cphbusiness
COPENHAGEN BUSINESS ACADEMY

Whistleblower

Sider	43
Ord	12.092
Tegn (uden mellemrum)	70.913
Tegn (med mellemrum)	83.856
Afsnit	1.067
Linjer	1.753

BACHELOR PROJEKT

Software udvikler

Mahnaz Karimi

Indhold

Indledning

1. Formål

Formålet med denne rapport er at dokumentere og beskrive resultaterne fra arbejdet med mit bachelorprojekt.

I mit projekt vil jeg lave en CI/CD-chain som kan bruges til automatisk at teste integreret kode og deliver/deploye koden til et testmiljø og et produktionsmiljø.

Problemformulering

I softwareudvikling kan det være meget dyrt og farligt at release software med fejl i produktionen. At sende satellitter ud i rummet og lade dem brænde op på grund af softwarefejl eller have funktionsfejl på pacemakere på grund af softwarefejl kan være både dyrt og livstruende.

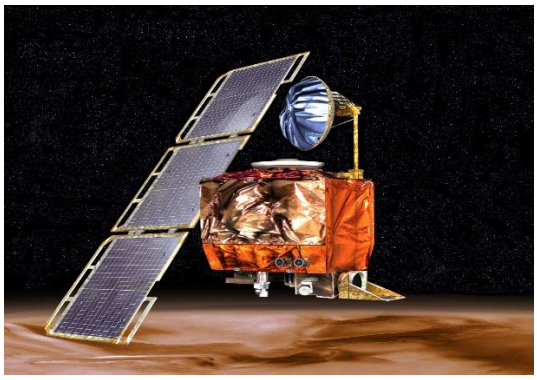


Figure 1.: Mars Climate Orbiter der brændte op i Mars-atmosfæren på grund af softwarefej l- en fejl på 328 millioner USD (ifølge Wikipedia)

Da flere og flere produkter er afhængige af nyudviklet software, vokser risikoen for at der dukker en fejl op og derfor også risikoen for alvorlige problemer på grund af disse softwarefejl. Risikoen kan reduceres på mange måder, og en måde kan være at vi sikrer os, at softwaren testes og sørge for, at softwaren leveres og deployes med de rette procedurer. At lave med en CI/CD-chain som automatiserer test, levering og deploying, er muligvis en god løsning.

Med denne bachelorrapport vil jeg forsøge at undersøge dette mere detaljeret ved at oprette et udviklingsworkflow (CI/CD-chain) understøttet af automatiseret test og automatiseret levering for at se, om en sådan et workflow er værd at udføre og faktisk understøtter udviklingen af en applikation på en god måde.

Opsætningen

Jeg arbejder alene, så mange af de problemer, der opstår, når man arbejder i teams, vil ikke være nøjagtigt de samme for mig. Dette gør det vanskeligt at argumentere for den virkelige værdi af denne måde at arbejde med softwareudvikling på i et rigtigt udviklingsmiljø i et firma. Selvfølgelig er meget software udviklet af enkelte programmører, der arbejder alene, og i det mindste vil det blive dækket af mit arbejde. Jeg kan også bruge mine egne tidligere erfaringer med at udvikle software med andre, når jeg analyserer udbyttet af workflowet.

Løsningen

Hyppig integration af software hjælper med at reducere omkostningerne ved misforståelser ved at reducere det tabte arbejde, når misforståelser opdages. Det er meget billigere at finde og rette en fejl, hvis

den udviklede software kun er en dag gammel end at opdage fejlen i software, der er to måneder (eller mere) gammel, da kompleksiteten af et dagsarbejde sandsynligvis vil være meget mindre end kompleksiteten af to måneders arbejde og da udviklerne bedre kan huske det de lige har lavet end det de lavede for lang tid siden. Dette gælder, uanset om man arbejder alene eller arbejder i teams, så hyppig integration mellem den nye kode og kodebasen er vigtig i begge tilfælde.

For at kunne integrere ny software i en kodebase skal man vide, at den nye software fungerer i sig selv, at den fungerer med kodebasen, og at den ikke ødelægger noget i kodebasen, når der laves integration.

Automatisk test

En måde at vide, om noget er ødelagt, er at teste det, men det er et "etableret faktum", at test er kedeligt sammenlignet med udvikling, og gentagelse af test for at finde regressioner er endnu mere kedeligt. Faktisk er det så kedeligt, at det i mange tilfælde slet ikke gøres, og derfor kommer fejl i produktion.

Løsningen er: automatisk test. Automatisk test har fordelene ved at være sjovt at se kørende, sjovt at udvikle (det er en udviklingsopgave mere end en test-opgave), og det kan gentages uendeligt uden indsats fra udvikleren.

Automatisk test sikrer, at omkostningerne ved at køre testcases igen (regressionstest) er lav sammenlignet med at køre sådanne tests manuelt, og at have evnen til at køre automatiske tests, når man udvikler eller refaktorere software, gør det meget lettere at afgøre om den nye software fungerer som forventet uden at ødelægge noget, der allerede har fungeret.

Fejl ved levering og deploy

Softwarefejl kan komme fra dårlig kodning, og det kan blive opdaget ved test. Softwarefejl kan også komme fra dårlig deploying, for eksempel at glemme at opdatere databasen, når en ny version releases, eller at glemme at køre tests, før softwaren releases. Fejl kan komme fra trætte mennesker under stress, der ikke kan huske alle de vigtige steps, der skal tages. Derfor kan automatisk leveringer og deploying af software hjælpe med at reducere risikoen af disse fejl, og det kan hjælpe at inkludere automatisk test i et leverings- eller deployingsworkflow.

Opsummerer fordelene

I denne rapport forsøger jeg at bruge definitionerne af kontinuerlig integration, kontinuerlig levering og kontinuerlig deploy som defineret af Atlassian (https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment).

At være i stand til at integrere kontinuerligt kan få fordelene (ifølge Atlassian):

- Færre bugs overføres til produktion, fordi regressioner opdages tidligt i automatiserede test.
- Opbygning af release er lettere, da alle integrationsproblemer er blevet løst tidligt.
- Det er mindre kontekstskift, så snart de ødelægger build og kan arbejde på at rette det, før de går til en anden opgave.
- Testomkostningerne reduceres meget. CI-serveren kan køre hundreder af tests i løbet af kort tid.
- QA-team bruger mindre tid på at teste og kan fokusere på accept-test og eksplorative test.

At være i stand til at levere kontinuerligt kan få fordelene (også ifølge Atlassian):

- Kompleksiteten ved deploying af software er mindre. Teamet behøver ikke længere bruge dage på at forberede sig på en release.
- Man kan release oftere og dermed fremskynde feedback-loop med kunderne.

- Der er meget mindre pres på beslutninger for små ændringer, hvilket opmuntrer udviklerne til en hurtigere iterering.

At være i stand til at deploy kontinuerligt vil få fordelene (endnu en gang ifølge Atlassian):

- Man kan udvikle hurtigere, da der ikke er behov for at sætte udviklingen på pause for releases. Deployments-pipelines kan udløses automatisk for hver ændring, når kode integreres og leveres.
- Releases er mindre risikable og lettere at rette i tilfælde af et problem, da man deploy små batcher af ændringer.
- Kunder ser en kontinuerlig strøm af forbedringer hver dag i stedet for hver måned, kvartal eller år, og kvaliteten stiger hver dag.

Forventet resultat

Jeg forventer at skabe en udviklingsopsætning, der giver mulighed for let kontinuerlig integration, let kontinuerlig levering og let kontinuerlig deployment til produktion. I alle tre faser skal der udføres automatisk test for at sikre, at kvaliteten af softwaren er kendt.

Jeg forventer også at få noget praktisk erfaring med denne måde at arbejde på og med at oprette sådanne workflows for at få specifik indsigt i fordelene ved denne måde at arbejde på plus at få erfaring med automatisk test.

Jeg håber til sidst at kunne sige noget om, hvordan de automatiske pipelines hjælper mig med at udvikle software.

Projektopgaver

Før jeg startede projektet, havde jeg ikke meget tidligere erfaring med agil udvikling og testning i en CI/CD-chain, og jeg kendte ikke så mange værktøjer, der blev brugt til dette, så en af projektopgaverne var at undersøge, hvilke værktøjer der kunne findes og at beslutte, hvilken platform der skal bruges og implementeres.

Efter at havde besluttet et værktøj til CI/CD-chain havde jeg også en opgave med at finde ud af, hvilke platforme jeg ville bruge til min integration, min levering og min deployment til produktion.

En anden opgave var at forsøge at bruge min viden om testning, i en automatisk opsætning, som jeg ikke tidligere havde arbejdet med. Her skulle de praktiske implementeringer afklares.

For at opsummere havde jeg brug for:

1. Find ud af, hvilket værktøj jeg skal bruge til mine CI/CD-chains
2. Find ud af, hvilke platforme jeg vil bruge til integration, levering og deployment
3. Find ud af, hvordan man implementerer automatisk test i de forskellige workflows
4. Implementere den kontinuerlige integration
5. Gennemfør den kontinuerlige levering
6. Implementere den kontinuerlige deployment
7. Oplev hele workflow på en praktisk applikation

For at løse 1. og 2. forsøgte jeg at lytte til eksperter (f.eks. min vejleder) og undersøgte sider på Internettet med en liste over forskellige muligheder for værktøjer til brug og samt kommentarerne på deres egnethed.

For at løse 3. læste jeg forskellige tutorials og så videoer, der beskriver værktøjer og testimplementeringer.

For at løse 4., 5. Og 6. læste jeg en del dokumentation og forsøgte at få det til at virke, mens jeg gik fremad.

For at løse 7. udviklede jeg en applikation til registrering af whistleblower-anmeldelser og brugte dette som driveren til løbende at forbedre mine workflows.

2. Læsevejledning

Første del af rapporten handler om at vælge en platform til CI/CD. Næste del af rapporten handler om den applikation jeg vil udvikle for at bruge min CI/CD-løsning. Tredje del handler om Implementering CI/CD og konfigurering af servere jeg vil bruge. Fjerde del handler om automatisk test, og til slut vil jeg sammenfatte, diskutere og konkludere.

3. Mulige CI/CD-systemer

Ifølge LAMBDATEST (https://www.lambdatest.com/blog/27-best-ci-cd-tools/) og Katalon (https://www.katalon.com/resources-center/blog/ci-cd-tools/), er der følgende CI/CD-systemer man skal kende:

1. Jenkins	8. GitLab CI	15. Buildbot	22. CruiseControl
2. TeamCity	9. Jenkins X	16. Semaphore	23. Bitrise
3. CircleCI	10. Shippable	17. Wercker	24. Drone CI
4. Travis CI	11. Buildkite	18. Integrity	25. UrbanCode
5. Bamboo	12. Concourse CI	19. Weave Flux	26. StridergtiFinalBuilder
6. GoCD	13. Codefresh	20. NeverCode	27. Spinnaker
7. CodeShip	14. Buddy	21. AutoRABIT	28. GitHub actions

For at finde ud af hvilke værktøjer der er gode at vælge undersøgte jeg på de to https://stackshare.io/ og https://www.capterra.com

Hjemmesiden Stackshare.io er god fordi i den har samlet mange it-værktøjer på sig som folk kan skrive kommenter til hvert værktøj og stemme på værktøjets muligheder. For at man kan stemme på værktøjer i https://stackshare.io/ skal man tilknytte til en GitHubside eller google konto. Dette betyder at folk er certificeret, men den er informationer som er på hjemmesiden, er ikke noget man helt kan stole på at de er korrekte men der antal af personer der har stemt på et værktøj, eller stemte på en kommenter vil det betyde at det er noget man kan betragte når man vil vælge værktøjet.

Jeg har valgt at læse mere om nogle CI/CD-værktøjer for at prøve at finde ud af hvordan de fungerer som er god for hjemmesider: Jenkins, CircleCI, GitLab, GitHub-Actions, Travic CI. De sammenligninger gør det nemmere for at vælge et CI/CD-værktøj, der passer til mit projektkrav. Jeg har kigget efter antal af udviklere der stemte og hvilken rating/stjerne den har fået. Jeg har kigget efter hvilke store firmaer og antal firmaer arbejde med hvilket CI/CD-værktøjer og hvor mange udvikler arbejder med den.

Denne liste indeholder de fælles CI/CD-værktøjer fra de to hjemmesider sammen med deres funktioner Dette tabellen gør udvælgelsesprocessen lettere:

	De Stor firma arbejder med	Hosting	Til Private projekt	prise	Antal/ anmeldelse	Antal af firma	Antal af Tools integreret	Antal af developers
Jenkins	Facebook, Netflix, Linkedin, ebay,	Linode, AWS, Google Cloud, Azure,	ja	Free version	4.5/5 (339)	2805	142	32001

	Zalando. Robinhood osv	DigitalOcean, and more						
Circleci	Spotify, Coinbase, Stitch Fix, and BuzzFeed, StackShare, Delivery Hero, Tokopedia, Frontend, Scale, MAKE IT	Linode, DigitalOcean AWS, Google Cloud, Azure, and more.	ja	$30.00/ måned	4.6/5 (64)	1461	96	4924
Travis CI	IBM, Zendesk, BitTorrent, Heroku, MOZ, Lyft, Delivery Hero, Graphy, Heroku	Linode, DigitalOcean AWS, Google Cloud, Kubernetes, Azure, and more.	ja	free	4.4/5 (44)	969	81	5449
GitLab CI	Alibaba Travels, fødevarestyrelsen, Remote Team osv	Linode, multi-cloud YAML file named .gitlab-ci.yml self-hosted	ja	$ 4,00 / måned	4.6/5 (536)	499	13	1050
GitHub-Actions	Bepro Company, GitBook, Frontend, Kong, JS osv	our self-hosted runners, multi-cloud f.eks. Linode, DigitalOcean	ja	$ 4,00 / måned		127	11	196

Jeg forsøgte også at undersøge dokumentationen til nogle af systemerne og forsøgte at finde materiale som artikler og vejledninger til opsætning og brug af dem, og nogle af dem var virkelig komplicerede at forstå (f.eks. prøvede jeg Jenkins, men kunne ikke få det til at fungere), og nogle af dem var lettere.

