Oliver Leo Lassen, Kasper Alexander Olesen, Kasper Hornum Pontoppidan, Christian Sindt

Introduktion

Vi har i forbindelse med projektet lavet en ny applikation (Database projektet), bestående af blandt andet REST web-service, konfigureret i JAX-RS, samt en MySQL og Neo4J database.

Formålet med projektet var at opnå erfaring indenfor automatisering af forskellige tests. Vi har brugt Maven til at bygge vores projekt og konfigureret til at eksekvere testene ved hver build. For at automatisere det hele vil vi opsætte en Continuous Integration (CI) server ved hjælp af Travis.

Som udviklingsmetode har vi arbejdet efter den agile metode, især med fokus på Test driven development (TDD), så vi på den måde tvinger os selv til at lave flere og bedre unit tests.

Læringsmål for test-projektet:

- Hvorfor og hvordan udføres Performance test
- Opnå praktisk erfaring med CI (Travis)
- Opnå praktisk erfaring med TDD / automatiserede tests
- Opnå praktisk erfaring med de forskellige typer af test indenfor Performance testing.
- Opnå praktisk erfaring med Selenium
- Tydeliggøre performance-forskelle mellem MySQL og Neo4J

Continuous integration (Travis)

Vores plan var at vi ville have Travis til at bygge og køre vores tests, men vi har haft lært at der er en del udfordringer ved at arbejde i skyen, frem for lokalt på vores maskiner. Vi prøvede at sætte Travis op til uploade til en gratis online host Heroku, efter den havde kørt vores tests.

Når vi pusher til git, så vil Travis automatisk gå i gang med at køre “mvn test”, og dermed køre vores tests. Vi har haft nogle udfordringer i forhold til Travis og vores databaser da Travis fungerer i “skyen” og dermed ikke har adgang til vores databaser som vi har liggende på lokalt.

På vores maskiner kan vi køre projektet lokalt med lokale databaser, men vi har ikke kunne finde en host, som har kunnet hoste Neo4J og MySQL, så derfor virker vores Travis og Heroku ikke, men ville i teorien fungerer det, hvis Travis havde kørt på vores lokale maskiner.

Performance testing

Vi har i forbindelse med projektet lavet non-funktionel testning, via Performance test. Vi har brugt værktøjet JMeter til at til at lave forskellige performance test, som kunne måle svar tider, fra henholdsvis MySQL og Neo4J databaserne.

Vi opsat vores JMeter test til at teste en form for integration test. Vi tester vores REST api, igennem http request's og videre til vores persistence lag, som laver en forespørgsel til den valgte database som sender et svar tilbage til REST api'et, og svaret fra http responset, logger JMeter efterfølgende i en .CSV fil.
Vi lavede også vores performance test's, med henblik på Test Driven Development, hvilket vil sige vi opsatte vores performance test, og http requests, inden vi havde lavet vores REST web-service. Dette resulterede i at det var meget nemmere og hurtigere at satte api'et op i Java, da vi på forhånd kendte alle URI'er.

Selve eksekveringen af performance testene blev fuldført gennem opsætning i Mavens pom fil i Java, hvor og kommandoen "Mvn Verify". Ved hjælp af Travis og continues integration, byggede Maven projektet hver gang der blev push'et til Git, efter projektet var blevet bygge og deployet, blev performance testene eksekveret og resultaterne blev skrevet til .CSV filen. På den måde blev vores performance tests automatiseret og det gjorde det muligt hele tiden at følge med i udviklingen af projektet, og sikre at de ændringer vi foretog ikke forringede projektets performance i en sådanne grad at vores program blev ubrugeligt.

Grundet udfordringer med at hoste Neo4J, blev vi til sidst i projektet nødt til at kører vores program lokalt, hvilket selvfølgelig resulterede i at vi mistede det automatiserede aspekt, i forhold til vore performance tests. Vi har dog stadig brugt JMeter som værktøj til performance sammenligning mellem de to databaser, vi har brugt.

Selenium

Ved hjælp af Selenium har vi foretaget tests af vores front-end WEB-UI. Vi har brugt Selenium til System Testing, ved at vi bl.a. tester at sitet bliver indlæst og at data bliver returneret til brugeren når der søges. Vi har skrevet en række tests som alle er med til at sikrer at funktionaliteten bag de forskellige elementer, er på plads. Vi har brugt Selenium's WebDriver til bl.a. at håndtere valg af browser og destinationen for test-sitet, og vi brugt WebElement funktionen til at håndtere de forskellige elementer vi gerne ville teste på vores WEB-UI.

Da vi gerne ville sikre at brugeren kun kunne indtaste og sende et søgekriterie pr. søgning, valgte vi at teste at input felter og checkbokse bliver nulstillet, hver gang brugeren vælger et nyt søgefelt. Vi tester også, at alle elementer på siden er blevet indlæst og at de dermed er tilgængelige for brugeren, samt at en liste med resultater bliver synlig, når brugeren har indtastet et korrekt søgekriterie.

Selenium er et udmærket værktøj til at teste front-end, men vi oplevede at det kunne være lidt svært, at skrive vores tests færdige før vi skrev koden, fordi det var udfordrende at få et overblik over, hvordan funktionaliteten og elementerne på sitet skulle bygges op.

TDD & Unit tests

Vi har forsøgt så vidt som muligt at følge tilgangen til component testing kendt fra Extreme Programming, ved at skrive vores units tests inden vi skrev selve koden, bedre kendt som Test-driven development. På den måde sikre vi at vores udviklingsmetode bliver iterativ og at vores komponenter ikke bliver for store og besværlig at teste. Vi har udvalgt de Java klasse som vi mente gav mest værdi at skrive tests til, så som Controllere og Persistence klasser, frem for f.eks. Entity klasserne.

Vi brugte jUnit framework og hamcrest matchers til at lave vores unit tests.
Vi har som tidliger nævnt stræbet efter at lave test først og det er lykkedes en del ad vejen. Vi synes vi har fået tilført projektet en stor del værdi ved at skrive vores test inden vi skrev selve koden, da vi helt sikkert har fået skrevet flere og bedre test end vi ellers ville. Det har hjulpet os med at undgå at commit kode der ikke fungerer efter hensigten og har sikret os en kendt kvalitet af koden. Vi har også forsøgt så vidt muligt at skrive kode der kan testes lettere, bl.a. ved at lave dependency injection i constructore.