

Metadata Abonnement System

Januar 2024.

Datamatiker, 5. Semester, afsluttende projekt.

Udviklet for:

Databank kontoret - Styrelsen for Dataforsyning og Infrastruktur.

Udviklet af:

Nikolaj Skibsted.

Hold: Dat21C/Dat21a-ny - Københavns Erhvervsakademi.

“Design and programming are human activities; forget that and all is lost.”

— Bjarne Stroustrup

Page 1 of 32

Indholdsfortegnelse

1	Indledning.....	3
1.1	Problemformulering.....	3
1.2	Krav til applikationen.....	3
2	Udviklingsforløb.....	4
2.1	Agile værdier.....	4
2.2	SCRUM.....	5
2.3	Meta Data Abonnement Systemets udviklingsforløb.....	6
3	Softwarekonstruktion.....	8
3.1	Django.....	8
3.2	PostgreSQL.....	9
3.2.1	Databasemodellering med migrations og ORM.....	10
3.3	The Fullstack Process.....	11
3.3.1	Brugerinteraktion.....	11
3.3.2	URL Routing.....	12
3.3.3	View Processing.....	13
3.3.4	Formdata håndtering og Validering.....	14
3.3.5	Databaseinteraktion.....	15
3.3.6	Respons og navigering til Personal_page.....	16
3.3.7	Kommentar.....	17
3.4	Testing.....	18
4	Infrastruktur med docker.....	20
4.1	dockerfile.....	21
4.2	entrypoint.sh.....	22
4.3	docker compose.....	23
4.3.1	compose-init.yaml.....	24
4.3.2	compose.nikolaj.yaml.....	25
4.4	Deployment.....	28
5	Konklusion.....	29
6	Litteraturliste.....	30

1 Indledning.

Velkommen til rapporten om mit afsluttende projekt på datamatiker uddannelsen. Projektet er udført i samarbejde med Databank kontoret hos SDFI – Styrelsen for Dataforsyning og Infrastruktur (SDFI), hvor jeg også har været i praktik forløb i 10 uger forud for udarbejdelsen af hovedprojektet. SDFI er en offentlig styrelse under Klima, Energi- og forsyningsministeriet, der indsamler, vedligeholder og distribuerer de vigtigste geografiske data om Danmark. Arbejdet med dette projekt spænder over størsteparten af de fagområder, der har været fokus på i løbet af uddannelses forløbet. Denne rapport vil fokusere på de fagområder der har været mest væsentlige i udarbejdelsen af dette projekt.

1.1 Problemformulering.

Databanken administrerer, blandt andet, metadata for styrelsens datasæt. Ændringer i et datasæt kan have stor indflydelse på kontorerne i styrelsens arbejde. Derfor ønsker databanken at udvikle en web applikation til internt brug, der kan anvendes til at registrere brug af datasæt, og til at informere brugerene om ændringer i datasættene og deres metadata.

1.2 Krav til applikationen.

1. *Bruger komponent.*

- Ansatte i styrelsen kan registrere sig som bruger af et eller flere datasæt.
- Brugeren skal ved registrering indtaste en beskrivelse af deres arbejde med datasættet.
- Brugeren skal kunne håndtere egne registreringer på datasæt fra en personlig bruger side.
- Brugeren skal kunne abonnere på information om ændringer i datasæt.

2. *Admin komponent*

- En administrativ bruger af applikationen kan se brugernes registreringer.
- Den administrative bruger kan informere brugerene om ændringer i datasættene.
- Den administrative bruger kan slette en almindelig bruger og alle tilhørende registreringer.

Informeringen af brugerne kommer til at foregå manuelt, den administrative medarbejder sender emails ud til de registrerede brugere når der er ændringer i datasæt, der vurderes til at kunne have indflydelse på brugernes arbejde.

Tekniske krav:

Det er besluttet at applikationen skal bygges med python som generelt programmeringssprog, og Django som framework.