Da jeg allerede vidste lidt om at arbejde med Git og branching, og da dokumentationen til GitHub-actions (https://docs.github.com/en/actions) syntes rimelig tilgængelig, besluttede jeg at arbejde med det. Der var ikke så mange kendte firmaer der bruger det og ikke mange der havde stemt på det, men det var nyt og det kunne forklare det. Af en eller anden grund blev det det jeg foretrak.

Noget at arbejde på - softwareprojektet

For at have noget at arbejde med, så jeg kunne få erfaring med softwareudviklings flow, havde jeg brug for et projekt. En applikation jeg kunne forsøge at udvikle ved hjælp af disse workflows.

Jeg kom i kontakt med et firma, som sendte mig nogle krav til en prototype-whistleblower-applikation. Dette virkede som en vigtig applikation og et interessant projekt at arbejde på, så jeg besluttede, at det skulle være mit udviklingsprojekt til dette bachelorprojekt.

Baseret på dette, jeg valgte at bruge:

- Git som et versionskontrolsystem.
- GitHub som kode-repository.
- Python for det sprog, der blev brugt til at programmere.

- Django som web-framework til udvikling af applikationen.
- Den indbyggede testrunner i Django eller Pytest som test-runner til de automatiske test.
- GitHub-runner og GitHub til automatisk test (kontinuerlig integration).
- Heroku og Gunicorn til testmiljøet (kontinuerlig levering).
- Linode Linux-baserede servere til produktionsmiljøet (kontinuerlig deployment).
- PostgreSql som applikationsdatabasesystem.
- Apache og mod_wsgi som webserver i produktion.

Whistleblower-applikation

Da en stor del af mit arbejde er at udvikle den applikation, synes jeg, det er vigtigt, at denne applikation er dokumenteret. Ellers er deploy-beslutningerne og tankerne over test og workflow måske ikke så nemme at forstå. Det viser også, hvilken slags applikationsudvikling jeg baserer mine konklusioner på.

1. Formålet med applikationen

Det ser ud til, at der foregår mange svindel ting i virksomheder og regeringer, og at mange af disse ting kun kommer til offentlighedens kendskab, hvis en whistleblower, som for eksempel Chelsey Manning eller Edward Snowden, blæser i fløjten og afslører dem.

For en whistleblower kan afsløring af forkerte handlinger komme til at have en meget stor pris (fx Chelsey Manning, der gik i fængsel), men det er dog vigtigt at have disse sager afsløret alligevel. For at hjælpe med dette opretter jeg et program, der muliggør at sladre anonymt, så det får ikke mange konsekvenser for whistlebloweren.

Applikationen er beregnet til at være en prototype, og jeg tror ikke, jeg dækker enhver risiko for at afsløre whistleblowers-identiteten, men jeg gør det muligt at oprette sager uden at afsløre identiteten direkte og også, ved at bruge et specielt login, hvilket gør det vanskeligt at lave falske sager.

Kravene til applikation er specificeret bilag til denne rapport. Målet er at forberede en prototype for at afgøre, om en whistleblower-applikation er nyttig. Kravene fremsætter følgende argument:

> *Mange virksomheder er bekymrede for at få ødelagt deres renommé eller deres økonomi på grund af fusk eller svindel, og vil af den grund gerne gøre det lettere at få ansatte til at kunne anmelde, hvis de har kendskab til sådanne ting. Finansielle virksomheder skal efter lovgivningen have etableret en ordning så ansatte kan anmelde sådanne sager. Ansatte der anmelder sådanne ting, kalder man for whistleblowere*.

Dette menes som en prototype, der viser en mulig applikation, så der er mange ting, der ikke adresseres i denne applikation Der er for eksempel ikke angivet et admin-interface, og der er ikke gjort specielle tanker om at øge sikkerhed for at gøre applikationen svært imod stærke modstandere (for eksempel regeringer eller store virksomheder).

2. Use Cases

Fra kravdokumentet har jeg konkluderet, at der er følgende relevante use-cases og aktører i whistleblower-systemet.

Aktører

- Anonym medarbejder i firma som er whistleblower
- Sagsbehandler for firma som skal sagbsbehandle anmeldte sager

- Administrator af systemet der også er sagsbehandler men med særlige opgaver

Use-cases

Mulighederne for en medarbejder

- Sende en anmeldelse
- Upload filer
- Slette den hvis status er 'Ny oprettet'
- Gense sin sag

Mulighederne for en sagsbehandler

- Oversigt over alle de relevante sager
- Redigere sagerne
- Registrer sagsbehandler
- Release sagen til andre sagsbehandlere
- Læs tekst i tilfælde af
- Download vedhæftet fil til sag
- Upload vedhæftet fil til case
- Skriv ny tekst i sag
- Indstil sagsstatus
- Slet vedhæftede filer fra sag
- Slet sag

Administrator kan (administratorrettigheder)

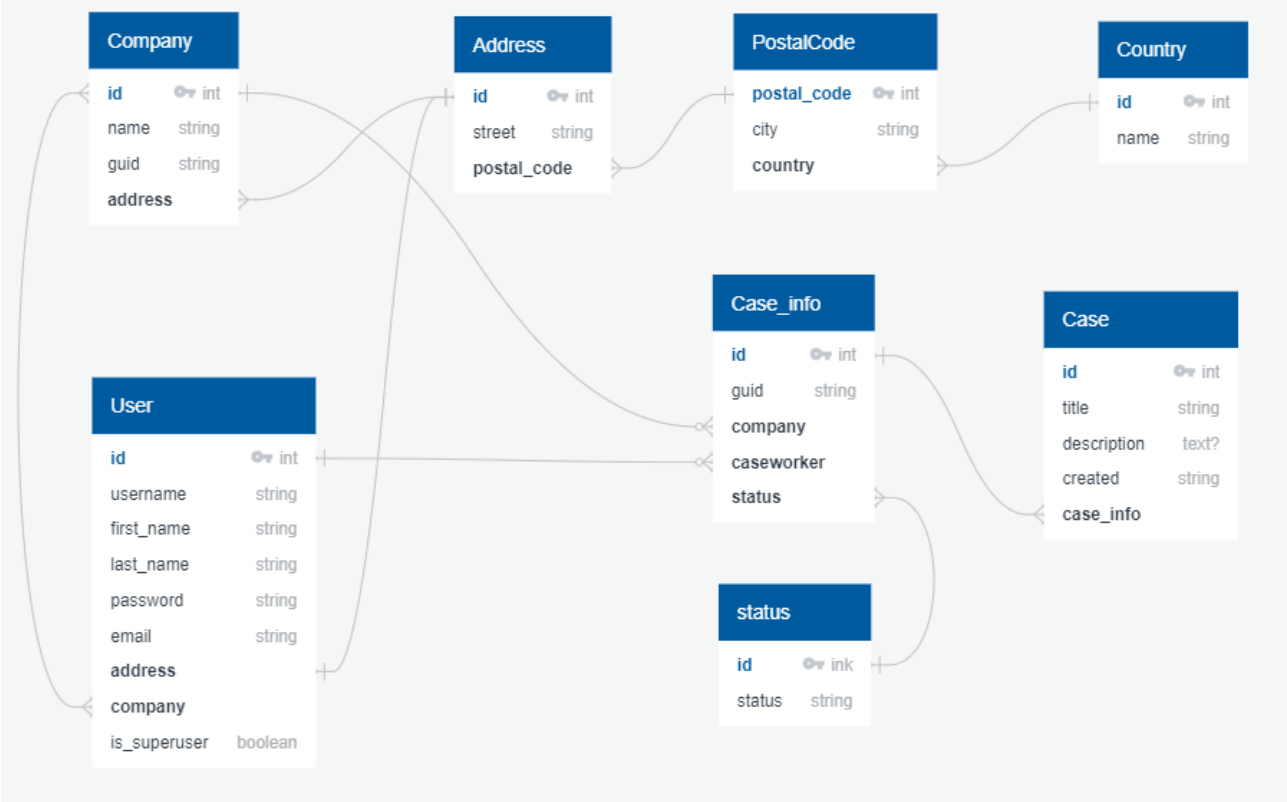
- Opret firma
- Opret sagsbehandler

Jeg har fundet ud af at der er tre måder med at login på. Man kan login med firma-guid, sags-guid eller med personlig login, som man normalt skal, når man arbejder som sagsbehandler.

Use-cases, og hvordan de er forbundet, vises i UML use-case diagram. Alle Use-cases til sagsbehandleren og administratoren kræver, at de logger ind med deres personlige brugernavn og adgangskode. Dette login er ikke en del af use case-diagrammet, men login med en firma-guid og en case-guid er i diagrammet, da det er specielle login (anonym).

Figur 1 Use-case diagram

Database model diagram



Figur 2 Database tabeller

Applikationen har brug for en database til at gemme forskellige typer data. Baseret på use case-diagrammet og kravdokumentet besluttede jeg at lave en databasemodel som vist i databasemodeldiagrammet. Dette diagram viser de tabeller, der er involveret i applikationen, og relationen mellem dem.

Datamodellen til whistleblower-applikationen består af 8 tabeller i en relations-database til model for sagsbehandlere, der arbejder med whistleblower-sagerne, de virksomheder, der vedrører sagerne og selve sagerne. User-tabellen er indbygget i Django.

Database tabeller

For at lave database struktur har jeg valgt at normalisere data i tredje normale form. I tredje form fjernes felter, der ikke afhænger af nøglen. Generelt, når som helst, indholdet af en gruppe af felter kan anvendes for mere end en enkelt række i tabellen, skal overvejes at placere disse felter i en separat tabel. Derfor har jeg separate tabeller for status, postnummer, land, adresse, postnumre, case og case-info. Selvom mange små tabeller kan dog reducere performance eller overstige open-file og ledig hukommelseskapacitet.

Det kan være en god ide kun at anvende tredje normale form på data, der ændres ofte.

Der er tre aktører involveret i applikationen: whistleblower, sagsbehandler og administrator. Whistleblower vil være anonym, og der vil derfor ikke være noget tabel for whistleblower. Administratoren er en speciel sagsbehandler, så administratoren og sagsbehandleren deler en tabel, nemlig User-tabellen.

En sag er altid relateret til et firma, så der er en firmatabel, og firmaets adresse er gemt i tre tabeller: adresse, postnummer og land. Ud over at have et navn og en adresse har et firma også en GUID (Global Unique Identifier), der også kaldes en UUID (Universal Unique Identifier). Denne GUID er identifikatoren, der også bruges af whistleblower til at logge ind og rapportere snyd.

Oplysninger om en sag opdeles mellem forskellige tabeller, en tabel med oplysninger om en sag, en tabel med selve sagens indhold. Oplysninger for en sag er det firma, sagen handler om, sagsbehandleren (der kan kun være én ad gangen), en status for sagen og en GUID, der kan bruges til at identificere sagen, så whistleblower kan se det igen.

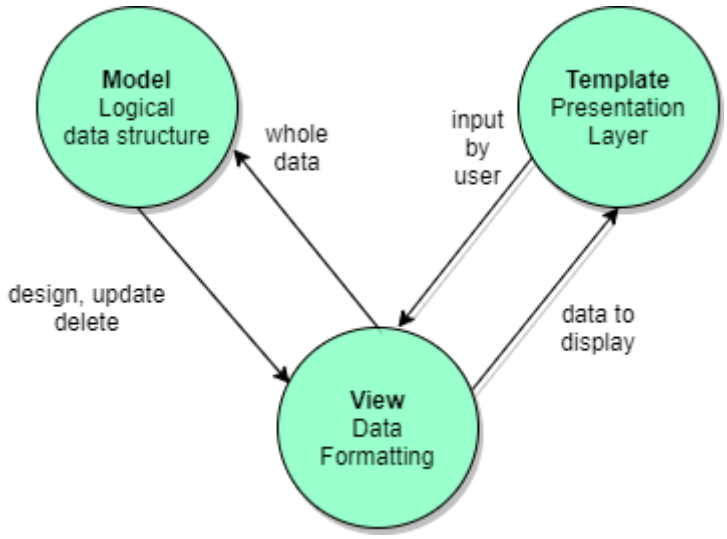
Der behøver ikke være en sagsbehandler i sagen, så sagsbehandlerfeltet kan være null.

3. Applikationens struktur/arkitektur

Whistleblower-applikationen er udviklet i Python ved hjælp af web-framework Django.

Django følger et MVT (Model View Template) design- pattern til webløsninger. Dette pattern modellerer dataene i applikationen i Model-komponenten. View-komponenten er ansvarlig for business-logikken, og Template-komponenten er ansvarlig for at præsentere dataene for verden. I modsætning til MVC (Model View Controller) pattern, der har en controller-del, der manipulerer modellen og renderer view. Dette gøres i Django af selve framework eller af view-component. View-component er ansvarlig for at acceptere http-requests og returnere http-responses.

