og hvert <plugin> element kan ha mange ulike innstillinger. Fra kjernens side er det kun definert tre og kun to av dem er obligatoriske.

**Tabell C-4: Beskrivelse av konfigurasjonselementene for moduler (plugins).**

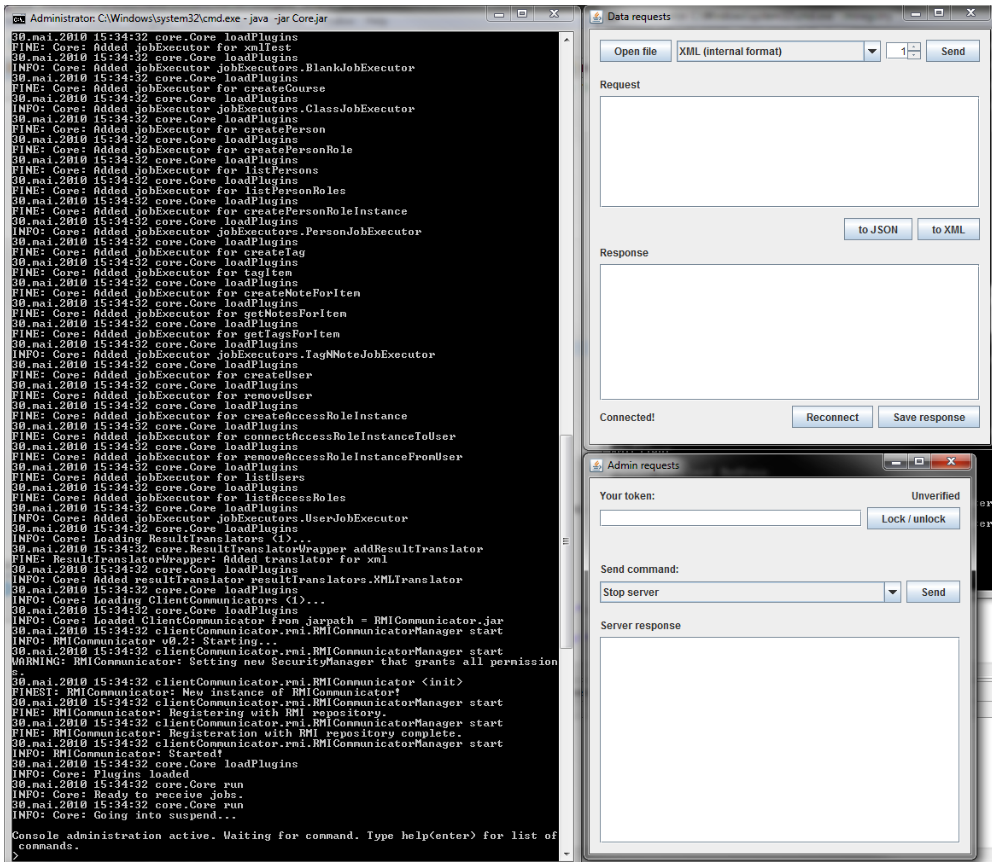
| Element | Beskrivelse |
|---|---|
| **name** | Navnet på modulen. Dette benyttes i loggingsammenheng og ved konfigurasjon av modulene. Navnet må være unikt for alle moduler (også moduler fra andre undergrupper). |
| **classPath** | Fullstendig bane til oppstartsklassen for modulen. |
| **jarPath** | (Kun ved lasting av eksterne moduler) Banen til jar filen der modulen ligger. |

Fordi kjernen støtter lasting av flere moduler skal hver modul ha sitt eget <plugin> element under sin respektive gruppe. Modulene lastes inn i rekkefølgen beskrevet over. Det betyr at om en jar fil inneholder både en kommunikasjonsmodul og en oversettelsesmodul skal kun <plugin> elementet under ”clientCommunicators” inneholde et <jarPath> element. Når kjernen så kommer til ”resultTranslators” vil allerede jar filen være lastet.

Det er fult mulig og benytte config.xml til å konfigurere modulene. Ved å bruke metoden ”getConfigForPluginByName” i kjernen kan man hente ut konfigurasjonsinnstillingene beskrevet for en modul etter navnet på denne. Elementer som ikke er inkludert i Tabell C-4 blir ignorert av kjernen, men kan benyttes av modulene.

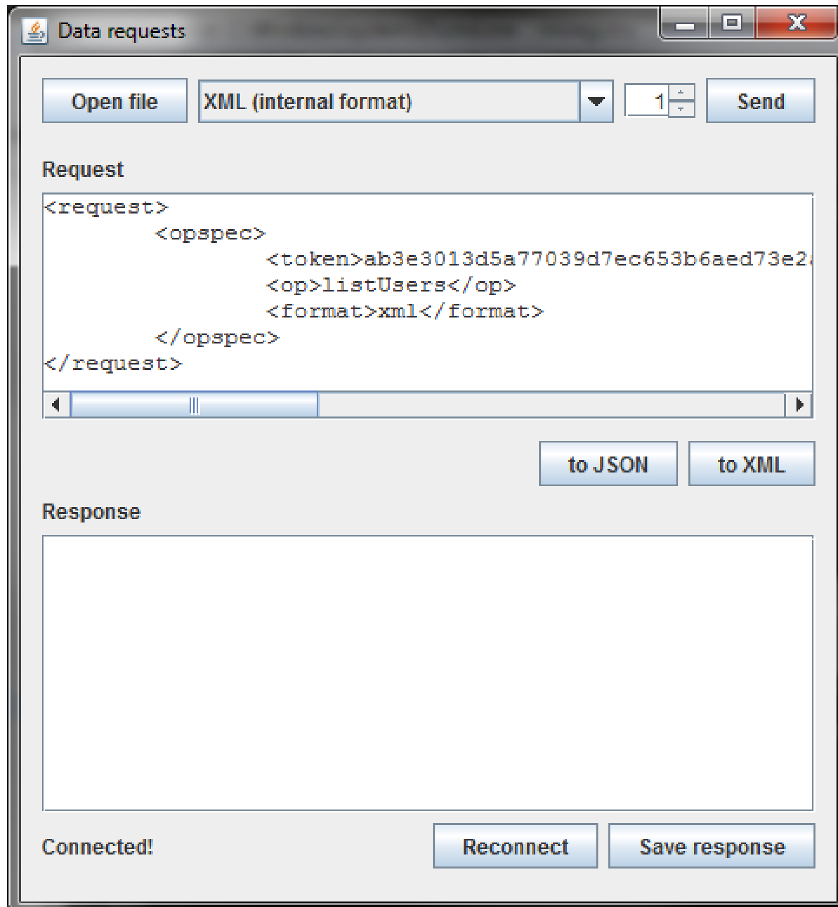
# Appendiks D Testing med RMI klienten og ferdige forespørselsdokumenter

Prototypen har allerede implementert noen ferdige funksjoner som kan benyttes under testing. Disse sammen med de nødvendige XML dokumentene er beskrevet i denne seksjonen. Det antas at både kjernen og RMI klienten kjører og er tilkoblet. Se Appendiks B for informasjon om hvordan man starter opp disse.



