Test af Gutenberg Data Mining

Gruppe Firebug

Mathias Meldgaard Pedersen

Jonathan Egebak Carlsen

Joachim Dittman Jepsen

Nikolai Hansen

Vi vil lære at opsætte en Continuous integration chain (Travis), for at teste vores kode automatisk efter vi har merged vores forskellige versioner.

Hvad er en CI chain

En CI chain er et miljø man kan sætte op til at teste sit projekt når man merger koden.
Alt efter om testen fejler eller går igennem, så kan man deploye eller give udviklingsteamet besked om at der er noget galt med den mergede kode.

Hvad har vi gjort

I vores sprint 0 opsatte vi en basic CI chain med travis, som der sendte en mail til os med resultatet af builded.
Da vi skulle lave vores integration test, udvidede vi vores CI chain til at opsætte en PostgreSql database.

Hvad fik vi ud af at bruge en CI chain

Ved at have benyttet en CI chain, fik vi automatisk testet om vores merget kode virkede og vi kunne oprette et test database testmiljø.
Derudover havde vi tænkt os at få Travis til at deploye vores løsning hvis testen gik igennem.

Vi vil bruge Test driven development for at se om vores kodestandard bliver bedre når vi laver koden igennem test.

Hvad er TDD

TDD er en udviklingsproces hvor man udarbejder kravene til test cases.
Man starter med at lave en test der fejler, den fejler fordi det komponent man tester ikke er lavet, derefter laver man komponentet som man har lyst, når testen så lyser grønt, refaktorerer man koden f eks. så den lever op til nogle kodestandarder man har aftalt på forhånd i teamet.

Hvordan har vi brugt det

Vi har desværre ikke arbejdet særlig meget med TDD da vi har været meget i en eksperimenterende process igennem hele forløbet.

Hvad ville vi havde fået ud af det

Ved at arbejde med TDD sørger man for at hvert komponent bliver småt og det får vi visse fordele ud af:

- Reduced debugging effort - når der er Test failures, så er det nemmere at tracke fejlene
- Self-documenting tests - små test cases er nemmere at læse og forstå.

Derudover laver man også koden ud fra testen og ikke omvendt, og det giver et mere kvalitetsfuldt system.

Vi vil bruge Unit testing for at lære at test enkle komponenter fra resten af programmet, f eks. ved brugen mock mockito.

Hvad er Unit testing

En unit test er en test hvor man tester enkelte komponenter.
For at kunne Unit teste komponenter som er afhængig af andre komponenter kan man benytte
Mock eller stubs.

Hvad har vi gjort

For at kunne Unit teste har vi benyttet Stubs og Mock - Mockito for at isolere hvert komponent.
Vi valgte at lave en Stub af en enkelt metode, da vi skulle aflæse og indsætte bøgerne og finde byerne i de enkelte bøger.
Grunden til at vi valgte en stub i denne case, er at i det kombonent vi ville teste, havde en hjælpe metode som har en BufferedReader som parameter.
Ellers har vi benyttet Mockito til at isolere hvert komponent.

Hvad fik vi ud af at Unit teste

Vi fik følgende ud af at Unit teste.

- Finde dårlig kode / bugs.
- Sikre os at hvert komponent virkede efter hensigten.
- Collective ownership - Extreme programming princip.
- Det har hjulpet os når vi har haft ændringer i koden, for at se om alting stadig virkede efter vi refaktorerede noget f eks. da vi ændrede Interfacet.

Vi vil gerne have bedre indsigt i hvordan man kan gøre brug af Integration test til at sikre sig at de forskellige komponenter i applikationen fungerer korrekt i forhold til hinanden.

Hvad er Integrationstest

Integrationstest er stadiet efter Unit test og stadiet før System testing.
En integrationstest er en test hvor man tester komponenter som afhænger af andre komponenter for at teste at de integrerer korrekt sammen.

Hvad har vi gjort

Da vi lavede vores integrationstest, kaldte vi på en metode i vores facade, og tjekkede derefter om resultatet var som vi regnede med.
For at lave en integrationstest på de komponenter som er afhængig af en database opsatte vi en test database, som vi vil komme mere ind på i næste afsnit.

Hvad fik vi ud af Integrationstest

Ved at have lavet en Integrationstest, fik vi testet om vores komponenter arbejdede sammen på den måde som vi ønskede og returnerede det vi regnede med i forhold til hinanden.
Derudover kunne vi nemt se hvis en refactoring ødelagde noget et andet sted i programmet.

Vi vil gerne have bedre erfaring med konkrete problemstillinger når man tester en database(i forbindelse med integration test, unit test og mocking).

Hvad er en database test og hvad er problemstillingerne

En database test er en test hvor man tester sin database, hvor man tester følgende:

- Checking the data Mapping.
- ACID (Atomicity, Consistency, Isolation, Durability) properties validation.
- Data Integrity
- Business rule conformance

Vi fandt en problemstilling vedrørende test af vores database, hvis man tester sin produktions database så kender man ikke den data der bliver returneret, da den jævnligt kan blive ændret.

Hvad har vi gjort

I vores unit tests har vi mocket følgende classes DataSource, Connection, PreparedStatement, Statement og ResultSet for at kunne håndtere hvad vores queries skal returnere når vi har kaldt vores query.

Til vores integrations test har vi oprettet to test databaser en på vores server og en som der bliver oprettet ved brug af Travis (som vi nævnte tidligere).
Derudover har vi oprettet et script som sletter og opretter dataen i test databaserne, så vi altid kender den data der skal testes.

Hvad fik vi ud af at teste vores database

Ved at have testet vores database med en test database kan vi være sikre på at vores queries virker da vi kender dataen, det havde været usikkert at konkludere ud fra produktions databasen da den indeholder flere tusinde rækker.