Der er ingen separat controller, og komplet applikation er baseret på Model, View og Template. Derfor kaldes det MVT-applikation.



Figur 3 Inspireret af https://data-flair.training/blogs/django-mtv-architecture/

Whistleblower-applikationsarkitekturen består af selve applikationen, der findes på en webserver og en PostgreSql-database, enten som en service eller som et enkeltstående databasesystem på en server.

I GitHub kører PostgreSql-databasen som en service på GitHub-runner. PostgreSql som en service bruges på Heroku, og databaseserverløsningen bruges til produktion på Linode.

4. Deployment

Applikationen deployes i to environment, hvis man ikke betragter GitHub som et environment. Det mest vigtige environment er produktions-environment, hvor applikationen bruges af slutbrugerne. I dette miljø er der to servere, en server til applikationen og en server til databasesystemet.

Webapplikationen i produktion serves af Apache2 via et WSGI-interface, mod_wsgi.

Det andet environment er testmiljøet, hvor applikationen kan demonstreres og testes af brugerne. Dette environment har en server til applikationen, men databasen leveres som en service fra cloud-leverandøren.

I testmiljøet serves webapplikationen af en webserver kaldet Gunicorn, også via en WSGI- interface.

Deployment diagrams

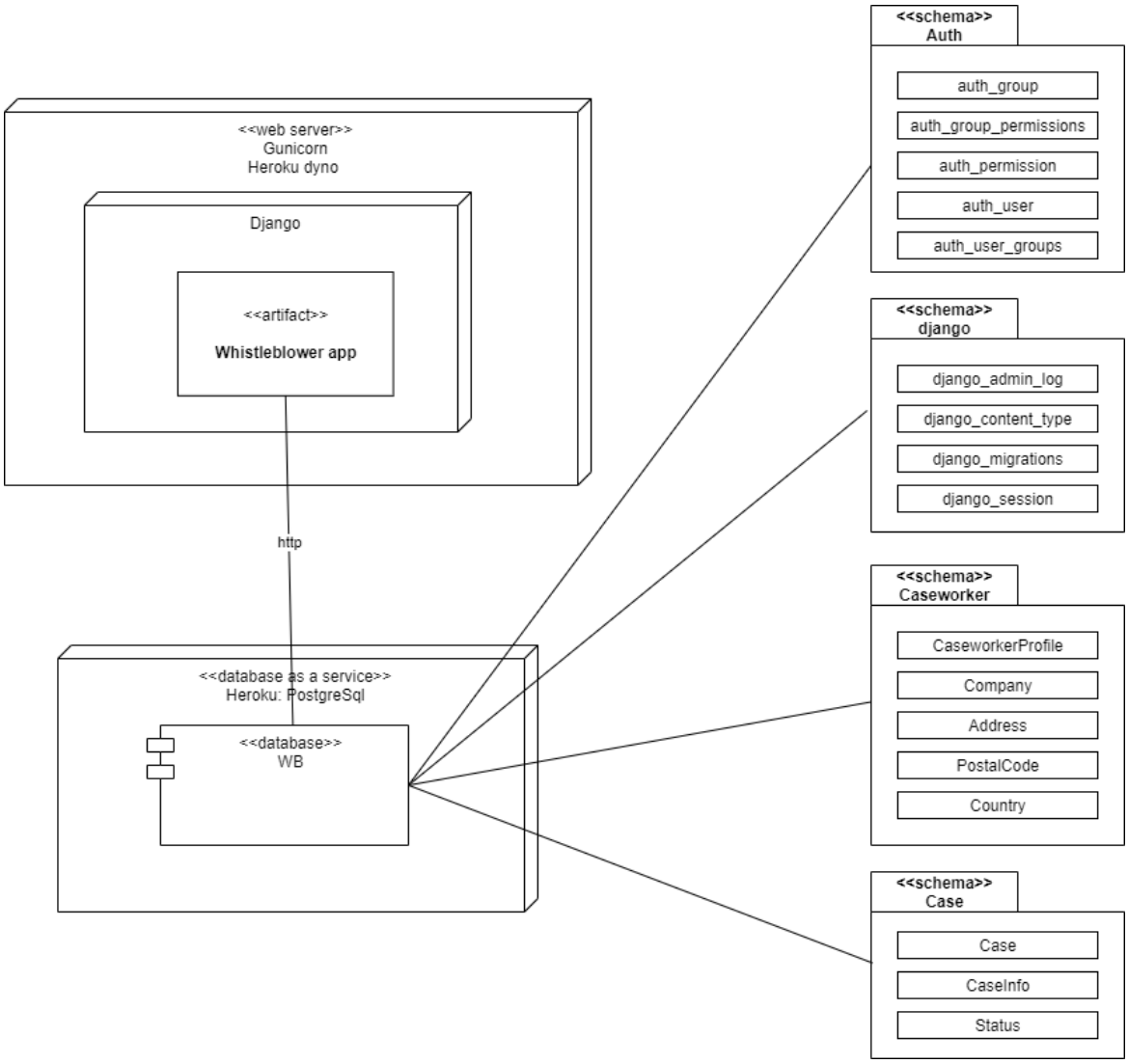
Jeg har tegnet to diagrammer, der viser den nødvendige deployment til de to miljø test og produktion. Jeg har tegnet applikationen, der ligger i Django og selve Django-frameworket, der bliver distribueret til en webserver, som kører på en virtuel maskine. På Heruko kaldes en virtuel maskine en “Dyno” og på Linode kaldes en virtuel maskine en “Linode”.

Jeg bruger PostgreSQL til både testmiljø og til produktionsmiljø. I testmiljøet er dette en service, der leveres af Heroku, og i Linode distribueres på sin egen virtuelle maskine.

Jeg har også tegnet de tabeller, der indgår i databaserne. De er de samme for begge environments. Når jeg søger på Internettet, ser det ud til, at der er forskellige måder at tegne deployment-diagrammer på, og den version, jeg endte med, er baseret på en måde, der gav mening for mig. Det viser de forskellige “hardware” -komponenter (de er virtuelle maskiner) og hjælper med at få en forståelse af, hvad der skal deployes og hvorhen det skal deployes og hvilken struktur der skal være i databasen.

Test-environment deployment diagram

Dette diagram viser implementeringen i testmiljøet på Heroku.

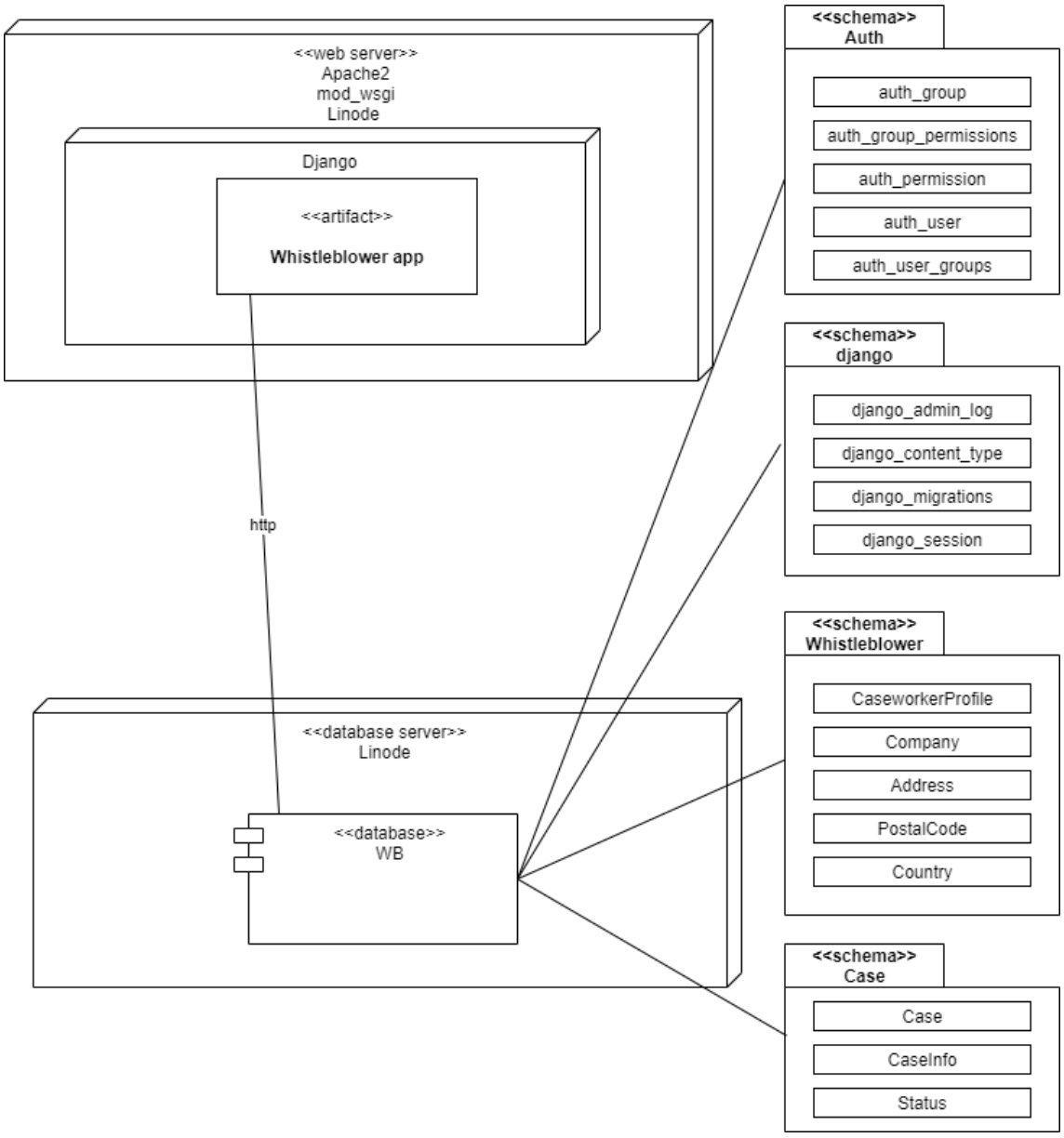


Figur 4 Test deployment diagram

Production-environment deployment-diagram

Dette diagram viser deploy af applikationen ind i produktionen. En webserver er installeret på en Linode (en virtuel maskine i Linode-cloud). Denne webserver server Django-applikationen installeret på den.

En anden Linode server PostgreSql-databasen med navnet WB vist som en komponent. Denne database indeholder skemaerne vist i højre side af diagrammet.



Figur 5 Produktion deployment diagram

5. Brug af applikationen

Der er tre typer brugere af applikationen og jeg vil beskrive disse tre mulige brugernes aktiviteter.

Der er en whistleblower, en sagsbehandler og en administrator.

En Whistleblower går til applikations url https://www.reporteasily.com/case/login/ og logger ind anonymt ved at indtaste et firma GUID. Efter login kan whistleblower skrive om de ting, der er forkerte.

Efter at have gjort det, når whistleblower submitter sagen, får whistleblower en response som en GUID, der fungerer som ID til sagen. Hvis whistleblower på et tidspunkt ønsker at se sagen igen med kommentarer fra en sagsbehandler, kan whistleblower gå til applikations url https://www.reporteasily.com/case/retrieve, indtaste GUID-nummer og se sagen inklusive alle kommentarer fra sagsbehandleren.

Sagsbehandleren/administratoren logger ind på applikationen, kan se på en liste over sager for de firmaer som sagsbehandleren er ansat for og har valgt at arbejde på. I listen kan hun også se de sager der ikke har en sagsbehandler endnu. Hun kan vælge de sager hun vil arbejde på. Der er kun en sagsbehander der kan arbejde på en sag ad gangen.

Sagsbehandleren kan arbejde ved at indstille status og tilføje kommentarer, hvis det er nødvendigt, og gøre hvad en Sagsbehandler gør for at håndtere whistleblowersager.

Administratoren kan gøre, hvad en sagsbehandler gør, men kan også være ansvarlig for registrering af nye sagsbehandlere og nye virksomheder. Administratoren bør have tillid fra alle virksomheder (for eksempel kan det være en advokat) til at udføre disse opgaver.

Anonymlogin, hvordan det fungerer:

For at sikre, at en medarbejder kan forblive anonym, men alligevel logge ind for at rapportere, skal de være logget ind med en nøgle, der kun er kendt af ansatte i den pågældende virksomhed, men som er delt for alle medarbejdere i virksomheden. På den måde kan det kun bestemmes, at whistleblower kommer fra en bestemt virksomhed, men ikke hvem whistleblower er.

Alle virksomheder tildeles en nøgle hver (en firma-guid), som de kan dele mellem alle medarbejdere. Folk uden for virksomheden må ikke kende nøglen, for det vil gøre det muligt for dem at oprette rapporter, som om de var ansat i virksomheden. Hvis virksomheds-guid bliver kompromitteret, kan virksomheds-guid ændres. Det kan også ske automatisk f.eks. hver måned, og når den nye Guid er blevet distribueret blandt den nuværende firmas medarbejder, kan de fortsætte med at rapportere.

6. Udvikling af applikationen

For at udvikle programmet har jeg valgt at arbejde i Ubuntu med Pycharm IDE og at bruge Django framework.