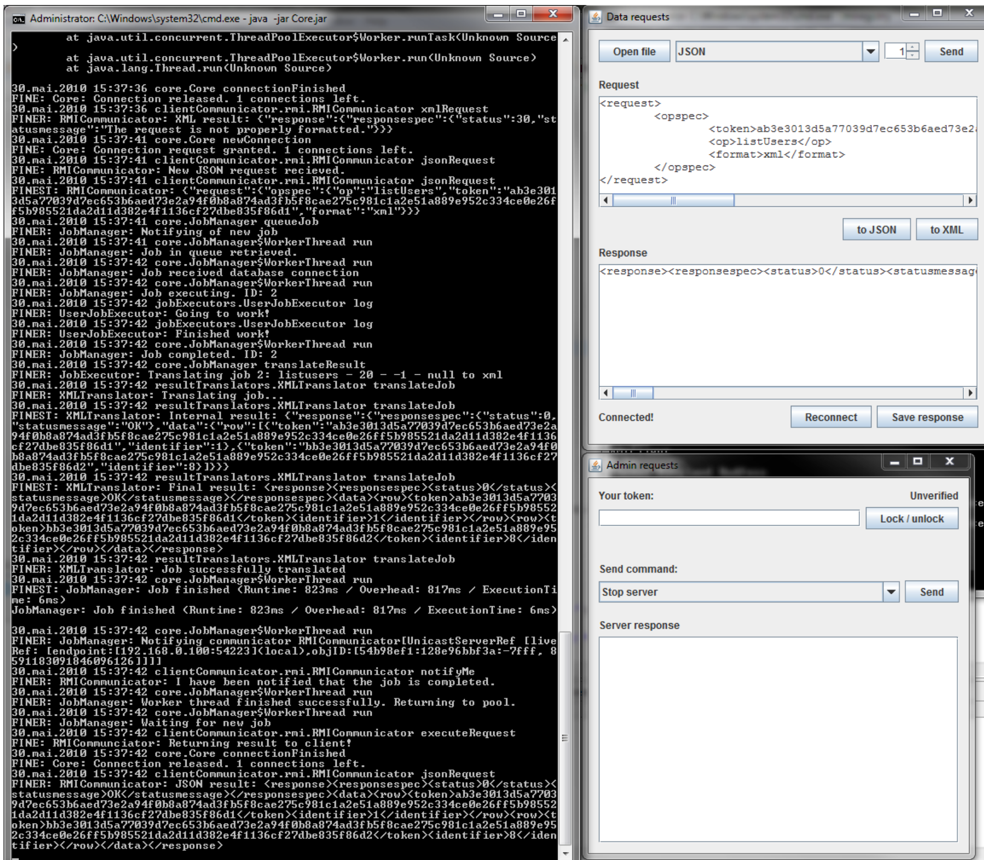

**Figur D-2: Kjernen og RMI klienten kjører.**

Når du har startet opp programmene skal de se omtrent ut som de gjør i Figur D-2. Til venstre vises loggen for kjernen mens RMI klienten er til høyre. For å utføre en forespørsel mot kjernen kan du enten skrive eller lime inn XML dokumentet i "Data request" vinduet (Request boksen) eller klikke "Open file" og bla fram XML filen med forespørselen du vil kjøre. Det er lagt ved et sett med ferdige forespørsler i XML dokumenter.




**Figur D-3: RMI klienten med en forespørsel klar for sending.**

Du kan endre på forespørselen ved å endre XML dokumentet i request boksen (Figur D-3). Før du sender sjekker du at det står XML (internal format) i typeboksen. Klienten kan sende to typer forespørsler, men den må definere typen for at kjernen skal tolke dem riktig. Hvis du ønsker å sende JSON forespørsel isteden kan du benytte knappen "to JSON" for å konvertere XML dokumentet til JSON format. Husk da å endre typeboksen til JSON før du sender. Boksen med tallet 1 bestemmer hvor mange ganger forespørselen skal sendes. Dette kan benyttes om du vil teste hvor raskt systemet svarer eller liknende. Forespørslene sendes i serie og ikke parallelt. Klikk "Send" når du er klar.




**Figur D-4: Kjernen og RMI klienten etter at en forespørsel er sendt.**

Når en forespørsel er sendt vil du se den bli behandlet i kjerneloggen og resultatet vist i "Response" boksen i RMI klienten.

Hvis du av en eller annen grunn blir nødt til å restarte kjernen mens RMI klienten kjører må du koble til på nytt. Fordi kjernen ikke forteller RMI klienten at den har blitt avslått vil klienten fortsatt tro at den er koblet til kjernen. Du kobler til på nytt ved å klikke på "Reconnect" knappen. Om alt er i orden vil statusfeltet vise "Connected!" hvis ikke vil det stå "Failed to reconnect!".

Om du vil sjekke at både klienten og kjernen er oppe kan du benytte administratorvinduet for klienten. Ignorer feltet "Your token" og velg kommandoen Ping, klikk så send. Du vil da se i loggen at kjernen har mottatt et "ping" mens i klienten vil det stå "Server pinged: …". Det betyr at du er koblet til og RMI grensesnittet virker.

# Appendiks E XML standard og støttede operasjoner

Denne seksjonen inneholder en liste over alle operasjoner som er støttet av prototypen. Du kan benytte RMI klient programmet for å teste disse operasjonene. For en beskrivelse på hvordan du gjennomfører testing av prototypen se Appendiks D.

