Individuell rapport

03 - Testing

Gruppe 56 - 525194

Totale ord: 1362

Martin Skatvedt
4. desember 2021

1.1 Innledning

Prosjektet vårt ble delt opp i to deler; en rest server som ble skrevet i Java ved hjelp av Spring-Boot, og klienten vår som vi skrev i Typescript med React. Vi brukte derfor ulike testrammeverk for henholdsvis klienten og rest serveren vår. For klienten vår bruke vi Jest og React-Testing-Library. Og for rest serveren vår brukte vi Junit5. Vi satte også opp ci/cd i gitlab for å kunne kjøre testene automatisk når man overførte til en grein i kodelageret vårt, det ble også rapport test-dekning som man kunne se i kodelageret med gitlab-skilt. For klienten vår bruke vi øyeblikk-testing. Det betyr at koden tar et «bilde» av nettsiden vår, og hver gang testene kjøres vil de sammenlignes. Hvis siden har endret på seg vil testen feile. For rest serveren vår utførte vi de ulike kallene til endepunktene våre. Hvis testen ikke fikk forventet resultat tilbake fra serveren ville testene feile. Under rest serveren hadde vi også noen enklere kjerneklasser som også ble testet med Junit5. Vi brukte testingen gjennom hele prosjektet som en måte å verifisere at koden vår oppførte seg slik vi ønsket, og at man ikke ubevisst endret på viktig funksjonalitet som kunne endret oppførselen til programmet vårt. Vi skrev hovedsakelig enhets-tester, men hadde også integrasjonstester for å teste rest serveren vår.

1.2 Faktorer som økte test-kvaliteten

For å øke kvaliteten på testingen vår arbeidet vi med kontinuerlig testing. Kontinuering testing handler å utføre testene gjennom hele utviklingen av prosjektet. Det vil redusere tiden fra en utvikler programmerer en feil, til feilen blir oppdaget (Continuous testing, 2021). Måten vi utførte kontinuerlig testing på var ved hjelp av gitlab sine innebygde DevOps funksjoner. Her kan man skrive kode som blir utførte ved spesielle handlinger. I vårt kodelager satte vi opp koden til å kjøre hver gang noe ble overført til en grein. Dette gjorde vi får å oppdage feil i koden tidlig, slik at hvis det tidlig oppstå en feil i grein kunne man oppdage det før man skulle innlemme det i hoved-greinen vår. Koden som kjørte rapporterte også testdekningen på prosjektet vårt og viste det frem som gitlab-skilt. Dette gjorde at vi hele tiden hadde overblikk på hvor stor testdekning vi hadde.

Testdekning er hvor mange prosent av koden som testes, dette blir gjerne regnet ut gjennom antall testene linjer i koden. Gjennom hele prosjektet hadde vi et stort søkelys på å ha en høy testdekning, noe vi greide å gjennomføre. Vi skilte mellom testdekning for klienten og rest serveren, hvor rest serveren hadde noe høyere testdekning enn klienten. Testdekningen ga oss

tillit til koden vi hadde skrevet og var viktig for oss gjennom hele prosjektet. Ved hjelp av den kontinuerlige testingen vi brukte, hadde vi også mulighet til å alltid se hvor høy testdekning vi hadde i hver grein i kodelageret.

Valget av verktøy vi brukte økte også kvaliteten på testene våre. Det finnes mange biblioteker man kan bruke for testing, men vi brukte tid på å undersøke hvilke test-biblioteker som passet best for oss. Vi la vekt på dokumentasjon, enkelthet og ressurser som fantes rundt biblioteket. Til slutt endte vi opp med Junit5 for rest serveren, og en blanding av Jest og React-Testing-Library for klienten. Valget av bibliotekene våre var vi fornøyde med, og var enige om at tiden brukt på å bestemme hvilke bibliotek vi skulle bruke var verdifull. Vi var også fornøyde med valget av å bruke gitlab sin ci/cd for å kunne hele tiden holdes oppdatert på teststatus og testdekning.

1.3 Faktorer som reduserte test-kvaliteten

Noe som reduserte kvaliteten på testingen vår var søkelyset på høyest mulig testdekning. Testdekningen sier ikke noe om kvaliteten på test-samlingen, men heller hvor mange linjer i programmet som er testet. Selv om testdekningen er 100% vil ikke det bety at alle mulige konsekvenser er testet. I ettertid av prosjektet har vi sett over testene og oppdaget at flere av testene våre ikke er tilstrekkelige. Hvis vi i tillegg til å ha et høyt fokus på testdekning, også hatt et større søkelys på *hva* som ble testet, kunne det økt kvaliteten på testene våre betraktelig.