Det ville have været rart at erklære, at jeg analyserede kravene og baseret på det, skabte en masse features og stories, som jeg satte i udviklings-branch til udvikling med de automatiserede workflow i GitHub-actions.

Desværre skete det ikke. Jeg blev lidt overvældet af de forskellige opgaver, jeg havde brug for at udføre, så min fremgangsmåde endte med at blive mere ustruktureret, hvor jeg besluttede at lave en grundlæggende analyse med data-modellen og en use-caseanalyse af applikationen.

Så fortsatte jeg med at implementere meget af sagsbehandler-funktionaliteten og de medfølgende tests for at komme mere ind i Django-framework og for at kunne lave nogle automatiske test. Jeg kiggede også på, hvordan man fik GitHub-actions til at fungere og brugte en hel del tid på at finde ud af, hvordan jeg får

deploying til at fungere i mit test- og produktionsmiljø. Til sidst arbejdede jeg med det anonyme login og oprettelse af sager til whistleblower og endelig automatisk test.

Jeg startede med den indbyggede test-runner (kommandolinjen: "`python manage.py test`") og værktøjet Coverage, men indså, at Pytest var et bedre valg, og derfor arbejdede jeg med det i stedet. Pytest leverer også dækningsanalyse.

Særlige kodningsovervejelser og løsninger

Det normale login for sagsbehandlere og administratorer håndteres af det indbyggede Django brugervedligeholdelsessystem, der tager sig af login og logout og alle involverede sikkerhedsforanstaltninger. Dette kan ikke bruges til anonyme brugere, da det indbyggede system kræver et brugernavn.

I stedet skal de anonyme brugere logge ind med en nøgle, en GUID, der er specifik for deres virksomhed, og for at implementere det har jeg brugt følgende flow:

1. Whistleblower går til url "`https://www.reporteasily.com/case/login`".
2. Whistleblower indtaster en guid der hører til et firma.
3. Whistleblower ledes til en side hvor en anmeldelse kan oprettes.
4. Hvis guid er forkert (ukendt som firma-guid) bliver whistleblower bedt om at indtaste en guid igen.

Adgang til konfigurationsoplysninger

Whistleblower-applikationen har brug for at kunne tilgå oplysninger om databaselogin, hemmelige nøgler og andre ting som ændrer sig fra miljø til miljø. De oplysnger kan enten ligge i environment-variable eller konfigurationsfiler. Det håndteres i filen `settings.py` på denne måde:

```
if os.path.exists('/etc/config.json'):
    with open('/etc/config.json') as config_file:
        config = json.load(config_file)
        SECRET_KEY = config.get('SECRET_KEY')
        EMAIL_HOST_USER = config.get('EMAIL_USER')
        EMAIL_HOST_PASSWORD = config.get('EMAIL_PASS')
        AWS_ACCESS_KEY_ID = config.get('AWS_ACCESS_KEY_ID')
        AWS_SECRET_ACCESS_KEY = config.get('AWS_SECRET_ACCESS_KEY')
        AWS_STORAGE_BUCKET_NAME = config.get('AWS_STORAGE_BUCKET_NAME')
        DEBUG = config.get('DEBUG_VALUE')
        ALLOWED_HOSTS = config.get('ALLOWED_HOSTS')
        DATABASES = {
            'default': {
                'ENGINE': config.get('DB_ENGINE'),
                'NAME': config.get('DB_NAME'),
                'USER': config.get('DB_USER'),
                'PASSWORD': config.get('DB_PASSWORD'),
                'HOST': config.get('DB_HOST'),
                'PORT': config.get('DB_PORT'),
            }
        }
else:
    SECRET_KEY = os.environ.get('SECRET_KEY')
    EMAIL_HOST_USER = os.environ.get('EMAIL_HOST_USER')
    EMAIL_HOST_PASSWORD = os.environ.get('EMAIL_HOST_PASSWORD')
    AWS_ACCESS_KEY_ID = os.environ.get('AWS_ACCESS_KEY_ID')
    AWS_SECRET_ACCESS_KEY = os.environ.get('AWS_SECRET_ACCESS_KEY')
    AWS_STORAGE_BUCKET_NAME = os.environ.get('AWS_STORAGE_BUCKET_NAME')
```

```
    DEBUG = os.environ.get('DEBUG_VALUE')
    ALLOWED_HOSTS = os.environ.get('ALLOWED_HOSTS')
    DATABASES = {}
    if 'DYNO' in os.environ:
        DATABASES['default'] = dj_database_url.config(conn_max_age=600,
ssl_require=True)
    else:
        DATABASES = {
            'default': {
                'ENGINE': 'django.db.backends.postgresql',
                'NAME': os.environ.get('DB_NAME'),
                'USER': os.environ.get('DB_USER'),
                'PASSWORD': os.environ.get('DB_PASSWORD'),
                'HOST': os.environ.get('DB_HOST'),
                'PORT': os.environ.get('DB_PORT'),
            }
        }
```

Første del af if-sætningen indlæser data fra konfigurationsfilen hvis den eksisterer. Gør den ikke det, indlæser anden del af if-sætningen (else-delen) konfigurationsdata fra environment-variable.

Sikkerhed ved at benytte en GUID som login

Fremgangsmåden er sikker. Jeg bruger version 4 GUID'er, og de er baseret på et tilfældige tal. Ifølge Wikipedia (https://en.wikipedia.org/wiki/Universally_unique_identifier) kræver det mindst 2,71 x 10^{18} gæt for at opnå 50% chance for en enkelt kollision. Det er ikke en stor risiko da det vil tage ca. 85 år at lave så mange gæt, hvis man laver en milliard gæt i sekundet. En fil med så mange gæt ville fylde ca. 45 exabytes og det er meget.

Kodens placering på GitHub

https://github.com/mmmnaz/Whistle_blower

7. Opsætning af testmiljøet på Heroku

For at lave testmiljø for projektet har jeg valgt at lave testmiljø og testdatabase på cloud-leverandøren Heroku.

Heroku er en cloud-platforme som en tjeneste (PaaS), der understøtter flere programmeringssprog som f.eks. Python. Heroku er en af de første cloud-platforme og en udbyder for hosting og server. Man kan leje en server hos dem og sætter sit projekt hos dem online. At deploye et projekt på Heroku er gratis og man kan nemt sætte sit projekt online, men den har noget begrænsninger for de store projekter og man skal betale for at sætte store projekter op.

På Herokus hjemmeside https://devcenter.heroku.com/ kan man få et link til at downloade de filer man skal bruge til at forbinde til Heroku.

Efter at download og installere Heroku, kan man skrive ”Heroku –version” i bash og se at Heroku virker.

Den nedstående tabel viser beskrivelse og kommandoer der bliver brugt til at oprette en server hos Heroku og en Postgress-database i Heroku via Gitbash.

Beskrivelser	**Kommandoer**
1.Med Heroku login, åbnes Heroku-hjemmesiden derefter man registrerer sig og logger sig ind i Heroku hjemmesiden	`heroku login`
2.For at leje en server in i Heroku skrives:	`heroku create <et navn for website>`

Så får vi en ny hjemmeside-adresse på Heroku som man selv kan også navngivet.	
3.For at åbne hjemmesiden der lige er oprettet ind i Heroku:	`heroku open`
4.For at lægges projektet ind i Heroku, ind i roden af projektet skrives:	`git push heroku master`
5.De evt. fejl kan ses med	`heroku logs –tail`
6. Oprette en file med navnet Procfile ind i rod projektet og indskrives ”web: gunicorn config.wsgi” i Procfile-filen	`web: gunicorn config.wsgi`
7.Ind i settings.py under ALLOWED_HOSTS informere hjemmesiden man har på heroku fx: 'whistleblowertest.herokuapp.com'	

8. for at lave en ny sekret-key for projektet ind i root af projektet skreves: (Den sekret-key informeres til Heroku med næste skridt	`python`
	`>>import secrets`
	`>>secrets.token_hex(24)`

9. Når vi får en hemmelig nøgle, indsættes dette nøgle ind i heroku	`heroku config:set SECRET_KEY=”secret-token-hex24”`
10.Da vi behov for AWS for at sætte op Heroku så indtaster vi de AWS access-key og AWS bucket-navn ind	`heroku config:set AWS_STORAGE_BUCKET_NAME=”secret-token-hex24”`
11.For at installere Postgress database: (hobby-dev betyder den gratis version)	`heroku addons:create heroku-postgresql:hobby-dev`
12. For at se hvilken database man har på Heroku	`heroku addons`
13.For at se databasens information:	`heroku pg`
14. For at få alle de keys og andre ting på plads	`pip install django-heroku`
15. ind i setting.py skal tilføjes: import django_heroku django_heroku.settings(locals())	
16.For at migrere projektet-modeler til Heroku database	`heroku run python manage.py migrate`
17.Database er tomt. For at man kan komme ind i admin page skal man være superuser. For at oprette en superuser	`heroku run bash` `python3 manage.py createsuperuser`
18. For at gøre automation deploy to Heroku får en token med:	`heroku authorizations:create`

Derefter indsættes domain-navnet og den token vi fik fra Heroku ind i GitHub-action, ind i settings under ”secrets” tilføjes en ”new repository secret”.

Derefter sættes disse secrets ind i .yml-file for at automatisere deploy

.yml file

```
- name: Deploy to Heroku
      env:
        HEROKU_API_TOKEN: ${{ secrets.HEROKU_API_TOKEN }}
        HEROKU_APP_NAME: ${{ secrets.HEROKU_APP_NAME }}
      if: github.ref == 'refs/heads/master && job.status == 'success'
      run: |
        git remote add heroku
https://heroku:$HEROKU_API_TOKEN@git.heroku.com/$HEROKU_APP_NAME.git
```

```
git push heroku HEAD:master -f
```

8. Opsætning af produktionsmiljøet på Linode

Først opretter man sig i Linode hjemmeside og opretter en Linode server.

For at opsætte Linode er vi nødt til at have to åbne terminaler. En for vores lokale udviklingsmaskine og en for Linode serveren!

Handling	**Kommando**
1.Ssh ind på Linode-serveren	`ssh root@<linode-ip-adresse>`
2.Opdatere serverens software	`sudo apt-get update && apt-get upgrade`
3.Giv serveren et hostname	`hostnamectl set-hostname django-server`

4.For at Ubuntu og Linode-serveren kender hinanden,
Skriv serveren-hostname i hosts-fil
	`sudo nano /etc/hosts`
	
	`indsættes denne under linje med 127.0.0.1 localhost`
	`<Linode-ip-adresse> <tab> <hostname>`

5.Tilføj ny bruger som vi logger ind med i stedet for at være root	`adduser <brugernavn>`
6.Giv ny bruger rettighed til sudo	`adduser <brugernavn> sudo`

7.Log ud og ind igen som <brugernavn>
oprette directory
	`ssh <brugernavn>@<ipdaresse>`
	`mkdir -p ~/.ssh`
	`ls -la`

8.På lokal maskine og oprette ssh-key og kopi den i Linode-serveren.
Med ssh (secure shell) man kan secure connection, den er en encrypted protocol user til administrere.
	`ssh-keygen -b 4096 (nøgle længde)`
	`scp ~/.ssh/id_rsa.pub`
	`<brugernavn>@<ipadresse>:~/.ssh/authorized_keys`
	`less authorized_keys (for at læsse public nøgle)`

9.For at se ssh-key	`less authorized_keys`

10.Login på brugeren i Linode server.
Tjekke ændring af ssh-permission.
Til sidst restarter ssh!
	`sudo chmod 700 ~/.ssh/`
	`sudo chmod 600 ~/.ssh/*`
	`sudo nano /etc/ssh/sshd_config`
	`Sæt PermitRootLogin til no`
	`PasswordAuthentication sæt den til no`
	`sudo systemctl restart sshd`

11.Opsæt firewall ind i Linode-	`sudo apt-get install ufw` `sudo ufw default allow outgoing`

Brugeren server	sudo ufw default deny incoming sudo ufw allow ssh sudo ufw allow 8000 sudo ufw status
12.Læg projektet op på Linode-server fra Github	git clone eller scp -r django_project <brugernavn>@<Linode_ip_adresse>:~/
13.Før overflytning sikres som alle de nødvendige værktøjer er med	pip freeze > requirements.txt python -m pip freeze > requirements.txt
14.Ind user-Linode server Installer python pip	sudo apt-get install python3-pip
15.Installer python venv	sudo apt-get install python3-venv
16.Opret et ny virtuelt miljø	python3 -m venv django_project/venv
17.Aktiver virtuelt miljø	source venv/bin/activate
18.Installer pakker	pip install -r requirements.txt (skal være uploadet til server) sudo apt install python3-testresources sudo apt-get install libpq-dev
19.Ret Django-settings	sudo nano django_project/settings.py Tilføj <ip-adresse> til ALLOWED_HOSTS som tekststreng Tilføj STATIC_ROOT = os.path.join(BASE_DIR, 'static') lige over STATTIC_URL ='/static/'
20.Få statik-files til at fungere	python3 manage.py collectstatic
21.Kør udviklingsserveren og test	python3 manage.py runserver 0.0.0.0:8000
22.Stop serveren	Ctrl-C
23.Installer Apache web-serveren	sudo apt-get install apache2
24.Installer mod-wsgi	sudo apt-get install libapache2-mod-wsgi-py3
25.Konfigurer Apache webserver	cd /etc/apache2/sites-available/ sudo cp 000-default.conf django_project.conf sudo nano django_project.conf
26.Ind I django_project.conf Lige før det afsluttende </virtualHost>-tag ind i skreves:	Alias /static /home/<brugernavn>/<django_project>/static <Directory /home/<brugernavn>/<django_project>/static> Require all granted </Directory> Alias /media /home/<brugernavn>/<django_project>/media <Directory /home/<brugernavn>/<django_project>/media> Require all granted </Directory> <Directory /home/<brugernavn>/<django_project>/<core-project> <Files wsgi.py> Require all granted </Files> </Directory> WSGIScriptAlias / /home/<brugernavn>/<django_project>/<core_project>/wsgi.py

	WSGIDaemonProcess django_app python-path=/home/<brugernavn>/<django_project> python-home=/home/<brugernavn>/<django_project>/venv WSGIProcessGroup django_app
27.Aktiver site på Apache	sudo a2ensite django_project
28.Deaktiver defaultsite	sudo a2dissite 000-default.conf
29.Giv filrettigheder og med ls -la Kan man se rettighederne!	sudo chown :www-data ~/<django_project>/db.sqlite3 sudo chmod 664 ~/<django_project>/db.sqlite3 sudo chown :www-data ~/<django_project>/ sudo chmod 775 ~/<django_project>/ sudo chown -R :www-data ~/<django_project>/media sudo chmod -R 775 ~/<django_project>/media
30.Opretter en json-file	sudo touch /etc/config.json

31.Opsæt email-passwords m.v.
 	
