CphBusiness

Gutenberg project

Github: https://github.com/Databasserne

Databasserne: Alexander Steen, Jonas Simonsen, Kasper Worm & Martin Karlsen.
26-05-2017

Gutenberg project

Test fag

Gruppe (Databasserne): Alexander Steen, Jonas Simonsen,
Kasper Worm & Martin Karlsen

Læringsmål

1. Bruge Test Driven Development (TDD) for at holde os på sporet af de mål vi har med i opgaven.
2. Stifte bekendtskab med testing af mobile applikationer (IOS & Android)
3. Teste vores REST api'er så vi er sikre på at vi får det rigtige data fra vores backend.
4. Mocke vores database i testene til API'et.
5. Performance teste vores backend for at sikre at vores kode er solidt opbygget.

Reflektioner

Vi har valgt at blande Database og Test fagene sammen og har lavet et Gutenberg projekt, med to databaser (MySQL og Neo4j). Vi har udviklet en backend samt to smartphone apps (Android og iOS). Til alle 3 dele, har vi brugt TDD og har dermed testet vores produkt.
Nedenfor vil vi reflektere over de forskellige læringsmål, og konkluderer på hvad der er gået godt og hvad der kunne have fungeret bedre samt, hvad der skulle til for at opnå et bedre resultat.

TDD Refleksion

I løbet af vores projekt periode har vi benyttet os af test driven development og skrevet tests før vi har begyndt at kode selve programmet.
I vores projekt har vi ikke fundet den store fordel ved at bruge test-first.
Vi har holdt os på sporet af det som vi skulle lave, med et klart mål i sigte, og der har test-first hjulpet os på vej, hvilket har givet os en fordel ved at teste først.
Men vi syntes ikke rigtigt det har gavnet os så meget, da vi har haft meget klare krav til hvad vi skulle lave i projektet og vi derfor ikke har været ude i den typiske faldgruppe hvor udviklerne laver "mere end de er blevet bedt om" og dermed spilder ressourcer på det.
Ud over det har vi haft lidt problemer når vi har skullet designe nogle af testene da det har været svært nogle gange at vide præcist hvad der skulle komme ud og hvordan man skal stille det hele op, dette skyldes nok vores uerfarenhed som testere og test-first princippet.
Ikke desto mindre har vi benyttet os af det, og da ingen af os rigtigt har brugt det i vores projekter før, har det været en god læringsprocess da vi har stiftet bekendtskab med det og prøvet at bruge det, og det har som sådan også lykkedes os da vi fik lavet tests først som fejlede og derefter fået skrevet koden til det med klare mål om hvad den skulle kunne.

Mobil app testing:

I løbet af vores projektet har vi udviklet to applikationer, både en iOS og Android app til at præsenterer vores data. Til at teste vores iOS applikation, har vi i iOS benyttet os af XCTest, ved at bruge dette framework, har vi hurtigt og enkelt haft mulighed for at lave Unit test, ved at teste vores applikation på denne måde sikrer vi at vores applikation, er robust, samtidig med at vi minimerer antallet af bugs. Vi har også benyttet XCTest til at teste vores UI, ved at teste vores UI sikrer vi at selve appens frontend virker efter hensigten. Den største udfordring i forhold til testing i iOS er TDD delen.
Til at teste vores Android applikation, har vi i Android benyttet os af Espresso,
Espresso er et UI testing framework som tester android appens UI. Espresso er meget enkelt og lige til at komme i gang med, og har ikke voldt de store problemer.
Selve testene er let læselige da de bruger en variant af hamcrest, som gør at det er mere "engelsk talende" og minder meget om "normalt sprog" i stedet for at være meget tekniske.
Den største udfordring ved Espresso er når man skal finde de enkelte elementer i appen, og gøre noget ved dem, men selv det er normalt enkelt nok, da de fleste visuelle elementer har et unikt id som man kan matche op imod.
TIl android har vi også brugt mocking, hvor vi har mocket server kaldene væk og indsat et bestemt response når der kommer et kald, som så returnere specifikt data tilbage, som vi tester op imod. Test dataen er taget ud fra et rigtigt http kald og ligner det.
Under testing af android har vi også brugt travis, og dette er ikke altid ligetil at få til at fungere. Dels er travis gerne 15min om at builde de nødvendige dependencies bare for at få android til at køre, derefter kommer selve buildet af projektet, starten af en emulator etc. Hvad værre er at det ikke er altid at emulatoren kan køre testene ordentligt, en test kan virke 100% på den lokale maskine, men

fejle på travis pga emulatoren måske er for langsom, eller ikke kan finde det enkelte element. Hvilket tit kan give problemer og stoppe processen unødigt.