Persistente data gemmes i en PostgreSQL-database.

Komponenterne konfigureres til at køre i Docker-containerer på et Ubuntu-operativsystem.

2 Udviklingsforløb.

I Dette kapitel vil jeg beskrive de arbejdsmetoder og processer, der er anvendt under udviklingen af Meta Data Abonnement web applikationen. Selvom jeg gennemfører dette projekt som en enkeltmands gruppe og derfor ikke har et udviklingsteam at interagere med, har jeg alligevel valgt at implementere agile¹ arbejdsmetoder og teknikker ad hoc. Disse valg er baseret på min tidligere erfaring og viden erhvervet i løbet af uddannelsen. Jeg fandt især Scrum teknikker anvendelige, selv i en solo-team kontekst, for at holde projektet struktureret og progressivt.

2.1 Agile metode og værdier.

Agile metode handler grundlæggende om at være tilpasningsparat, med et konstant beredskab til at håndtere forandringer i projektets krav og nyopståede udfordringer. Denne tilgang prioriterer fleksibilitet og hurtig respons på ændringer, hvilket sikrer, at udviklingsprojektet forbliver relevant og effektivt, selv når omstændighederne skifter. Udviklingen af projektet i forhold til de 4 agile grundlæggende værdier:

Individer og interaktioner over processer og værktøjer.

Selvom jeg har arbejdet alene på dette projekt og derfor ikke haft et traditionelt team at interagere med, har kommunikation og individuel problemløsning været centralt. Jeg har haft adgang til kompetent personale for sparring, hvilket har været uvurderligt for projektets progression. At arbejde selvstændigt medfører både frihed og ansvar – friheden til at træffe beslutninger selvstændigt og ansvaret for at navigere i udfordringer uden et teams direkte support. Denne balance har fremhævet vigtigheden af fleksibilitet og selvledelse i agile metoder.

Fungerende software over omfattende dokumentation.

Dette agile princip har haft afgørende betydning for mit projektarbejde. I min rolle som eneste udvikler har behovet for intern dokumentation været lille, idet fokus har været på at skabe og forbedre fungerende software. Denne tilgang har muliggjort en løbende præsentation af produktet for arbejdsgiveren, hvilket har åbnet op for værdifuld feedback og mulighed for løbende forbedringer. Dette har vist hvor effektivt det kan være, at prioritere produktets funktionalitet over detaljeret dokumentation.

Kundesamarbejde over kontrakt.

I dette projekt har 'kunden', min arbejdsgiver, spillet en central rolle. Selv uden en formel kontrakt har samarbejdet været essentielt, styret af de deadlines, der er sat af uddannelsesinstitutionen. Dette samarbejde har været grundlæggende for projektets udvikling, idet løbende dialog og feedback har været med til at forme og tilpasse projektets retning. Denne tilgang understreger vigtigheden af fleksibelt og responsivt samarbejde i agile projekter, frem for stive kontraktbaserede forhold.

¹ Pressman & Maxim, Chapter 3, p. 43.

Reaktion på ændringer over at følge en plan.

Gennem udviklingen og implementeringen af projektet er der opstået flere ændringer i krav og behovet for tilføjelse af nye features. Disse ændringer har udfordret mig til at tilpasse projektets struktur og arbejdsplan dynamisk, uden at kompromittere det overordnede mål. Min evne til at omfavne og håndtere disse forandringer bekræfter agil metodes styrker i at reagere på ændringer effektivt, fremfor stift at holde sig til en forudbestemt plan.

2.2 SCRUM.

Scrum er en meget populær udviklingsmetode udtænkt af Jeff Sutherland og hans udviklingsteam i begyndelsen af 1990'erne.²

Overordnet betragtes Scrum som en agile metodologi, fordi den overholder de agile værdier og principper. Da Scrum som udgangspunkt er et framework for arbejde i teams, har jeg ikke benyttet mig af det man kan kalde et komplet scrum arbejds forløb. Jeg har udvalgt de elementer jeg har fundet nyttigt for mit arbejde som enkeltmandsudvikler.