Her er brugtes json men der er andre muligheder som yml og xml.
	sudo nano django_project/core_project/settings.py
	kopier værdien af SECRET_KEY til clipboard
	
	sudo nano /etc/config.json
	Skriv:
	{
	"SECRET_KEY": "<hemmelig nøgle fra settings.py>",
	"EMAIL_USER": "<emailkonto>",
	"EMAIL_PASS": "<email-password>"
	}

32.Configure settings.py	sudo nano <django_project>/<core_project>/settings.py **Skriv:** import json with open('/etc/config.json') as config_file: config = json.load(config_file) **Ret:** SECRET_KEY = config['SECRET_KEY'] DEBUG = False EMAIL_HOST_USER = config.get('EMAIL_USER') EMAIL_HOST_PASSWORD = config.get('EMAIL_PASS')
33.Luk for port 8000	sudo ufw delete allow 8000
34.Tillad http	sudo ufw allow http/tcp
35.Genstart Apacheserver	sudo service apache2 restart

HTTPS

Her er de skridt jeg brugte for at opsætte https på Linode web-server:

På denne hjemmeside https://letsencrypt.org skal trykkes på Cerbot og på Cerbot-siden valgte jeg webserver og operativsystemet man arbejder på. Jeg arbejder med Apache og Ubuntu 20.04 LTC derfor valgte jeg dem. Derefter vil dukke frem de kommandoer man skal bruge til at opsætte https på Linonode serveren.

- sudo snap install core; sudo snap refresh core
- sudo apt-get remove certbot, sudo dnf remove certbot, or sudo yum remove certbot.
- sudo snap install --classic certbot
- sudo ln -s /snap/bin/certbot /usr/bin/certbot
- sudo certbot --apache

- `sudo certbot certonly --apache`
- `sudo certbot renew --dry-run`

Opsætning af database Postgress på Linode:

- Log ind i root af Linode serveren: ssh root@172.105.74.176
- Og derefter ind i terminal har jeg indtaste de kommandoer
- `sudo apt update`
- `apt install postgresql postgresql-contrip`
- `update-rc.d postgresql enable`
- `Service postgresql start`
- `Service postgresql status`

Efter at postgres er started så skal konfitures configuration-files.

`cd ../etc/postgresql` og defter `ls` for at se hvilke filer man har. Navnet for filen betyder version af Postgress. Ind i mappen og igen med `ls` kommando kan man se der er "main" mappe, ind i "main" mappe med "`ls -la`" kan man se de konfigurations filer og derefter skal først konfigurere `pg_hba.conf` :

- `Sudo nano pg_hba.conf`

Ind i configuration-file skrev jeg i den sidste linje:

```
host    all       all             0.0.0.0/0               md5
```

Derefter skal jeg konfigurere postgresql.conf:

- `Sudo nano postgresql.conf`

Ind i konfigurations-file udkommandere jeg listen_addresses og slettet localhost og har lagt den lige med stigerne '*', så vil den høre for alle IP-adresse:

- `listen_addresses = '*'`

så skal postgresql restartet:

- `Service postgresql restart`

Jeg har brugt DBeaver Community for at kontakte to den database og se dataene eller tilføje noget database.

For at log ind som en postgres user i det samme directory som konfigurations filer var, skrives:

- `su – postgres`

Ind i postgres database for at lisen for databasen:

- `psql`
- `/l`

Så vil de databaser man har på Postgress blive vist.

Konfigurer en GitHub self-hosted-runner

For at sikre at ny kode blev indlæst på produktionsserveren, troede jeg, at jeg havde brug for at reloade Apache-serveren, og for at gøre det var jeg nødt til at have en runner installeret på serveren. Senere viste det sig, at jeg kunne nå en kode-reloading-task fra en remote-runner ved hjælp af en kommando "`rsync`", men jeg fandt ud af det så sent inden deadline, at jeg ikke havde tid til at ændre og teste opsætningen, og da den self-hosted-runner fungerede fint, holdt jeg fast på det.

For at oprette en self-hosted runner på Github-actions har jeg tilføjet en runner i settings under Actions. Derefter kræver det, at man vælger det environment man arbejder i. Jeg har valgt Linux X64 og så dukkede de kommander frem, som man skal bruge for at man kan downloade, konfigurere og udfører GitHub Actions Runner på sin egen server. Jeg har lavet disse skridt:

	Jobs	Kommandoer for at jobs udføres
1	Opret en mappe og ind i mappen	`mkdir actions-runner && cd actions-runner`
2	Download den nyeste runner-pakke	`curl -o actions-runner-linux-x64-2.278.0.tar.gz -L https://github.com/actions/runner/releases/download/v2.278.0/actions-runner-linux-x64-2.278.0.tar.gz`
3	Extract installationspakke	`tar xzf ./actions-runner-linux-x64-2.278.0.tar.gz`
4	Opret runner og start konfigurationsoplevelsen	`./config.sh --url https://github.com/mmmnaz/Whistle_blower --token ARHESGDUTQOO2T26UABKMXTAXFTRA`
5	For runner køre automatisk som service	`sudo ./svc.sh install`
6	For at start svc.sh filen	`sudo ./svc.sh start`
7	For at se svc.sh status	`sudo ./svc.sh status`

Konfiguration af applikationen på forskellige miljøer og servere

En Django-applikation er afhængig af en "`settings.py`" -fil, der skal konfigureres. En af konfigurationerne er database-secrets som password. En anden hemmelighed er den secret-nøgle, som Django bruger til ting som password reset-token, form-security, random-salts for password hashes og lignende.

Disse hemmeligheder kan ikke være en del af python-filen, da de muligvis skal skifte fra miljø til miljø (man vil ikke dele sin hemmeligheder mellem test og produktion). I stedet skal de opbevares i enten environment-variables eller en konfigurationsfil, der kan skifte fra miljø til miljø.

Applikationen køres på forskellige servere/computere og forskellige miljøer. Hvis applikationen kører på en linux-computer (som udviklingscomputeren eller produktionsserveren) kan konfigurationen af applikationen kan være filbaseret, da filsystemets struktur vil være tilgængelig og statisk, og der vil være adgang til katalog "`/etc`" i filsystemet.

Hvis applikationen kører på et server-as-a-service-system, som f.eks. Docker-containere fra GitHub-Actions eller som Heroku, kan konfigurationen holdes bedre i environment-variables, der kan fås adgang til fra environment, da der ikke vil være adgang til et "`/etc`" -katalog.

Selvfølgelig kan de statiske computere som produktionsserveren også indeholde environment-variables, men da jeg f.eks. har valgt at køre Apache som web-server, og da jeg ikke har været i stand til at finde den nødvendige dokumentation til opsætning af miljøvariabler, som kan deles med applikationen, med en

Apache-webserver, vendte jeg tilbage til en filbaseret konfiguration. På de statiske computere har jeg valgt at gemme konfigurationsvariabler i en fil kaldet "/etc/config.json".

På udviklingscomputeren, hvor jeg kører Ubuntu, ville miljøvariabler også være mulige, men jeg bruger den til udvikling af forskellige ting, så for at få en lettere konfiguration uden kollisioner bruger jeg også konfigurationsfilen her.

Konfigurationsfilerne kan ikke være en del af code-repository på GitHub, da det ville afsløre de hemmelige koder, for alle der har adgang til repositoriet. Konfigurationen af GitHub-hemmeligheder og Heroku foretages manuelt, og dette er ok, da dette kun gøres en gang under opsætningen.

konfigurationsfilen config.json er formateret sådan:

```
{
    "SECRET_KEY ": "Den hemmelige nøgle"
    "EMAIL_USER": "Brugeren E-mail-adressen"
    "EMAIL_PASS": "Adgangskoden til adgang til e-mail-serveren"
    "AWS_ACCESS_KEY_ID": " Adgangs-id'et til AWS"
    "AWS_SECRET_ACCESS_KEY": "Den hemmelige nøgle til adgang til AWS"
    "AWS_STORAGE_BUCKET_NAME": "Navnet af bucket på AWS for opbevaring
filerne"
    "DEBUG_VALUE": "Sand / falsk afhængigt af environment (falsk til
produktion)
    "ALLOWED_HOSTS": "En liste over hosts, som Django-app kan serve. Dette er
en sikkerhedsforanstaltning for at forhindre HTTP Host-header attacks"
    "DB_ENGINE": "Databasen f.eks. 'Django.db.backends.postgresql_psycopg2'
til en PostgreSql-database"
    "DB_NAME": "Navnet på databasen på databaseserveren f.eks. 'Wb' "
    "DB_USER": "Navnet på databasebrugeren"
    "DB_PASSWORD": "Databasebrugerens adgangskode"
    "DB_HOST": "Host-ip or host-name på  databaseserveren"
    "DB_PORT": "Porten der bruges af databaseserveren, til PostgreSql normalt
er 5432"
}
```

Continous Integration/Continous Delivery

Der er forskellige måder at lave en automatisk arbejdsgang til CI og CD. Mit mål har været at gøre integrationer og implementeringer automatiske, men triggered af specifikke events, så workflows startes manuelt men afvikles automatisk. Dette giver en god kontrol over, hvornår workflows kører, men også en automatisk workflows.

1. Automatiske workflows

Jeg har besluttet, at jeg vil have tre automatiske workflows:

1. En workflow, der kører mine tests automatisk, når jeg pusher til min feature branch.
2. En workflow, der kører mine tests automatisk, når jeg pusher til integrations branch, og som også kan bruges til at deploye automatisk til testmiljøet på Heroku
3. En workflow, der kører mine tests og pusher til produktion, når jeg vil have en ny version i produktionen

Udvikling af en funktion opdeles normalt i forskellige stories (f.eks. Lav modellen, lav Views, binde den sammen osv.), så det er fornuftigt at oprette en feature/story-branch til de feature/story, udvikle hver af de

stories og commit dem individuelt til feature-branch, og sørg for, at alle nye tests køres automatisk på hvert commit/push.

Når det besluttes, at integrations branch er klar til demo/test, pushes den til Heroku, hvor jeg har mit test-environment. Workflowet kører alle tests og deployer appen.

Endelig, når det besluttes, at appen på testmiljøet er klar til produktion, kan den automatisk deployes til produktion ved at merge integrations-branch til master og pushe til GitHub.

Forskellige team kan have forskellige måder at udvikle og committe software til repository, integrere software og release software til produktion. Jeg vil forsøge at lave min CI/CD-løsning på en måde, der understøtter forskellige arbejdsmetoder, men her vil jeg beskrive min måde, jeg foretrækker at udvikle flow på.

Jeg har en liste over features til at begynde med, og disse features er opdelt i stories, der kan udvikles på en dag. På den måde kan jeg levere min kode til repository i slutningen af hver dag. Dette krav på en dag vil være min forståelse af udtrykket "kontinuerlig" i CI og CD. Selvom jeg forventer at integrere kode dagligt, forventer jeg ikke nødvendigvis at frigive kode i produktion dagligt, men jeg forventer at være i stand til at gøre det.

Jeg vil have to branches, der vil være permanente i mit repository:

- integration
- master

Integration-branch skal indeholde al eksisterende kode i projektet og opdateres løbende med ny kode (forhåbentlig hver dag).

Master-branch skal indeholde koden til den release der er i produktion.

Disse to branches er de eneste langlivede branches i mit repository. De udviklings-branches, som udviklere bruger, hedder enten " feature<noget> " eller "story<noget>", som f.eks. ”feature_anonymous_login”. Disse er udviklere-branches, der pushes til repository for automatisk test, og efter automatisk test er passed (og måske med en pull-requests) merges i integrations-branch.

Flow af udvikling vil være sådan:

1. Udvikleren opretter en ny branch med navnet "feature<noget> eller "story<noget>" og udvikler en ny feature eller story inklusive tests af den nye kode som er skrevet.

2. Branch pushes til GitHub

3. Efter bestået automatisk statisk og dynamisk test merges branches sammen i integrations-branch. Hvis pull-request bruges, kan der også være manuel statisk test i form af en code-review.