Operasjonene kan kjøres ved hjelp av en av de ferdigdefinerte XML dokumentene. Disse følger en relativt enkel standard. Det er også mulig å benytte JSON dokumenter. JSON standarden tilsvarer den du vil få om du benytter RMI klientens konverteringsverktøy til å konvertere fra XML til JSON og er ikke beskrevet her.

Kommunikasjon med kjernen skjer ved hjelp av to typer dokumenter: Request og Response. Response dokumentets JSON konvertering er det interne formatet som benyttes inne i kjernen. Du kan benytte RMIClient programmet til å konvertere resultatdokumentet (XML) til det tilsvarende JSON formatet.

## E.1 REQUEST

Et request dokument er delt inn i tre seksjoner omsluttet av <request> og </request> tagene. <opspec> inneholder metainformasjon om operasjonen i elementene beskrevet i Tabell E-5:

**Tabell E-5: Beskrivelse av elementer i "opspec" tagen.**

| Element | Beskrivelse |
|---|---|
| **token** | En unik nøkkel som identifiserer brukeren som ønsker å utføre operasjonen. Token strengen benyttes for å tildele rettigheter til brukeren. |
| **op** | Navnet på operasjonen som skal utføres. For eksempel "listUsers" |
| **format** | Ønsket returformat på forespørselen. Muligheter i prototypen er: XML, JSON og i noen tilfeller CSV (Comma Separated Values). |
| **opdetails** | Dette er en underseksjon som kan inneholde ytterligere detaljer om operasjonen. For operasjoner som ikke trenger slike behøves ikke dette elementet. |

Den andre seksjonen i request dokumentet er <details> seksjonen. Ved handlinger som utfører endringer på et eller flere spesifikke elementer i databasen benyttes denne seksjonen til å identifisere disse elementene. I dokumentet "getNotesForItem_XML" inneholder <details> seksjonen informasjon om hvilket element man ønsker notatene for. Fordi seksjonen varierer avhengig av hvilken operasjon som utføres er det ikke definert noen faste tager her.

Den siste seksjonen er <data>. Den inneholder informasjon som skal endres eller opprettes. I dokumentet "createPerson_XML" inneholder <data> informasjon om personen som skal opprettes. Tilsvarende vil XML dokumentet som beskriver en forespørsel der et personobjekt oppdateres både inneholde en <details> seksjon der personobjektet identifiseres og en <data> seksjon der de nye verdiene til feltene i personobjektet ligger. Som i <details> seksjonen er det ikke definert noen faste elementer da disse også er avhengig av operasjonen.

```
}
```

### *E.1.1 Eksempel på request dokument på XML og JSON form*

```
<request>
	<opspec>

	<token>78ak56qte42kaq77i83rc6al276yqo756akvew7rt678
36k7785qkf8dkaf24tkvawgf8qw</token>
		<op>modifyUser</op>
		<format>xml</format>
	</opspec>
	<details>
		<identifier>1</identifier> <!-- Definerer
iden på personen som skal oppdateres -->
	</details>
	<data>
		<identifier>1337</identifier> <!-- Det er
mulig å endre identifier (tilsvarer elementid) -->
		<firstname>Ola</firstname>
		<lastname>Nordmann</lastname>
	</data>
</request>
```

```
{
	"request":{
		"opspec":{
			"op":"modifyUser",
			"token":"78ak56qte42kaq77i83rc6al276yqo7
			56akvew7rt67836k7785qkf8dkaf24tkvawgf8qw
			",
			"format":"xml"
		},
		"details":{
			"identifier":1
		},
		"data":{
			"lastname":"Nordmann",
			"firstname":"Ola",
			"identifier":1337
		}
	}
}
```

## E.2 RESPONSE

Response dokumentet definerer hvordan resultatet av en forespørsel skal se ut. Resultatet er alltid pakket inn i tagene <response> og </response>. Et responsedokument inneholder to underseksjoner: <responsespec> og <data>.

Responsedokumentet er et XML dokument som tilsvarer JSON formatet som benyttes som intern-resultat for jobbene. Ved å bruke RMIClient programmet kan du konvertere XML dokumentet som kommer i retur fra operasjonen til JSON og dermed se nøyaktig hvordan det interne formatet ser ut.

<responsespec> definerer informasjon om status for operasjonen og inneholder alltid to elementer.

**Tabell E-6: Beskrivelse av elementene i "responsespec" tagen.**

| Element | Beskrivelse |
|---|---|
| **status** | En statuskode som beskriver tilstanden til resultatet. Se Appendiks L for informasjon om disse. |
| **statusmessage** | En menneskelig lesbar statusmelding som beskriver statuskoden. |

<data> seksjonen inneholder de faktiske dataene fra forespørselen. Her varierer tagene og navnene avhengig av hvilken operasjon som utføres, men for resultater som lister opp data vil tagen <row> inneholder informasjon om en rad i listen.

### *E.2.1 Eksempel på response dokument på XML og JSON form*

```xml
<response>
	<responsespec>
		<status>0</status>
		<statusmessage>Ok</statusmessage>
	</responsespec>
	<data>
		<row>
			<firstname>Ola</firstname>
			<lastname>Nordmann</lastname>
		</row>
		<row>
			<firstname>Kari</firstname>
			<lastname>Nordkvinne</lastname>
		</row>
	</data>
</response>
```

```json
{
	"response":{
		"responsespec":{
			"status":0,
			"statusmessage":"Ok"
		},
		"data":{
			"row":[
				{
					"lastname":"Nordmann",
					"firstname":"Ola"
				},
				{
					"lastname":"Nordkvinne",
					"firstname":"Kari"
				}
			]
		}
	}
}
```

## E.3 STØTTEDE OPERASJONER

Under følger en liste med operasjoner som er støttet i prototypen og en beskrivelse av disse. Alle modulene som implementerer disse operasjonene følger med i kjernen og lastes ikke inn fra eksterne jar filer. For å se eksempler på dokumentene for hver operasjon se mappen ”Request templates” i den medfølgende kildekoden. Operasjonsnavnet skrives i <op> elementet på XML forespørsler uten mellomrom eller bindestrek.