Rest API test:

Vi overvejede hvorvidt vi skulle teste vores API kald, via Rest Assured eller jMeter. Fordelen ved at bruge jMeter er at, det er hurtigt at sætte op og da vi allerede skulle lave performance test, via jMeter, kunne det være oplagt også at bruge det til API test. Ulempen er dog at testene ikke bliver en integreret del af projektet, men et sideløbende projekt og vil kræve mere at integrere i et continuous delivery pipeline.
Fordelen ved Rest Assured er at vi kan integrere det sammen med resten af projektet og kører det sammen med de andre unit tests. Ulempen er at det er problematisk at få til at fungere på Travis, hvilket vi har brugt under udviklingen.
Vi har valgt at bruge Rest Assured til at teste vores API kald, for at integrere det med resten af systemet og dermed gør det lettere at køre.

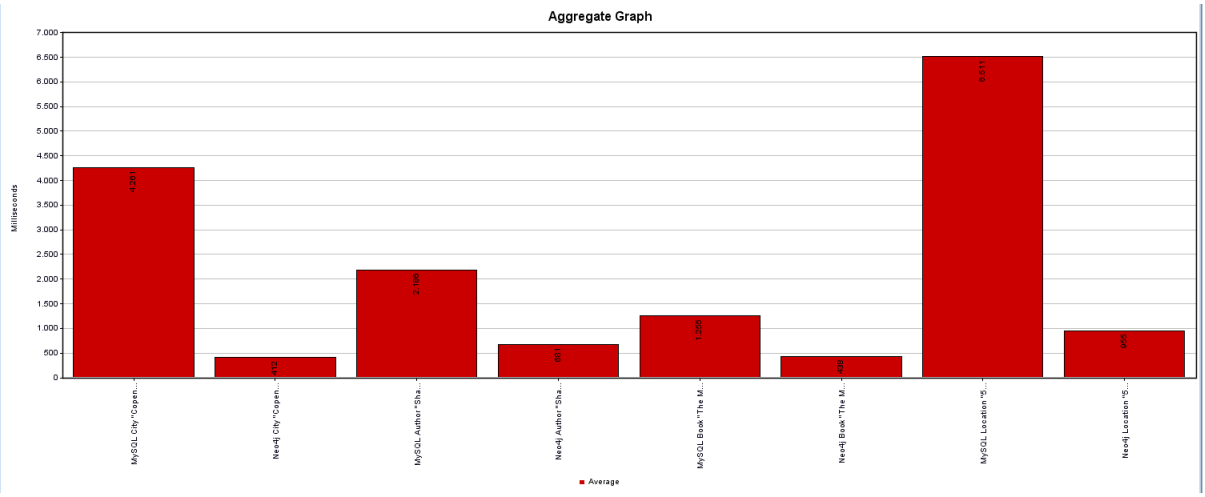
Database mocking:

I android appens test er der blevet mocket når der blev lavet et kald til serveren, dette er som sådan ikke et direkte mocking af databasen, men snarere serveren, idet vi går ind og tager det kald der bliver sendt, og stopper det, for så at sende vores eget response tilbage som indeholder det data vi ønsker. Vi bruger så den data i vores tests og lader som om det er det rette data vi arbejder med.

I backend har vi valgt kun at mocke (stubbe) Neo4j databasen, da denne ikke er en fungerende database i Travis' miljø. For at kunne kører testene, på Travis, uden at fejle har vi derfor valgt at mocke dem med Mockito.

Performance testing:

Vi har, via jMeter, kørt 1000 test for hvert af vores API'er, for at teste performance. Vi har ikke haft nogen krav til performance, andet end at kaldende ikke skulle resultere i fejl. Vores performance test tog 14 minutter at udføre i alt. Den er lavet på vores lokale maskine og vi forventer derfor at resultatet ville være langt bedre på en produktions server.
Man kan generelt se at Neo4j databasen er hurtigere end MySQL databasen, på alle kald.



Konklusion

Vi har igennem projektet arbejdet med at opnå ovenstående læringsmål. Ud fra det resultat, vi er nået frem til i projektet, kan vi konkluderer at vi har arbejdet med alle de opsatte læringsmål. Ved at vi har haft læringsmål til projektet, har det hjulpet os til at gennemarbejde projektet, på en mere effektiv, præcis og struktureret måde.