4. Efter at have bestået automatisk test og nået en releasable tilstand pushes integrations-branch til Heroku, hvor den automatisk deployes og gøres tilgængelig til manuel accepttest.

5. Når udgivelsen accepteres og er klar til at gå i produktion, flettes integrations-branch med master (main). Dette vil udløse automatisk test, og hvis testen går godt, pushes release til produktionsmiljøet

At tale om "næste release " kan virke underligt i et kontinuerligt delivery-system, men mange virksomheder er ikke sikre på at frigive ny software til produktion i en kontinuerlig flow. De har muligvis brug for at softwaren skal bestå manuelle accepttest først, eller de skal muligvis træne personale eller kunder først, de skal muligvis synkronisere med juridiske krav eller marketings- behov eller hvad som helst, så begrebet "release" forekommer ret udbredt.

Hvis en virksomhed ønsker at levere kontinuerligt i produktionen uden risiko for at forstyrre personale eller kunder, kan det gøres på et par måder.

En måde er at man kan levere kontinuerligt til et produktions-environment, der ikke er det aktive produktions-environment, og når leverancerne er tilstrækkelige til en release, skiftes der mellem det inaktive-produktions-environment og det aktive environment, for eksempel ved at pege en url fra nuværende aktive environment til det nye produktions-environment. Dette kan forstås som at adskille delivery fra deployment til produktion. Jeg har set måden at levere sådan kaldet "dark launching".

En andet måde er, i stedet for at have et duplikeret produktions-environment kan man levere hver af de opdaterings-feature til det virkelige produktions-environment kontinuerligt, men beskyttet bag et sæt af feature-toggles, hvilket betyder, at release-features ikke er aktive, hvis feature-toggle er false, men er aktiv, når feature-toggle er true. På den måde findes mange komplette releases muligvis i samme environment, men det er kun de released med enabled-features (feature-toggle = True) der vil være synlige for brugerne.

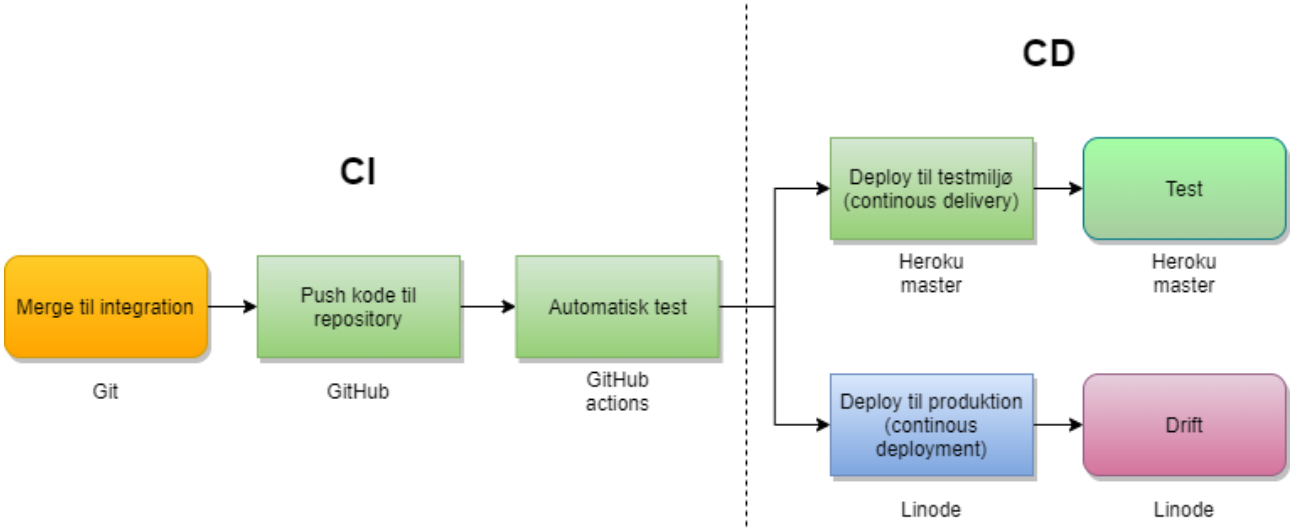
Den tredje måde hedder Canary-testing. En canary-testing-release introducerer ændringerne til en lille gruppe af brugere på produktion for at teste de nye funktioner. Når resultaterne er tilfredsstillende, udgives nye version til alle bruger. På den måde reducerer risikoen for at introducere en defekt for alle brugere. Her skal man være sikker på at de nye funktioner ikke påvirker tilstanden på data så man ikke kan slukke for dem igen.

Alle måder for at udføre continuous-delivery, er der fordele og ulemper ved. Den første metode er dyrt på grund af et duplikeret environment, men det kan være lettere at håndtere, hvis environment eller databaserne skal opdateres. I den anden opstår let et spørgsmål om, i hvilken tilstand softwaren testes, og det kan være sværere at håndtere ændringer i environment og databaser. Det kan også være vanskeligere at analysere og styre, hvilke toggles der kan være aktive samtidigt. Det kræver til gengæld kun et enkelt produktions-environment.

Der kan være flere måder, men de tre måder at gøre det på har jeg fundet på https://www.ibm.com/garage/method/practices/run/practice_dark_launch_feature_toggles/

Jeg har ikke implementeret nogen af disse metoder i dette projekt.

Den pipline-architectture, jeg vil oprette, er noget som vist i dette diagram.



2. GitHub Actions

I GitHub Actions er workflow-scriptet skrevet i YAML-formatet og gemt i en YAML-fil (skal have suffikset .yml eller .yaml), der skal være i .gihub/workflows-directory.

Et workflow-script består af de job, der kan have separate ansvar, og hvert job består af trin. Hver Job har et navn og en runner, der er en maskine til at køre jobbet. Runner kan enten være en GitHub-docker-container med et operativsystem installeret, eller det kan være en remote-runner, som er en runner, der kører på en computer efter eget valg. Jobs køres som standard parallelt, men kan også køres i rækkefølge, hvis det er **specificere**t Der er mange flere detaljer i GitHub Actions, og de er meget veldokumenterede på siden https://docs.github.com/en/actions.

3. Continous Integration

Flere gange om dagen pusher udviklere kode til repository. Man kan oprette et script til at opbygge og teste applikationen automatisk, hvilket mindsker chancen for at der dukker en fejl op i appen. Denne praksis er kendt som kontinuerlig integration; Det er vigtigt for projekter med hyppige ændringer fordi når projektet skal integreres kan der ske mange integrationskonflikter og kan tage for meget tid at løse, hvilket fører til forsinkelser i projektet, men med CI notificeres teamet med det samme når test i integration fejler.

Jeg vil gennemgå hver af de tre yaml-filer, jeg har konstrueret, og give en forklaring på indholdet.

4. CI-workflow

Workflow er i filen “feature.yml”

```
name: Whistle_blower stories
on:
  push:
    branches: [ 'story*', 'feature*' ]
  pull_request:
    branches: [ 'story*', 'feature*' ]
jobs:
  unit-integration-test:
    runs-on: ubuntu-latest

    services:
```

```
      postgres:
        image: postgres
        env:
          POSTGRES_PASSWORD: postgres
        options:
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
        ports:
          - 5432:5432
    env:
      SECRET_KEY: ${{ secrets.SECRET_KEY }}
      DB_ENGINE: django.db.backends.postgresql_psycopg2
      DB_HOST: localhost
      DB_NAME: postgres
      DB_PASSWORD: postgres
      DB_PORT: 5432
      DB_USER: postgres
      DEBUG_VALUE: True
      ALLOWED_HOSTS: '*'
      AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
      AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      AWS_STORAGE_BUCKET_NAME: ${{ secrets.AWS_STORAGE_BUCKET_NAME }}
      EMAIL_PASS: ${{ secrets.EMAIL_PASS }}
      EMAIL_USER: ${{ secrets.EMAIL_USER }}

    steps:
      - uses: actions/checkout@v2
      - name: Set up Python 3.9
        uses: actions/setup-python@v2
        with:
          python-version: 3.9
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
      - name: Lint with flake8
        run: |
          flake8 case --count --select=E9,F4,F6,F8 --show-source --statistics
          flake8 caseworker --count --select=E9,F4,F6,F8 --show-source --
statistics
      - name: Migrate database
        run: |
          python manage.py migrate
      - name: Run pytest
        run: |
          pytest
```

Jeg vil ikke gå ind i alle detaljer, da jeg tror, de kan være selvforklarende, men vil bare nævne, at dette workflow kun har et enkelt job med navnet "`unit-integration-test`", og dette job kører på en GitHub docker runner med den sidste version af Ubuntu installeret.

Workflowet aktiveres, når der sendes en push- eller en pull-request-anmodning direkte til en branch, der hedder enten "story*" eller "feature*" (f.eks. "story_case_model").

Containeren leverer også en PostgreSQL-service, og alle connection-oplysninger er indtastes i environment-variables, som Django-applikationen kan få adgang til.

Source-kode tjekkes ud fra repository, og Python opsætter og installerer alle de samme pakker som er installeret under udvikling (disse er skrevet i requirements.txt-file). Linteren køres, og derefter migreres databasen. Endelig køres alle tests med Pytest, og det endelige resultat af testene vises på GitHub Actions-konsollen.

Dette script giver enhver udvikler mulighed for at oprette en ny branch med et navn med "story *" eller "feature *" og få dem testet, hver gang de pushet til GitHub.

Når udvikleren pusher den endelige commit, og testene er bestået, kan der sendes en pull-request mod "integration"-branch. Da jeg ikke bruger pull-requests, merger jeg bare den nye branch til "integration" og pusher til GitHub.

Dette push starter den næste workflowi filen “integration.yml”.

```
name: Integration workflow
on:
  push:
    branches: ['integration']
  pull_request:
    branches: ['integration']

jobs:
  build:
    services:
      postgres:
        image: postgres
        env:
          POSTGRES_PASSWORD: postgres
        options:
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
        ports:
          - 5432:5432
    env:
      SECRET_KEY: ${{ secrets.SECRET_KEY }}
      DB_ENGINE: django.db.backends.postgresql_psycopg2
      DB_HOST: localhost
      DB_NAME: postgres
      DB_PASSWORD: postgres
      DB_PORT: 5432
      DB_USER: postgres
      DEBUG_VALUE: True
      ALLOWED_HOSTS: '*'
      AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
      AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      AWS_STORAGE_BUCKET_NAME: ${{ secrets.AWS_STORAGE_BUCKET_NAME }}
      EMAIL_PASS: ${{ secrets.EMAIL_PASS }}
      EMAIL_USER: ${{ secrets.EMAIL_USER }}

    runs-on: ubuntu-latest
    strategy:
      matrix:
```

```
        python-version: [3.8]
    steps:
    - uses: actions/checkout@v2
    - run: |
        git fetch --prune --unshallow
    - name: Set up Python 3.9
      uses: actions/setup-python@v2
      with:
        python-version: 3.9
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Run migrations
      run: python3 manage.py migrate
    - name: Run pytest
      run: |
        pytest
    - name: Lint with flake8
      run: |
        # dont stop the build if there are syntax errors in case or caseworker
        flake8 case --count --select=E9,F4,F6,F8 --show-source --statistics
        flake8 caseworker --count --select=E9,F4,F6,F8 --show-source --
statistics
    - name: Deploy to Heroku
      env:
        HEROKU_API_TOKEN: ${{ secrets.HEROKU_API_TOKEN }}
        HEROKU_APP_NAME: ${{ secrets.HEROKU_APP_NAME }}
      if: github.ref == 'refs/heads/integration' && job.status == 'success'
      run: |
        git remote add heroku
https://heroku:$HEROKU_API_TOKEN@git.heroku.com/$HEROKU_APP_NAME.git
        git push heroku HEAD:integration -f
```

Dette workflow kaldes "Integrationsworkflow" og vil svare på pull-requests eller push til integrations-branch.

Dette workflow kører den samme linting og test som den foregående, men i en anden rækkefølge. Der er ingen grund til den anden rækkefølge, og det påvirker ikke slutresultatet.

I slutningen kontrollerer det sidste trin i scriptet ("deploy til Heroku"), om det er pushet til Heroku-master, og hvis ikke, afsluttes jobbet, og hvis alle testene gik godt, er integrationen udført.

5. Continuos Delivery

Dette et skridt ud over kontinuerlig integration og kan bruges når kontinuerlig integration slutter Det samme script som i integrationen bruges, men denne gang pushes det til Heroku-mester i stedet for GitHub kommandoen er:

```
git push Heroku integration:master
```

og dette vil pushe til Heroku. Her vil den sidste del af scriptet opdage at der er pushet til Heroku og at testene gik godt og starte den faktiske levering til Heroku Dyno og offentliggøre applikationen på en url:

https://m-whistleblower.herokuapp.com/

hvor den kan demonstreres og testes manuelt.

Som vist er levering og test automatiseret, men det udløses manuelt ved at pushe til Heroku. Dette giver kontrol til udviklerne eller andre personer, der har brug for, at testmiljøet holdes i en bestemt tilstand, inden de går videre til en ny release, f.eks. fordi de er i gang med at fejlsøge.