**Tabell E-7: Liste over alle støttede operasjoner med beskrivelse.**

| Operasjonsnavn | Beskrivelse |
|---|---|
| **connectAccessRole-InstanceToUser** | Knytt en instans av en tilgangsrolle til brukeren og gir dermed brukeren de rettighetene instansen tillater. |
| **createAccessRole-Instance** | Lag en ny instans av en tilgangsrolle som så kan knyttes til brukere. En tilgangsrolleinstans er for eksempel en instans av rollen "courseParticipant" der også fagets id er inkludert og gir dermed brukeren tilgang som deltager i faget. |
| **createCourse** | Opprett et nytt fagobjekt. |
| **createNoteForItem** | Opprett et nytt notat for et element (en rad i en tabell). |
| **createPerson** | Opprett et nytt personobjekt (ikke det samme som bruker og gir ikke tilgang til systemet). |
| **createPersonRole** | Opprett en ny rolle for en person (ikke det samme som tilgangsroller). |
| **createPersonRoleInstance** | Knytt en person opp til en rolle (ikke tilgangsrolle). |
| **createTag** | Opprett en tag som så kan benyttes til å beskrive et element. |
| **createUser** | Opprett en ny bruker (gir tilgang til systemet). |
| **getNotesForItem** | Henter ut alle notater som er lagret for et element. |
| **getTagsForItem** | Henter ut alle tager som er knyttet til et element. |
| **listAccessRoles** | Lister opp alle tilgangsroller. |
| **listPersonRoles** | Lister opp alle personroller. |
| **listPersons** | Lister opp alle registrerte personobjekter. |
| **listUsers** | Lister opp alle brukere. |
| **removeAccessRole-InstanceFromUser** | Fjerner en instans av en tilgangsrolle for en bruker. Dette fjerner alle rettighetene brukeren fikk tildelt som medlem av tilgangsrollen. |
| **removeUser** | Fjern en bruker fra systemet. Fjerner også alle tilknytninger i tilgangsroller. |
| **tagItem** | Knytt en tag opp mot et element for å beskrive det. |

# Appendiks F Grensesnitt for kjernen

De ulike modulene i prototypen opererer uavhengig av hverandre. De behøver ingen forhåndskunnskap og kommuniserer kun indirekte gjennom kjernen og jobb-objektet. Kjernen basert på singleton prinsippene som betyr at det kun eksisterer en instans av kjernen (Core-objektet) ved kjøring. En modul som ønsker å kommunisere med kjernen kaller først metoden getCore. På det returnerte objektet kan den så kalle en av de ønskede metodene. Dette er de viktigste metodene som er implementert i Core klassen (CoreInterface grensesnittet):

```
public static CoreInterface getCore()
```

*Denne metoden returnerer en referanse til kjernen via grensesnittet CoreInterface. Det er dette som benyttes for omtrent all kommunikasjon mellom moduler og kjernen.*

```
public int getState()
```

*Forteller kjernens tilstand. Tilstanden er definert i Core klassen med konstantene:*

**Tabell F-8: Konstanter benyttet i kjernen for å definere tilstanden.**

| Navn | Beskrivelse |
|---|---|
| `public static final int STATE_INITIALIZING = 1;` | Kjernen er i ferd med å starte opp. |
| `public static final int STATE_INITFAILURE = 3;` | Det oppstod en feil under oppstart. Kjernen vil avslutte. |
| `public static final int STATE_RUNNING = 5;` | Kjernen kjører som normalt. |
| `public static final int STATE_TERMINATING = 10;` | Kjernen er i ferd med å avslutte. |

```java
public void queueJob(Job job) throws
JobManagerNotRunningException...
```

*Dette er den viktigste metoden i kjernen. Når en klient er tilkoblet en kommunikasjonsmodul og den har konstruert et jobb-objekt sendes inn i systemet via denne metoden. Denne metoden venter ikke til jobben er ferdig, men returnerer umiddelbart etter at jobben er plassert i køen. Modulen kan derfor ikke forvente at jobben er gjort når denne metoden er ferdig.*

*Denne metoden kan også kaste et unntak (exception) om det oppstår problemer underveis. Disse er beskrevet i Tabell F-9.*

**Tabell F-9: Unntak som kan forekomme ved kall til queueJob i Core klassen.**

| Navn | Beskrivelse |
|---|---|
| `JobManagerNotRunningException` | Hvis kjernen mottar en jobb før den har startet ordentlig kan det oppstå en situasjon der JobManagern som håndterer ressursfordeling og jobbkøen ikke har startet. I disse tilfellene vil dette unntaket bli kastet. |
| `QueueIsFullException` | Om køen har nådd sin maksimallengde (definert i config.xml) vil dette unntaket bli kastet. |
| `NoJobExecutorException` | Hvis jobben ønsker å utføre en oppgave som det ikke finnes en arbeidermodul for vil dette unntaket bli kastet. |
| `InternalErrorException` | Dette unntaket oppstår om det er forekommet en eller annen intern feil i kjernen. Dette kan tyde på større problemer. |

```java
public ConfNode getConfig()
```

*Denne metoden gir tilgang på konfigurasjonsinnstillingene for programmet. Konfigurasjonsinnstillingene er alle elementene definert i filen config.xml. Det er ikke mulig å endre konfigurasjonen mens programmet kjører. For detaljer om hvordan ConfNode klassen og relaterte hjelpeklasser virker se Appendiks K.*

`public ConfNode getConfigForPluginByName(String name)`

*Dette er en hjelpemetode for modulene. Hvis en modul ikke er interessert i annet en sine egne konfigurasjonsinnstillinger kan den kalle denne metoden med sitt eget navn. Resultatet blir en ConfNode med roten i de spesifikke innstillingene for modulen. Se beskrivelsen av konfigurasjonsfilene i Appendiks C for flere detaljer.*

`public Logger getPluginLogger()`

*For å forenkle logging av hendelser i programmet benyttes "Logger" rammeverket i Java (21). Modulene kan benytte seg av samme loggingsmekanisme ved å kalle denne metoden. Her returneres Logger-objektet som kan benyttes av modulene. Det er ikke det samme som Logger-objektet benyttet av kjernen og man har derfor mulighet til å definere detaljnivå uavhengig for de to objektene.*

