IDATT2105 - Full-stack applikasjonsutvikling - ØVING 2

1 Preface

Vi skal i denne øvingen implementere et REST-API fra controlleren og ned *til* repo-/DAO-nivå. Som du kanskje har fått med deg, har jeg brukt disse to design-mønstrene litt om hverandre, og dette er fordi det er prinsipielt tre måter å tenke på repo og DAO på.

1. Det mange vil anse som mer eller mindre den kanoniske måten å implementere repositories på, kommer fra Eric Evans' bok ved navn Domain Driven Design. I korte trekk, her blir et repository ansett som det som skal innkapsle database-/persistensoperasjoner, og er det nederste laget over selve databasen. Dette betyr at repoene som implementeres har en tendens til å bli tykkere/rikere, dvs. de inneholder mer kode og funksjonalitet, enn det man gjerne vil gjøre i et tilfelle med DAO. Ofte har en databaseoperasjoner som går på tvers av domenemodellen, og i dette tilfellet behandles dette i repoene. Servicene vil i større grad forholde seg til ett (eller ytterst få) repoer i en slik sammenheng, når de trenger å få skrevet/hentet ting fra databasen.

2. Den andre måten, «DAO-måten»[1], er å implementere en DAO i stedet for et repo. Hovedforskjellen på dette og et repo, er at en DAO er mye mer anemisk og forholder seg (så langt det lar seg gjøre) til ett, eller ytterst få, domeneobjekter. Har man f.eks. en `Person`-klasse, vil man ha en korresponderende `PersonDao`, og denne DAOen skal behandle kun `Person`-objekter. Siden realiteten sjeldent lar oss jobbe så separert, vi har tross alt avhengigheter domeneobjektene i mellom, og oppdatering av ett av dem kan også føre til oppdatering av et annet, vil en i slike tilfeller må pushe ansvaret for å behandle operasjoner som går på tvers av DAOer opp til servicene. Dette bryter med prinsippet om å innkapsle persistens-/databaseoperasjoner i datalaget, og gjør det vanskelig å finne en definitiv grense på hva som har ansvaret for hva.

3. Den tredje måten, er å ha en kombinasjon av DAO og repo. Man har da relativt spinkle DAOer som utfører operasjoner mot databasen med et veldig avgrenset område, forhåpentligvis ikke mer enn ett domeneobjekt, mens om en har behov for mer avanserte operasjoner som å sy sammen flere databasekall til noe som kan benyttes lenger oppe i lagene, legger man et repo over, og lar dette repoet benytte flere DAOer. Man har da (fra et konseptuelt ståsted), en ganske clean separation of concerns, hvor alt som har med persistens/databaseoperasjoner å gjøre, blir liggende nede i datalaget. Denne måten, på samme måte som i DDD-måten, vil føre til slankere servicer.[2]

[1] Det er mange som vil hevde at dette er akkurat det samme som et repo, og er som jeg (forhåpentligvis) har nevnt i forlesningen, det som gjør det litt vanskelig å vite hva noen mener når de snakker om DAO/repo. Å krangle om semantikken rundt dette er ikke noe vi skal bruke tid på, for jeg ikke anser det som viktig. Det viktige er at du kjenner til de forskjellige måtene omtalt her.

[2] Dette er også et problem som kan oppstå blant servicer, hvor man får ekstremt feite servicer som har alt

2 Oppgave

Vi skal implementere et REST-API for å opprette, oppdatere, hente ut info om, og slette forfattere og bøker. Det vil si at en trenger metoder for å støtte dette i kontrolleren(e), tilhørende service-klasser, og repo. Merk: siden vi ikke har rørt persistens ennå, skal vi *ikke* programmere et reelt repo i denne omgangen, men heller bare implementere dummy-/stub-metoder for dette. Det vil si at repo-klassene ikke trenger å returnere annet enn noen faste svar (objekter/lister/whatever needed). Av de nevnte måtene i prefacet å implementere repo/DAO på, kan du selv velge hvilken en du forestiller du ville ha brukt om du skulle ha implementert alle lagene helt ned til databasen, men velg én av dem. Grunnen til at jeg vil du skal gjøre dette, er at det vil sannsynligvis ha en påvirkning for hvordan du implementerer servicene dine.

1. Skriv klassene for bok og forfatter, med tillhørende klassevariabler du mener er fornuftig. En forfatter kan ha skrevet flere bøker, og en bok kan være skrevet av flere forfattere. En forfatter har en adresse (som også skal implementeres som egen klasse).

2. Implementer metoder for å opprette, oppdatere, og slette forfattere og bøker i kontrolleren(e). Merk her: ved å oppdatere en forfatter, har man også mulighet til å oppdatere hvilke bøker forfatteren er knyttet til, og dette fremtvinger et lite spørsmål: dersom boka ikke eksisterer, bør man da ha mulighet til å sende med informasjon om boka, slik at denne opprettes først om den ikke eksisterer, og deretter knytter forfatteren til den? Hva sier prinsippene rundt REST om dette? Hvordan er det med adressen til forfatteren?

3. Implementer søk. Man skal kunne søke etter forfatter på navn. Man skal også kunne søke etter bøker basert på navnet på boka, men i tillegg skal man kunne søke etter bøker basert på forfatter. Hvordan blir dette i forhold til separering av klassene? Er det lurt å legge søket etter bøker basert på forfatter, i «stacken» (beklager, fant ikke bedre ord) for bøker, når vi allerede vet akkurat hvilke bøker som tilhører en forfatter? Er det andre måter å løse dette på, som tillater å holde koden ren og ha god struktur?

4. For hvert søk som gjøres, skal det også logges i servicen hvilket søk som ble gjort, og parameterene for dette søket. Bruk forskjellige logg-nivåer, for å få sjekket at det også fungerer.

Siden REST i stor grad er basert rundt at alle objektene har IDer, er du nødt til å programmere inn surrogat-IDer for objektene dine, slik at du har noe å bruke i REST-kallene.

Sjekk at koden din fungerer slik den skal vha. Postman eller curl.

Om du står fast på noe, kan jeg anbefale siden `https://www.baeldung.com` som har gode forklaringer og mange korte, men praktiske, eksempler på det meste når det kommer til Spring Boot (og mye annet). I tillegg har vi også læringsassistentene som er der for å hjelpe deg, om du trenger det.

for mye ansvar (og kode). Man kan da følge samme prinsippet med å legge til en «super-service», som dekker flere nedenforliggende servicer og tilbyr viewet noe mer fornuftig. Det er et design pattern som kalles facade, men prinsippet er akkurat det samme.