6. Continuos Deployment

Det sidste trin i udviklingsarbejdsprocessen er deploy til produktion. Dette kan opnås ved at merge integrations-branch til master eller hoved-branch (Jeg bruger "master" som min produktions-branch, men GitHub har ændret standard til "main" i stedet for "master") og derefter pushe det til GitHub. Dette vil udløse det tredje workflow som er det sidste workflow.

```
name: Production workflow
on:
  push:
    branches: ['master','main']
  pull_request:
    branches: ['master','main']
jobs:
  test:
    services:
      postgres:
        image: postgres
        env:
          POSTGRES_PASSWORD: postgres
        options:
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
        ports:
          - 5432:5432
    env:
      SECRET_KEY: ${{ secrets.SECRET_KEY }}
      DB_ENGINE: django.db.backends.postgresql_psycopg2
      DB_HOST: localhost
      DB_NAME: postgres
      DB_PASSWORD: postgres
      DB_PORT: 5432
      DB_USER: postgres
      DEBUG_VALUE: True
      ALLOWED_HOSTS: '*'
      AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
      AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      AWS_STORAGE_BUCKET_NAME: ${{ secrets.AWS_STORAGE_BUCKET_NAME }}
      EMAIL_PASS: ${{ secrets.EMAIL_PASS }}
      EMAIL_USER: ${{ secrets.EMAIL_USER }}
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Python 3.9
        uses: actions/setup-python@v2
        with:
          python-version: 3.9
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install flake8
          if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
```

```
      - name: Lint with flake8
        run: |
          flake8 case --count --select=E9,F4,F6,F8 --show-source --statistics
          flake8 caseworker --count --select=E9,F4,F6,F8 --show-source --
statistics
      - name: Pytest
        run: |
          pytest
  deploy:
    needs: [ test ]
    runs-on: self-hosted
    steps:
      - name: Deploy
        run: |
          cd /home/admin/Whistle_blower
          rm -rf staticfiles
          git pull
          source venv/bin/activate
          if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
          python manage.py collectstatic
          python manage.py migrate
          touch /home/admin/Whistle_blower/config/wsgi.py
```

Dette workflow gentager de tidligere workflow med hensyn til test, men har et ekstra job kaldet "deploy". Dette job er ansvarligt for deploy til produktion, og det har en række specielle ting, der er værd at nævne.

Jobbet venter på, at test-jobbet er afsluttet inden start, så disse job kører i rækkefølge efter hinanden og ikke parallelt.

Det kører på en self-hosted-runner, der i dette tilfælde er en runner, der er installeret som en service på Linode Ubuntu-webserveren. Denne runner overvåger hvornår dette workflow køres, og reagerer ved at køre de viste kommandoer.

Kommandoerne er almindelige bash-kommandoer. For at deploye skal jeg først rydde op i de statiske filer (html-filer) for ikke at git brokker sig senere. Det gøre jeg ved at slette dem med kommandoen ”`rm -rf staticfiles`”.

Derefter skal jeg pulle koden fra repository, og da jeg kører Python, har jeg ikke artefakter eller et build-step. Python-koden kører som den er.

Efter at have downloadet (med `git pull`) applikationskoden, aktiverer jeg et virtuelt Python-environment og installerer de nødvendige ekstra pakker (skrevet i requirements.txt).

Jeg indsamler staticfiles til Apache med kommandoen `python manage.py collectstatic` og migrerer databasen med kommandoen `python manage.py migrate`.

Endelig genindlæser jeg Python-koden ved touching af wsgi.py. Denne måde at genindlæse Python-koden på, virker fordi jeg har installeret mod_wsgi som en service. Ellers skulle jeg have genstartet Apache. Dette kunne gøres med kommandoen:

```
sudo -s systemctl restart apache2.service < /home/admin/Whistle_blower/password.txt
```

Jeg prøvede dette, og det fungerede fint, men det anbefales ikke at have en adgangskode, der ligger i klar tekst, selvom serveren er hærdet mod adgangskodelogins, som min server er. Især ikke når GitHub-

repository er offentligt og kan forkes. Jeg forsøgte også at konfigurere, at admin ikke skulle have brug for en adgangskode til sudo, men det kom aldrig til at fungere.

Faktisk advarer GitHub om aldrig at bruge self-hosted-runners på et offentligt repository https://docs.github.com/en/actions/learn-github-actions/security-hardening-for-github-actions, men da dette er til mit bachelorprojekt, og jeg ikke var sikker på, hvem der havde brug for adgang, gjorde jeg det alligevel. Jeg har ikke brugt mere tid på sikkerhedshensyn i dette projekt.

Test

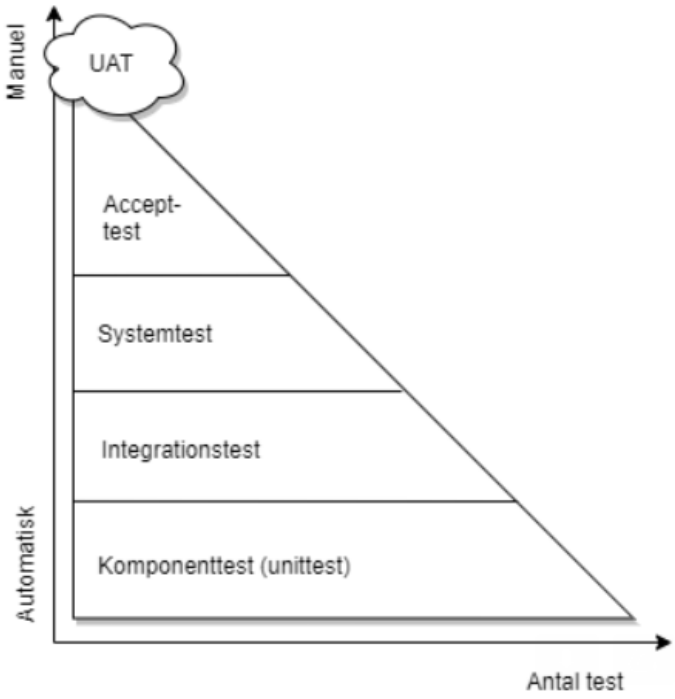
1. Indledning

Softwaretest kan vise kvaliteten af software. For at være i stand til at integrere og levere software kontinuerligt skal der også være kontinuerlig test så man hele tiden kender kvaliteten, og en stor del af testen skal være automatisk så man sikrer at testen gennemføres hver gang den skal.

På den anden side kan test normalt ikke automatiseres komplet. Dette vil være meget dyrt og kan være meget vanskeligt at gennemføres fordi automatiske test skal omskrives når selv små rettelser laves i softwaren. Automatiseret test udføres også gennem softwareudvikling (testcases implementeres i softwaren) og bør også testes for at sikre, at software fungerer som det forventes. Fordelene ved fuld automatisering af alle test overstiger muligvis ikke omkostningerne ved fuld automatisering, så det er vigtigt at tillade nogle manuelle test.

2. Testtyper og automatisering

En almindelig måde at se test på i agil udvikling med devops er testpyramiden. Dette viser forskellige lag af testniveauer, og hvor stor indsats hvert niveau skal modtage.



Figur 1 Testpyramiden

Allan Kelly har tegnet pyramiden i et diagram, som jeg synes bedre forklarer pyramidens hensigt https://dzone.com/articles/testing-triangle-circle-and.

Han siger, at da de fleste Unit-tests køres meget hurtigt, kan de udføres ofte, men da kompleksiteten stiger gennem niveauerne, vil test køres sjældnere for at holde udførelsestiderne balancerede. Han argumenterer også for, at manuel test stadig skal anvendes på mindst det højeste niveau (User Acceptance Tests), da det f.eks. kan være vanskeligt at automatisere disse tests.

ISTQB – International Software Testing Qualifications Board definerer følgende testniveauer i Tester Foundation Level Syllabus version 2018 v3.1:

Niveau	Testfokus
Komponenttest	Komponenter, der kan testes separat (f.eks. Klasser eller funktioner)
Integrationstest	Interaktion mellem komponenter eller systemer
Systemtest	Opførsel af hele systemfunktioner og end2end-opgaver
Test af brugeraccept	Det er meget som systemtest, men mere fokuseret på hele systemadfærd, test om systemet er komplet og fungerer som forventet!
Test af operationel accept	Test af ikke-funktionelle krav i hele systemet (f.eks. Backup / gendannelse, katastrofegendannelse osv.)

Jeg bruger ISTQB-niveauerne i mit arbejde, bortset fra at jeg muligvis bruger forskellige ord til nogle niveauer.

En ting, der normalt udelades i diskussioner om automatisk test, jeg har set på Internettet, er manuel statisk test (statisk test betyder at der testes uden kodeudførelse) foregår i kodevurderinger, for eksempel når du bruger GitHub og Pull-request med manuelt review.

Jeg bruger ikke pull-request, da jeg udvikler alene, men jeg har arbejdet med pull-request tidligere, hvor vi var flere udviklere der leverede til det samme system. Pull-request bruges som regel til at sikre, at kvaliteten af koden er tilstrækkeligt inden koden merges, fordi et pull-request gøre at man kan review koden inden man merger.

Reviewprocessen er normalt afhængig af resultaterne af automatiserede tests for at sikre kvaliteten er opfyldt, og gennemgangen undersøger også, om der foretages nok unit-tests. Statisk test kan automatiseres til en vis grad af linters som f.eks. flake8 til Python og disse vil dække et antal mulige kodningsfejl, men gennemgangen i Pull-Request flow kan teste for mere end linters. Det kan for eksempel være aftalt praksis af udviklerne, at de ikke bruger rekursion ved kodning, og det ville en linter ikke opdage.

3. Linting

Linting er et værktøj der kontrollerer kodesyntaks og giver vejledning om, hvordan du rette det. Linting hjælper med at kontrollere mod kodningsstandarder og hjælper med at forhindre ting som syntaksfejl, skrivefejl, dårlig formatering, forkert styling osv. Så derfor hjælper linting os med at spare tid som for folk, der gennemgår vores kode. Jeg bruger Flake8 til at teste koden for at se den er kompatibel med PEP-8 (Python Enhancement Proposal 8) style-guide https://www.python.org/dev/peps/pep-0008/.

Der er andre lintere (som pep8, autopep8 og pylint), men jeg synes egentligt, at Flake8 passer fint til formålet. PEP-8-guidens mål er at gøre koden så læsbar som muligt, da Guido Van Rossum, som har skabt Python, har udtalt:

"**The code is read much more often than it is written**."

Selvom jeg for det meste vil være kompatibel med PEP-8, finder jeg det ikke nødvendigt at være fuldt overholdende, og der er situationer, hvor jeg finder koden mere læselig, hvis kompatibilitet af guiden ikke er obligatorisk. Et eksempel taget fra Django-filen 'markers.py':

```
VARIABLE = (
    L("implementation_version")
    | L("platform_python_implementation")
    | L("implementation_name")
    ...)
```

vil resultere i en Flake8-error "`W503: line break before binary operator`". Jeg synes faktisk, at disse linjeskift her er meget rart, når det kommer til læsbarhed (det er let at få et overblik over alle forhold), så jeg tror ikke, at der er behov for en error i dette specifikke tilfælde.

For at undgå at Flake8 rapporterer om disse og andre specifikke overtrædelser af kodestandarder, har jeg konfigureret, at det kun skal svare på et specifikt sæt cases. Konfigurationerne er opdelt mellem en konfigurationsfil, "setup.cfg" i projektmappen og kommandolinjen:

```
flake8 case --count --select=E9,F4,F6,F8 --show-source –statistics
```

Den specifikke betydning af disse kommandolinjemuligheder forklares på Flake8- error-code pages https://flake8.pycqa.org/en/latest/user/error-codes.html and https://pycodestyle.pycqa.org/en/latest/intro.html#error-codes og andre Flake8 sider. Når jeg skriver f.eks.

–select = F4 betyder det, at jeg vil have Flake8 til at rapportere om overtrædelser F401 til F407. Hvis jeg kun ville have en advarsel om en af dem, kunne jeg bare specificere det i stedet.

Meget af Django-koden er ikke i kompatibel med PEP-8, så for at undgå meget støj in linting har jeg besluttet kun at linte den kode, jeg selv har skrevet i apps, og også at begrænse de rapporterede fejl til et specifikt sæt. De fejl, jeg har valgt at have rapporteret, er F4xx, F6XX, F8XX og E9XX. Disse fejl er beskrevet detaljeret på https://flake8.pycqa.org/en/latest/user/error-codes.html og https://pycodestyle.pycqa.org/en/latest/intro.html#error-codes.

Der er mange flere fejl at linte for, men jeg synes, det er et rimeligt udvalg fordi jeg bruger en editor der hedder Pycharm, der allerede gør et godt arbejde, med at minde mig om, når jeg træder uden for PEP-8.

4. Code-coverage

En vigtig ting ved testning er kodedækningen, altså hvor meget kode der er dækket af test. Jeg startede med at teste og analysere dækningen ved hjælp af værktøjet Coverage, men efter at have skiftet til Pytest behøvede jeg ikke længere at bruge Coverage, da Pytest også kan beregne Coverage.