`public boolean newConnection()`

*Når kommunikasjonsmoduler mottar forespørsler må disse registreres i kjernen. På den måten kan kjernen kontrollere antallet samtidige tilkoblinger og begrense disse om nødvendig. Når en kommunikasjonmodul ber om en ny tilkobling med denne metoden får den et ja/nei svar. Ja hvis det er en ledig tilkobling og nei hvis ikke. Fordi kjernen ikke har noen direkte kontroll over tilkoblingene må modulen også si ifra når den er ferdig med en tilkobling. Det gjøres via metoden "connectionFinised".*

`public void connectionFinished()`

*Når en kommunikasjonsmodul har returnert et resultat til klienten og klienten kobler fra må den si ifra til kjernen om at den er ferdig med en tilkobling. Dette gjøres via denne metoden. Hvis modulen ikke sier ifra vil det føre til at kjernen fortsatt tror tilkoblingen er i bruk og vil til slutt forhindre nye tilkoblinger ved svare nei på forespørsler til "newConnection" metoden.*

# Appendiks G Kommunikasjons-moduler

## G.1 Formål og beskrivelse

Kommunikasjonsmodulene er ansvarlig for å (naturlig nok) kommunisere med klientene ved hjelp av ulike protokoller. En kommunikasjonsmodul kan for eksempel være ansvarlig for å ta imot forespørsler i form av XML eller JSON dokumenter via RMI (se RMICommunicator modulen) protokollen for java. Utover grensesnittene som må benyttes er det ikke definert noen direkte krav til modulene. Dette er fordi protokollene kan måtte bli implementer på meget forskjellige måter.

## G.2 Grensesnitt

For kommunikasjonsmoduler finnes det to grensesnitt. Dette gjør det mulig for modulene å ha en administrerende klasse og andre arbeiderklasser under denne. Administrasjonsklassen er ansvarlig for å starte opp og avslutte alle arbeiderne og gjør det derfor enkelt for kjernen å kontrollere dem.

Administrasjonsklassens grensesnitt heter "ClientCommunicatorManager" og har følgende metoder:

```
public void start()
```

*Denne metoden kalles av kjernen etter at modulen er lastet. Det er helt opp til modulen selv hvordan denne metoden implementeres, men når den returnerer bør modulen være klar til å motta forespørsler fra klienter. Kjernen antar at modulen har startet opp og er klar når denne metoden returnerer.*

```
public void stop()
```

*Når kjernen skal avsluttes vil den si ifra til alle kommunikasjonsmodulene. Dette gjøres via denne metoden. Modulene bør da si ifra til eventuelle tilkoblede klienter om at kjernen avslutter og at oppgaven dermed ikke kan utføres. Kjernen antar at når denne metoden returnerer har kommunikasjonsmodulen avsluttet eventuelle ekstra tråder og avbrutt all kommunikasjon med klienter.*

Grensesnittet for de ulike arbeidermodulene heter "ClientCommunicatorInterface". Det inneholder kun en metode som benyttes til å fortelle modulen at resultatet av jobben er klart. Som med administrasjonsklassen er det også her helt opp til modulen hvordan den implementer alt bakenforliggende for denne metoden, men tanken har vært at det opprettes en ny tråd for hver klient som kobler seg til. Denne tråden gjør arbeidet som er nødvendig for å gjøre klar jobb-objektet og så venter den til den blir vekket opp av en arbeidstråd i JobManager-klassen.

```
public void notifyMe()
```

*Når en jobb er klar eller har feilet kalles denne metoden fra en arbeidstråd (WorkerThread) i JobManager-klassen. Tanken med denne metoden er at den skal vekke opp en tråd i kommunikasjonsmodulen som så kan returnere resultatet tilbake til klienten. Dette er derimot ikke et krav og modulen kan selv velge hvordan den ønsker å implementere denne funksjonaliteten. Uansett vil kjernen (og arbeidstråden i JobManager) anta at klienten vil få beskjed når denne metoden er kalt.*

# Appendiks H Arbeidermoduler

## H.1 FORMÅL OG BESKRIVELSE

Arbeidermodulene er den delen av koden som faktisk utfører forespørslene fra klientene mot databasen. De er ansvarlig for at jobbene blir gjort og at sikkerheten ivaretas. For å forenkle denne oppgaven er det laget noen hjelpeklasser for administrasjon av brukere og feilrapportering. Du finner mer informasjon om disse i Appendiks K. I tillegg til dette er også utførelsen av oppgavene tilrettelagt av kjernen og arbeidermodulen kan derfor gjøre følgende antagelser:

- Jeg har min egen arbeidstråd som jeg har for meg selv.
- Når oppgaven er gjort og resultatet satt i jobb-objektet behøver jeg kun å returnere fra "doJob"-metoden og kjernen vil ta seg av resten.
- Jeg får et ferdig definert grensesnitt mot databasen i form av et DBConnection objekt.
- Brukerens identitet er blitt verifisert og ved hjelp av objektet userData i Job-klassen kan jeg finne ut det jeg trenger for å kontrollere rettighetene til brukeren.
- Alle detaljene som trengs for å utføre oppgaven finnes i Job-klassen som jeg får en instans av gjennom "doJob"-metoden.
- Jeg behøver ikke å bry meg om formatet på det endelige resultatet og trenger kun å lagre internresultatet i jobben med JSON objekter.
- Jeg behøver ikke avslutte databaseforespørsler (statements) når jeg er ferdig. Dette gjøres for meg av JobManager-klassen når jeg returnerer fra "doJob"-metoden.

Disse antagelsene gjelder kun i forbindelse med kommunikasjon med hoveddatabasen. Hvis arbeidermodulen kobler seg til andre systemer eller databaser må den selv sørge for å rydde opp og holde orden.

Det er definert et bestemt sett med operasjoner som arbeidermodulen må gjøre hver gang den får en oppgave:

1. Sjekk at brukeren har tilstrekkelig rettigheter til å utføre operasjonen (bruk UserData objektet).
2. Utføre oppgaven.
3. Sett resultatet i Job-klassens "internalResult" objekt.