En annen faktor som reduserte kvaliteten på testene våre var at vi ofte skrev tester til vår egen kode. Ved å teste egen kode oppstår et visst bias, som reduserer kvaliteten på testene. Hvis man utvikler en klasse selv, vil man kanskje være for selvsikker i sin egen evne og unngå å teste visse deler av koden. Derimot hvis en annen person på gruppen hadde skrevet tester til en annen person sin kode ville man hatt et mer objektivt syn på koden, som igjen kanskje ville økt test-kvaliteten vår. Da vi arbeidet med prosjektet ble det sett på som mest meningsfylt å skrive testene selv, men i ettertid har det vist seg å ikke være gunstig.

En tredje faktor som reduserte test-kvaliteten vår var når vi skrev testene. Testene ble ofte utviklet i ettertid av at koden ble skrevet, og dermed ble testene formet av hvordan klassen var skrevet. Hadde testene blitt skrevet på forhånd eller i umiddelbar nærhet hadde man skrevet bedre tester samt bedre kode. Testene er også en form for dokumentasjon for hvordan koden skal oppføre seg, og man hadde lagt en plan for hvordan klassen skulle se ut. En slik prosess hvor man utarbeider tester før koden, kalles testdrevet utvikling. (TestDrevet Utvikling (TDD), 2016)

1.4 Hvordan sikre god test-kvalitet

Erfaringen gitt av prosjektet har gitt oss flere perspektiver på hvordan vi kan sikre god test-kvalitet i fremtiden. Den første faktoren som kan sikre suksess i fremtiden er når man skal skrive testene. Ved å skrive testene enten på forhånd eller rett etter man har skrevet koden som skal testes kan man forme koden etter testene, og ikke testene etter koden. På denne måten kan man sikre at koden oppfører seg slik man har tenkt på forhånd, og dermed legge til de riktige valideringsmetodene i koden. Dette var noe vi ikke praktiserte, og endte med å måtte skrive om mye kode i ettertid.

En annen faktor vi har fått erfare er at man ikke burde skrive tester til sin egen kode. Noe vi også gjorde. Mange virksomheter har egne ansatte som arbeider med kvalitetssikring, det gjør at utvikleren selv ikke skriver sine egne tester. Ved å ha noen som er helt objektive til koden kan lettere se hva som må testes, og hvordan. Man er ofte for selvsikker i sin egen kode og kan ende opp med å ikke teste all funksjonaliteten.

Bruk av automatisk testing er også svært viktig. Ved å bruke verktøy som kjører regelmessig, kan man tidlig oppdage feil. I vårt eksempel bruke vi gitlab sine DevOps funksjoner, men det finnes utallige nettapplikasjoner som kan gjøre det samme. I vårt eksempel var det mest hensiktsfullt å bruke gitlab, siden vi allerede hadde kodelageret vårt der. Men man har også for eksempel Azure DevOps, hvor Microsoft har bygget en hel Platform kun for dette.

Den siste faktoren for å sikre god test-kvalitet er å ha en varierende test-samling. I vårt prosjekt la vi vekt på enhetstesting, hvor tester små biter av koden som for eksempel en klasse. (Hamilton, 2021). Enhetstester er viktige og tester grunnblokken i prosjektet, men man har også mange andre typer testing. Integrasjonstesting vil teste en gruppe med kode. Dette kan ta vært hvordan klienten og rest serveren samhandlet. Vi visste at hver av delene fungere selvstendig, men ikke hvordan de samhandlet. Ende-til-ende testing vil «simulere» en bruker å interagere med koden. Da vil man kunne teste hele kodebasen fra de øverste lagene til de laveste. Hvis vi hadde hatt en varierende test-samling kunne vi ha vært mer sikre på at hver gang vi overførte noe til hovedgreinen i kodelagret, så fungerte alle komponentene i samhandling. I stedet opplevde i flere tilfeller at klienten ikke fungerte som den skulle, hvis vi hadde endret på kode i rest serveren.

Referanser

Continuous testing. (2021, Mai 14). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Continuous_testing

Hamilton, T. (2021, Oktober 8). *What is Software Testing? Definition, Basics & Types in Software Engineering.* Hentet fra guru99.com: https://www.guru99.com/software-testing-introduction-importance.html

TestDrevet Utvikling (TDD). (2016, Mai 10). Hentet fra uio.no: https://www.uio.no/for-ansatte/enhetssider/los/usit/arrangementer/forum/2010/20101124.html