Ideelt set skulle kodedækningen være høj, og i dette tilfælde er dækningen 100%.

```
---------- coverage: platform linux, python 3.8.5-final-0 ----------
Name                                 Stmts   Miss Branch BrPart  Cover
-----------------------------------------------------------------------
tests/conftest.py                       35      0      0      0   100%
tests/test_case_forms.py                25      0      0      0   100%
tests/test_case_models.py               41      0      0      0   100%
tests/test_case_url.py                   8      0      0      0   100%
tests/test_case_views.py                61      0      0      0   100%
tests/test_caseworker_forms.py          29      0      0      0   100%
tests/test_caseworker_models.py         55      0      0      0   100%
tests/test_caseworker_url.py            68      0      0      0   100%
tests/test_caseworker_views.py          14      0      0      0   100%
-----------------------------------------------------------------------
TOTAL                                  336      0      0      0   100%
```

Figure 2 Resultat af code coverage analyse

Kodedækning måles som antallet af statement, der er udført ved testene divideret med det samlede antal executable-statement i testobjektet, normalt udtrykt i procent. Når der opnås 100% dækning af statementene, sikrer det, at alle executable-statement i koden er blevet testet mindst én gang, men det sikrer ikke, at al beslutningslogik er blevet testet. Mit mål er 100% dækning af statements og beslutninger og det ser ud til at være nået fra ovenstående skema.

Unittest

Unittest er kode som kører anden kode for at tjekke om kodens opførsel som er forventet!

Jeg bruger ordet "unit test" til det ISTQB kalder "komponenttest", fordi det er det ord, jeg har hørt mest, når jeg har læst på internettet om kodetest.

Unit-testing handler om at teste koden, hvor en "Unit" er en enkelt funktion eller en enkelt klasse.

Unit-tests viser, om koden fungerer, som jeg vil, og vil blive automatiseret. Dette betyder, at jeg kan køre det automatisk, når jeg pusher kode til GitHub.

For at finde ud af, hvordan koden forventes at fungere, spørger jeg mig bare, da det var mig, der besluttede, hvilken kode jeg skulle skrive for at adressere en bestemt problem.

Jeg har i et par tilfælde brugt datadrevet test til at teste f.eks. url-respons. Det kan man gøre sådan her:

```
url_data = [
    ('caseworker:register', 302),
    ('caseworker:logout', 302),
    ('caseworker:login', 200),
]

@pytest.mark.parametrize("u, expected", url_data)
def test_logged_views(client, u, expected, user_data1):
    temp_url = urls.reverse(u)
    resp = client.get(temp_url)
    assert resp.status_code == expected
```

I det her eksempel har jeg kun skrevet en enkelt testcase som bliver kaldt med alle de data der er i `url_data` så det virker som 3 test. Det er en stor fordel at man kan lave mange testcase ved bare at ændre data.

Integrationstest

Integrationstest er test af systemet, når unit-test er integreret, eller når større moduler er integreret (for eksempel kan jeg få adgang til databasen). Disse tests tester også koden for at se, om den kører som forventet, men nu tester den koden, der fungerer sammen noget af koden der blev testet med unittests, der tester koden isoleret.

Systemtest

Systemtest er som regel at man laver

- Arrange
- Act
- Assert

Man starter med at opsætte sin test, så gør man noget og så ser man om resultatet er ok.

Systemtest handler om at teste, om systemet fungerer som forventet, mere end at teste, om koden fungerer som forventet. Koden fungerer muligvis perfekt, men hvis det er den forkerte kode, fungerer systemet muligvis ikke som det skal.

For at finde ud af, hvordan systemet skal fungere, kan jeg se på kravene og analysen af mit systemdesign.

Acceptttest

Acceptanstest udføres på højeste niveau. Det kan udføres manuelt på Heroku-systemet, eller det kan udføres automatisk via browseren ved hjælp af en browserdriver som Selenium. Acceptstest skal aftales fra begyndelsen, fordi det er den test, kunden udfører for at afgøre, om systemet er acceptabelt.

Testfiler

For at kunne have bedre overblik over de tests, som jeg har skrevet i Django web framework, besluttede jeg at oprette separate Python-filer for at teste hver del programmet.

Mine test ligger i disse filer i directory ”tests”, hvor ”conftest.py” indeholder de fixtures jeg bruger i mine test:

conftest.py
test_case_forms.py
test_case_models.py
test_case_url.py
test_case_views.py
test_caseworker_forms.py
test_caseworker_models.py
test_caseworker_url.py
test_caseworker_views.py

5. Pytest

Pytest kan køre test der er lavet til den indbyggede test runner, men også dem der er lavet til Pytest selv. At skifte til brug af Pytest betyder ikke, at jeg skal ændre alle test til et nyt format.

Årsagen til, at jeg skiftede til Pytest, var den nemme brug af fixtures og de lettere asserts.

I den indbyggede test-runner bruger man statements som "assertEqual (a, b)". I Pytest ville det samme være “assert a==b”. Jeg har lettere ved ikke at skulle huske alle de forskellige asserts i den built-in test-runner, men bare at huske " assert " og udtrykke logikken, som jeg er vant til.

Pytest har automatisk-opdagelse af tests, så hvis filer er præfikset med "test_", bliver de brugt til test. En del af resultatet af test kørt i Pytest ser sådan ud:

```
tests/test_caseworker_url.py::test_user_login PASSED
tests/test_caseworker_url.py::test_user_logout PASSED
tests/test_caseworker_url.py::test_user_detail PASSED
tests/test_caseworker_url.py::test_case_delete PASSED
tests/test_caseworker_url.py::test_user_create_db_data PASSED
tests/test_caseworker_url.py::test_user1_create_db new_user2:   Test_user
PASSED
tests/test_caseworker_url.py::test_user2_create_db new_user2:   Test_user
PASSED
tests/test_caseworker_url.py::test_user_db_not_data PASSED
tests/test_caseworker_views.py::TestCaseworkerRegister::test_register_Post_correct self.detail_url :::   /caseworker/register/
PASSED
tests/test_caseworker_views.py::TestCaseworkerRegister::test_register_Post_wrong PASSED
tests/test_caseworker_url.py::test_logged_views[caseworker:register-302] PASSED
tests/test_caseworker_url.py::test_logged_views[caseworker:logout-302] PASSED
tests/test_caseworker_url.py::test_logged_views[caseworker:login-200] PASSED
tests/test_caseworker_url.py::test_logged_views[case:caseinfo-view-302] PASSED
tests/test_caseworker_url.py::test_render1_views[caseworker:register] PASSED
```

Figure 3 Eksempel på resultat af testkørsel

Nogle tests kan forårsage, at programmet kører langsommere. Med den kommando `”pytest --durations=3”` kan man se hvor lang tid de tre første testcases har taget at køre.

Nogle af de test som kræver adgang til og forbindelse til databasen, kan man markere dem ved

@pytest.mark.database_access, og man kører det med den kommando:

- `Pytest -m database-access`

Python søger for en række symboler:

- Skip: som ubetinget ignorerer en test.
- Skipif som ignorerer testen, hvis betingelsen er true.
- Xfail som betyder, at testen skal mislykkes, så selvom testen mislykkes, vil hele testsættet køre med succes.
- Parametrize som skaber forskellige tests med forskellige værdier som input

Fixture

Fixture i pyton er funktioner, der genererer data eller duplikerede tests eller opretter systemstatus for et sæt af tests. Man kan importere fixture in i anden fixture og dette giver os stor flexibilitet. Enhver test, der afhænger af en fixture, skal eksplicit have den fixture som input.

Pytest ser efter conftest.py-modulet i hele mappestrukturen. Hvert conftest.py-modul indeholder indstillinger for filer, som Pytest vil finde. Det er bedste sted hvor fixtures kan lægges.

```python
# conftest.py

import pytest
import requests

@pytest.fixture(autouse=True)
def disable_network_calls(monkeypatch):
    def stunted_get():
        raise RuntimeError("Network access not allowed during testing!")
    monkeypatch.setattr(requests, "get", lambda *args, **kwargs: stunted_get())
```

Figure 4 Et eksempel på en testfixture

Med autouse=True kan man sikre sig, at netværksanmodningen er deaktiveret for hver af testene i denne testpakke.

Pytest giver mange muligheder for at filtrere og forbedre tests Ved plugins der øger deres værdi,

- Fixture er til styring af afhængigheder, status og genanvendelighed.
- Mærke er for at klassificere test og begrænse adgang til ressourcer.
- Parametrering (Parametrization) for at reducere duplikatkode mellem test.
- Tidsadministration (Durations) for at registrere den langsomste test.

Refleksion

Jeg fik kravene til en prototype af en whistleblower-applikationen fra et firma, jeg havde talt med. At implementere den applikation viste sig at være mere arbejde, end jeg havde forventet, og jeg er ikke blevet helt færdig med opgaven, men jeg synes, jeg er kommet et godt stykke ad vejen, og det jeg har lavet har været fint til at vise effekten af mine automatiske workflows.

Det har været svært at arbejde alene, men jeg startede med at skabe en god platform med et godt IDE (jeg bruger PyCharm) og en god hjemmearbejdsplads, og det hjalp med at motivere for det daglige arbejde. Min vejleder var tilgængelig for møder hver uge og det hjalp mig meget med at sætte et fremdriftsmål hver uge.

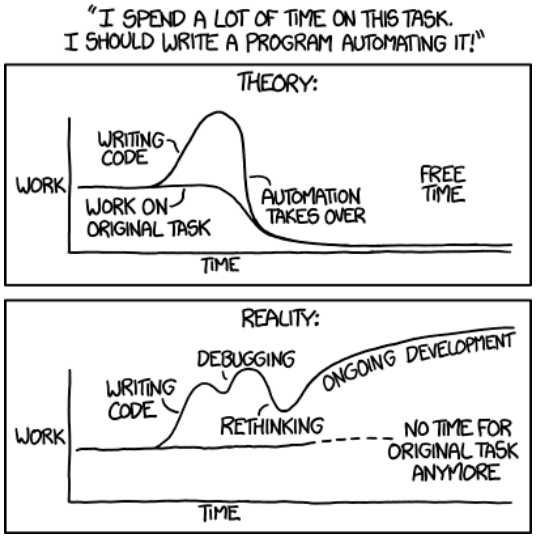


Figure 5 XKCD om automatisering af opgaver

Jeg var fra starten meget optaget af at lave en applikation og mindre optaget af at lave de automatiske CI/CD-flow. Det fortryder jeg, for havde jeg fra starten fået mine flow til at fungere, havde det været meget nemmere for mig at få udviklet applikationen.

Jeg valgte at benytte Heroku og det var en god ide, for Heroku er nem at få til at fungere og skal man skalere til højere ydelse er det ikke noget man selv skal bøvle med. Det kan Heroku. På Linode skal man selv kunne mange flere ting om servere, operativsystemer og load balacing og det er ikke så nemt når man nu er gladere for at programmere.

Django er en rigtig fin platform for en webapplikation. Jeg har tidligere erfaringer med React og andre frameworks, men Django er utrolig moden, dokumentationen er nem at finde og der er et stort community omkring Django, som hjælper meget når man skal løse svære ting. Til gengæld er Django også meget omfattende og det kan være nemt at fortabe sig i detaljer fordi man ikke har et overblik over frameworket og de muligheder der findes.

Konklusion

Jeg ville lave en CI/CD-chain for at opleve fordelene ved at bruge en CI/CD-chain i praktisk udviklingsarbejde og få nogle erfaringer med at arbejde med automatiske workflows og automatisk test under udvikling.

Jeg designede tre workflows til at tage sig af tre faser i udviklingen: udvikling af feature/story, integration til eksisterende kode og deploy til test samt deploy til produktion. Alle tre arbejdede som forventet og hjalp til at forbedre min udviklingsproces.

Jeg testede applikationen med automatisk test, og det viste sig at fungere rigtig godt i et CI/CD-setup. Jeg fik en rimelig dækning af koden og applikationen, men jeg fik desværre ikke tid til at teste alt ordentligt. Men jeg fik ifølge Pytest lavet 74 automatiske test, som testede 336 statement med en dækning på 100% af statements og branches.

Samlet set må jeg sige at automatiske workflows er geniale. Det giver en enorm tryghed at vide at man har testet sin software både i feature-udviklingen og i integrationen, så man tør lave nye ting uden at være

bange for at ødelægge noget eksisterende. Når man alligevel arbejder med et git-repository er det oplagt at sætte actions til at hjælpe med automatiseringen af test og deploy. Det er svært at forklare hvorfor det opleves som trygt, men det er virkelig rart at se sine test fungere når man har merget til integration og vide at hele koden fungerer sammen.

Det meste af det, jeg har opnået i dette projekt, har været nyt for mig, så jeg har lært en masse nye ting, og hvordan jeg sætter det sammen på en meningsfuld måde.

"A couple of weeks of programming can save a few hours of planning!"

kilder:

https://about.gitlab.com/blog/2019/11/06/gitlab-ci-cd-is-for-multi-cloud/

https://stackshare.io/feed

https://www.katalon.com/resources-center/blog/ci-cd-tools/

https://www.lambdatest.com/blog/27-best-ci-cd-tools/

https://stackoverflow.com/

https://www.youtube.com/watch?v=zujeb8VWncI

https://www.youtube.com/watch?v=t6RbanOhna4

https://www.enterprisedb.com/postgres-tutorials/how-use-postgresql-django

https://www.linode.com/docs/guides/how-to-install-use-postgresql-ubuntu-20-04/

https://www.tutorialspoint.com/uml/uml_basic_notations.htm

https://www.omg.org/spec/UML/2.5.1/PDFhttps://pynative.com/python-uuid-module-to-generate-universally-unique-identifiers/

https://nvie.com/posts/a-successful-git-branching-model/

https://en.wikipedia.org/wiki/Deployment_diagram

https://www.youtube.com/watch?v=X3F3El_yvFg

https://www.youtube.com/watch?v=WTofttoD2xg

og mange flere steder på internettet.