Når jobben er gjort eller om den feiler må også modulen sørge for å oppdatere statusen på jobben. Hvis ikke dette gjøres vil JobManager-objektet kunne tolke jobben som ikke utført selv om den kan være det. Se Appendiks J for detaljer om Job-klassen og hvordan denne skal benyttes.

## H.2 GRENSESNITT

I grensesnittet som brukes for arbeidermoduler kalt "JobExecutorInterface" er det kun definert en metode, men det er viktig at hovedklassen[1] også implementerer den statiske funksjonen "jobsICanDo". Uten denne vil ikke kjernen kunne finne ut hvilke oppgaver modulen kan utføre.

```
public static String[] jobsICanDo()
```

*Denne metoden benyttes en gang under lasting av modulen. For at kjernen skal kunne delegere oppgaver til de ulike arbeidermodulene er den nødt til å vite hvilke operasjoner modulene støtter. Den kaller da denne metoden og får igjen en liste (array) med alle operasjonsnavnene modulen kan utføre. Hvis flere moduler melder fra at de kan utføre samme operasjon vil den som ble lastet sist få tildelt oppgavene mens de andre blir lastet, men blir også ignorert. Hvis denne metoden mangler vil kjernen logge en feil og modulen vil ikke bli lastet inn.*

```
public synchronized void doJob(Job job)
```

*Dette er den egentlige kjernen i hele systemet. Når en jobb skal utføres opprettes et jobb-objekt som omsider finner veien hit. Når arbeidermodulens doJob-metode blir kalt skal oppgaven utføres og resultatet plasseres i det medfølgende jobb-objektet. Arbeidermodulen må sørge for å utføre de operasjonene som er nevnt i forrige seksjon og returnerer når den er ferdig.*

[1] Hovedklassen er den klassen som lastes inn når kjernen starter. Det er altså den klassen som er definert i config.xml filen.

# Appendiks I Oversettelsesmoduler

## I.1 FORMÅL OG BESKRIVELSE

For å gjøre det mulig for klientene å definere et ønsket returformat på resultatet og dermed å forenkle kommunikasjon med mange ulike systemer benyttes oversettelsesmoduler. Dette er et sett med moduler som kan konvertere det interne formatet (JSON) til et annet format. Med prototypen følger det to slike moduler: XML og CSV. Oversettelsesmodulen kan gjøre noen av de samme antagelsene som arbeidermodulene:

- Kjører fortsatt i en egen tråd.
- Har fortsatt tilgang på databasen og behøver ikke lukke databaseforespørsler (”statements”).
- Internresultatet for jobben ligger i jobbens ”internalResult” felt.

Når en jobb kommer til en oversettelsesmodul skal den utføre følgende oppgaver:

- Oversett resultatet til det ønskede formatet (hentes enten fra toFormat parametere eller Job-klassens resultFormat felt).
- Oppdatere status på jobben ved fullført eller feilet konvertering.

## I.2 GRENSESNITT

På samme måte som med JobExecutor modulene er det definert en metode i grensesnittet (”ResultTranslatorInterface”) og en statisk metode som hovedklassen må implementere.

```
public static String[] iCanTranslateTo()
```

*Når oversettelsesmodulene lastes benytter kjernen denne metoden til å finne ut hvilke formater modulen kan oversette til. Det antas at alle kan oversette fra det interne JSON formatet. Resultatet av denne metoden er en liste (array) med formatnavnet på oversettelser. Når en forespørsel sendes skal det ønskede resultatformatet stå i <format> elementet og må være det samme som meldes fra denne metoden (case-sensitive).*

```
public void translateJob(Job job, String toFormat)
```

*Dette er metoden som kalles av arbeidertråden (WorkerThread) når jobben er gjort og resultatet skal konverteres. Det er helt opp til modulen hvordan den vil løse konverteringsprosessen. Den kan benytte databasen eller andre systemer om nødvendig så lenge den rydder opp etter seg.*

# Appendiks J Job-klassen

Job-klassen (også kalt jobb-objektet) er klassen som beskriver oppgaven som skal utføres og består av et sett med felter og metoder som fasiliterer denne jobben. Denne seksjonen vil gå gjennom alle ”public”[1] felter og metoder som kan benyttes av andre klasser i systemet. Noen av feltene i klassen er definert som ”protected”. Det betyr at de kun er synlig for andre klasser innenfor den samme pakken. Dette er gjort for å gjøre det enklere for kjernen og administrere jobbene.

## J.1 FELTER

Noen felter er definert med nøkkelordet ”final”. Det betyr at de tolkes som konstanter og kan ikke endres etter at de først er satt. I Job-klassen finnes det noen slike felter med navn STATE_ etterfulgt av en tilstand. Disse feltene beskriver bestemte tilstander jobben kan ha og lagres i feltet ”state”. Om en modul vil endre eller lese tilstanden til jobben benyttes da dette feltet i kombinasjon med STATE konstantene.

[1] ”Public” betyr felter og metoder som er tilgjengelig også utenfor klassen. Siden det kun er disse som kan benyttes av andre klasser vil de bli gjennomgått mens ”private” felter og metoder hoppes over.

**Tabell J-10: Tilstandskonstanter for jobber.**

| Navn | Type | Beskrivelse |
|---|---|---|
| **STATE_NEW** | Const. int | Jobben er akkurat opprettet og blitt satt i køen. |
| **STATE_RUNNING** | Const. int | Jobben er tatt ut av køen og kjører. |
| **STATE_RESULT_SET** | Const. int | Jobben er utført og internresultatet er blitt satt. Settes av en JobExecutor-modul. |
| **STATE_RESULT_TRANSLATED** | Const. int | Jobbens resultat er oversatt og lagret. Settes av en ResultTranslator-modul. |
| **STATE_COMPLETE** | Const. int | Jobben er ferdig og oversatt. Settes av JobManager-klassen. |
| **STATE_CANCELLED** | Const. int | Jobben har blitt avbrutt (køen er full, manglende oversetter eller arbeider). |
| **STATE_TIMEOUT** | Const. int | Jobben er avbrutt fordi den brukte for lang tid. |
| **STATE_FAILED** | Const. int | Jobben har feilet av en eller annen årsak. Se jobbens ”statusCode” og ”statusMessage” felt for detaljer. |

**Tabell J-11: Vanlige felter i Job-klassen.**

| Navn | Type | Beskrivelse |
|---|---|---|
| **jobId** | Const. int | Jobbens unike ID. En ID tildeles når jobben opprettes og er kun unik så lenge applikasjonen kjører. Ved restart begynner IDen fra 1 igjen. |
| **dbConnection** | DBConnection | Referanse til databasetilkoblingsobjektet for denne jobben. Dette feltet settes av JobManager-klassen (se K.3). |
| **userData** | UserData | Referanse til et UserData objekt som beskriver brukeren som startet jobben (se K.1). |
| **state** | int | Jobbens tilstand (se tilstandskonstantene over). |
| **request** | JSONObject | Forespørselen konvertert til et JSON-objekt. |
| **resultFormat** | String | Ønsket format på resultatet. |
| **internalResult** | JSONObject | Det interne resultatet før det er konvertert til det ønskede endelige formatet. |
| **result** | String | Det endelige resultatet etter konvertering. |
| **error** | ErrorObject | Referanse til en instans av ErrorObject som beskriver en eventuell feil. Dette feltet er null om ingen feil har oppstått. |
| **statusMessage** | String | Statusmelding som beskriver jobbens tilstand og eventuelle feil. |
| **statusCode** | int | En kode som beskriver jobbens tilstand i større detalj enn tilstandskodene. |

```java
public Job(ClientCommunicatorInterface communicator, String
token, String jobTask, JSONObject request, String
resultFormat)
```

*Konstruktøren (constructor) for Job-klassen. Denne initialiserer jobb-objektet og tar inn følgende parametere:*

- ***communicator:*** *Referanse til kommunikasjonsmodulen som skal informeres når jobben er gjort. Denne klassen må implementere ClientCommunicatorInterface grensesnittet.*
- ***token:*** *Brukerens "token". Dette er en tekststreng som brukes til å identifisere brukeren som vil utføre oppgaven. Benyttes til å opprette et UserData objekt for jobben.*
- ***jobTask:*** *Navnet på oppgaven jobben skal utføre.*
- ***request:*** *Hele forespørselen lagret som et JSONObject slik at JobExecutor-klassen kan få tak i nødvendig informasjon.*
- ***resultFormat (ikke påkrevd):*** *Ønsket format på resultatet. Hvis den ikke er definert antas JSON som format.*

```java
public String getJobTask()
```

*Returnerer navnet på oppgaven som skal utføres.*

```java
public String getToken()
```

*Returnere tokenen for brukeren som opprettet jobben.*

```java
public String toString()
```

*Returnere en tekststreng med informasjon om jobben på formatet:*
*"[id]: [oppgavenavn] - [tilstand] - [statuskode] - [statusmelding]"*

```java
public String getTimes()
```

*Returnerer tiden jobben har vært i systemet i millisekunder som tekststreng på formatet:*
*"Total execution time: [tid]ms"*

# Appendiks K Hjelpeklasser

En hjelpeklasse er her definert som en klasse som ikke utfører noe direkte arbeid, men fasiliteter oppgavene systemet skal løse. Denne seksjonen vil gå gjennom noen felter og metoder å beskrive disse klassene og hvordan de kan brukes, men en fullstendig beskrivelse av alle metoder og felter finnes i kommentarene i kilden.

## K.1 BRUKERDATA (USERDATA)

Fordi enhver forespørsel gjøres av en bruker og det er nødvendig å sjekke at denne brukeren har tilstrekkelig rettigheter til å utføre operasjonen har jeg laget tre klasser som forenkler denne prosessen. Disse heter: UserData, UserRole og RoleOptionList. UserData benyttes som overordnet klasse og styrer de andre to. UserRole inneholder informasjon om alle tilgangsrollene til en bruker mens RoleOptionList holder på alle verdiene en enkelt attributt i rollen kan ha. De benytter seg av kjernens databasetilkobling (en egen tilkobling som ikke er tilgjengelig for modulene) til å hente ut informasjon om en bruker basert på enten en token eller en id. Når en jobb opprettes lages det automatisk et slikt objekt som knyttes til jobben.

Et UserData objekt kan opprettes ved å kalle en av disse metodene statisk på UserData objektet:

- `constructFromId(int id)`
  Oppretter et UserData objekt basert på en id. Returnerer enten en ny instans av UserData eller kaster et unntak om iden ikke finnes.
- `constructFromToken(String token)`
  Oppretter et UserData objekt basert på en token. Returnerer enten en ny instans av UserData eller kaster et unntak om tokenen ikke finnes.

Når UserData objektet er opprette kan man benytte det til å sjekke om brukeren har tilstrekkelig rettigheter for en operasjon ved å sjekke om han/hun er administrator (`isAdmin()`) eller har en eller flere nødvendige roller med riktige verdier på feltene for rollene.

Eksempel: En bruker prøver å finne informasjon om et fag med id **12**. Han er ikke administrator, men er en student som tar faget. For å sjekke at brukeren har tilstrekkelig rettigheter (har en tilgangsrolleinstans med alternativet courseId = 12) kan du gjøre følgende:

```
public boolean hasRights(){
	UserData ud = job.userData; //1

	//Check if the user is a member of a course
	//and if not return false
	//2
	if ( !ud.hasRole("courseParticipant") ) return false;


	//Grab the role
	UserRole ur = ud.getRole("courseParticipant"); //3

	//Check if the role has a courseId value with the
	courses id
	//4
	if ( !ur.hasOptionWithValue("courseId", theCourse_id) )
		return false;

	return true;
}
```

Hva det er som skjer i denne koden:

1. Hent ut UserData objektet fra jobben (et synlig felt et annet sted i klassen).
2. Sjekk om brukeren har tilgangsrollen ”courseParticipant”. Hvis ikke avbryt og returner ”false”.
3. Hent UserRole objektet som tilsvarer tilgangsrollen ”courseParticipant”.
4. Sjekk at brukeren har tilgang til riktig fag ved å se om en av verdiene for denne rolletypen er den samme som IDen til faget.

Det er viktig ikke bare å sjekke at brukeren har den riktige tilgangsrollen, men også at han/hun har de attributtene for rollen fylt ut med riktig verdier. Fordi

rollen ”courseParticipant” omhandler fagtilknytning vil den i omtrent alle tilfeller ha flere verdier for attributten ”courseId” siden personen sannsynligvis tar flere enn et fag.

## K.2 KONFIGURASJONSINNSTILLINGER (CONFNODE)

ConfNode er et abstraksjonslag oppå et XML dokument. Det forenkler uthenting av informasjon fra konfigurasjonsfiler ved å implementere en adapter mellom dokumentet og koden som gjør det mulig å spørre etter konfigurasjonsinnstillinger ved for eksempel å skrive:

```
config.get("logging.enableCoreLogger").asBoolean()
```

Her spørres det etter konfigurasjonsalternativet ”enableCoreLogger” under elementet ”logging” i en streng. Man kan gå dypere ved å legge på flere punktum og navn på elementer. Legg merke til at rotnoden (<core>) ikke er med i forespørselen. Hvis man ikke ønsker å bruke verdien direkte, men lagre den for senere bruk kan man kutte ut asBoolean() delen av linjen over. Da returneres et ConfNode-objekt som man kan spørre videre på. Alternativt til linjen over kunne man også gjort:

```
config.get("logging").get("enableCoreLogger").asBoolean()
```

Dette kallet gjør den samme operasjonen som over, men benytter ConfNoden som hentes ut mellom ”logging” og ”enableCoreLogger” kallene.

Den siste delen av koden `asBoolean()` definerer at resultatet ønskes som en boolsk verdi. Det er fem slike hjelpemetoder som gjør om resultatet så det enklere kan benyttes i for eksempel ”if” setninger eller lagres i variabler av bestemte typer:

- **asBoolean:** Konverterer verdien til en boolsk verdi (true/false). Verdien er opprinnelig en tekststreng og vil konverteres til true hvis den er ”true”, ”yes” eller ”1”. Alle andre verdier returneres som false.
- **asInt:** Konverterer verdien til et heltall hvis mulig.
- **asDouble:** Konverterer verdien til et desimaltall (dobbel presisjon) hvis mulig.
- **asLong:** Konverterer verdiene til et langt heltall (long)
- **asArray:** Benytter regular expression til å dele verdien opp og returnerer det resulterende arrayet. Eksempel: Gitt teksten ”a.b.c.d” og utrykket ”/\./” vil splitte strengen opp ved hvert punktum og returnere et array med elementene {a, b, c, d}.

For å benytte denne klassen kaller man metoden ”makeFromXML” med banen til xml dokumentet. Denne returnerer en ConfNode instans for rotnoden i dokumentet som så kan benyttes videre.

## K.3 DATABASETILKOBLING (DBCONNECTION)

De aller fleste operasjonene som systemet skal utføre behøver en tilkobling til databasen. For å forenkle denne prosessen og samtidig unngå mange feilkilder har jeg laget en klasse som kan benyttes ved kommunikasjon med databasen. Denne klassen heter ”DBConnection” og er relativt enkel.

Når JobManager-klassen starter opp kobler den seg opp mot databasen et definert antall ganger ved hjelp av konfigurasjonsinnstillingene. Den oppretter så like mange DBConnection instanser som holder på hver sin egen tilkobling. Når en JobExecutor eller ResultTranslator ønsker å koble til databasen kaller den en av metodene på DBConnection-objektet og får tilgang til et ”statement” objekt som benyttes for utveksling av informasjon. DBConnection holder også på referanser til disse statement-objektene slik at den kan rydde opp etter modulene før den returneres til kjernen. Denne klassen sørger dermed for mindre arbeid for modulene fordi de ikke behøver å tenke på verken hvordan de skal koble seg til databasen eller avslutting og opprydding etter databaseoperajonene.

De to metodene som kan benyttes er:

- **newStatement:** Returnerer et nytt ”statement”-objekt som kan benyttes til kommunikasjon med databasen.
- **newPreparedStatement:** Returnerer et nytt ”statement”-objekt basert på den gitte spørringen. Et prepared statement er en ”halvveis gjennomført” spørring der kun verdiene mangler. Med en slik spørring elimineres problemer med SQL Injection angrep der en angriper prøver å endre databasen ved å sette inn SQL tekst i forespørselen. Mer om Prepared Statements finnes her (22).

## K.4 ERROROBJECT

ErrorObject er en veldig spesialisert klasse som sørger for konsistente feilresultater. Når en feil oppstår er det opp til modulen jobben er i for øyeblikket å informere om feilen ved å sette status og feilmeldinger. Det er definert en del faste feilkoder som kan benyttes (se Appendiks L) til dette og for at alle modulene skal slippe å definere sine egne feilmeldinger kan de benytte ErrorObject klassen til å fylle ut de relevante feltene i Job-klassen for seg.

Metoden getErrorForCode er definert med tre nivåer:

- Kun kode og jobb: Brukes ved en av de standardiserte feilene som tilgangsfeil eller brukeren finnes ikke. Denne definerer både feilkode og melding.
- Kode, jobb og feilmelding: Benyttes for mer avanserte feil der modulen selv ønsker å definere en feilmelding.
- Kode, jobb, feilmelding og intern feilkode: Benyttes for feil der modulen ønsker å definere både feilmeldingen og en egen intern feilkode. Kan brukes for debugging av moduler og ved avanserte feil i modulen som ikke kjennes igjen i kjernen.

# Appendiks L Statuskoder for resultater

**Tabell L-12: Statuskoder som kan returneres fra kjernen.**

| Statuskode | Beskrivelse |
|---|---|
| **0** | Operasjonen ble vellykket utført. |
| **10** | Ikke tilstrekkelig rettigheter |
| **20** | Token finnes ikke (bruker finnes ikke) |
| **30** | Forespørselen er ikke gyldig formatert. |
| **40** | Systemet er opptatt (ikke plass til flere jobber) |
| **42** | Kan ikke utføre denne jobben (ingen arbeider registrert for oppgaven) |
| **44** | Kan ikke utføre jobben. Det finnes ingen oversetter for det formatet. |
| **50** | Jobben timet ut (var for lenge i systemet) |
| **60** | Jobben ble kansellert før den fikk starte |
| **100** | Intern serverfeil. Jobben kunne ikke utføres. |
| **200** | Udefinert feil. Se melding for detaljer. Kan komme fra en JobExeutor som mangler data eller har støtt på en annen uspesifisert feil. |