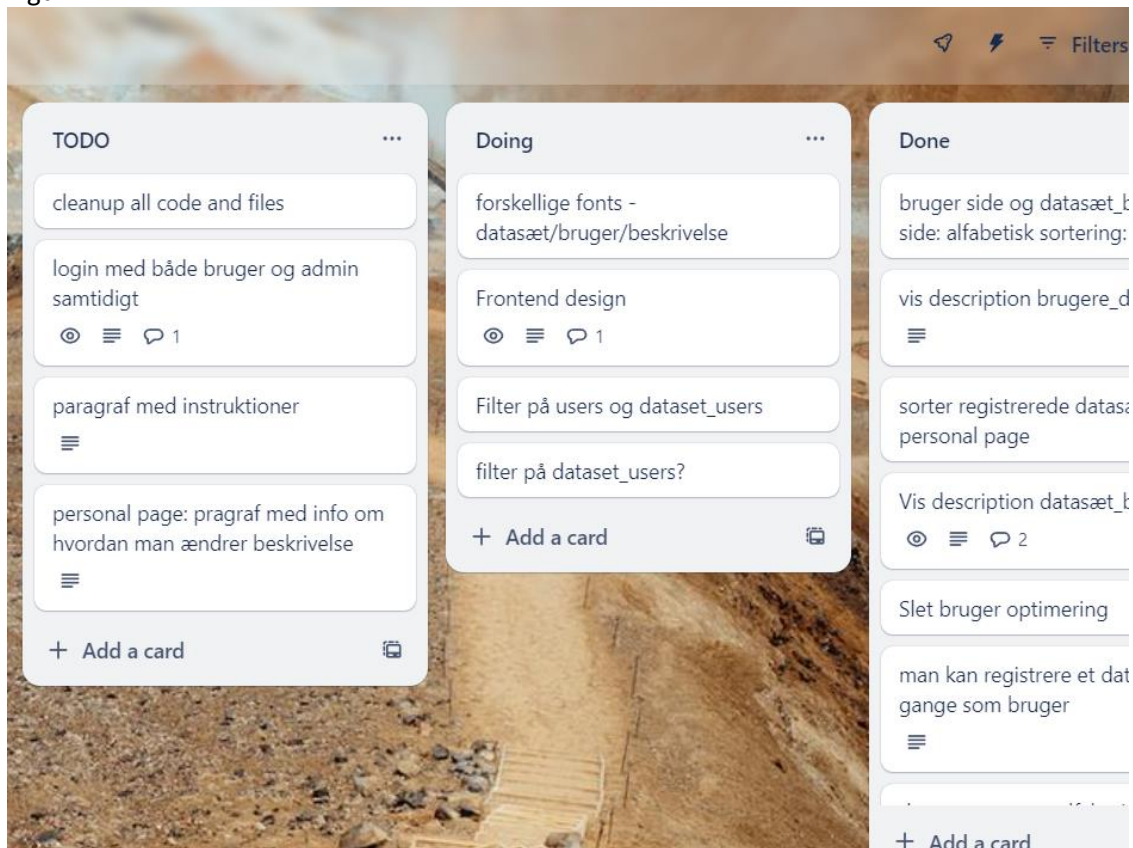
Anvendte SCRUM elementer i dette projekt:

- *Userstories*
User Stories er en måde at dokumentere kundens/brugernes krav og forventninger i et sprog som kunden/brugerne selv kan forstå.³
Jeg har anvendt userstories for at få en dybere forståelse for kravene til projektet.
- *Product Backlog*
En product backlog er en prioriteret liste over krav og features til produktet. Listen opdateres løbende som udviklingen skrider frem. Jeg har løbende ført en product backlog under udviklingen af projektet, backloggen bliver til rygraden for næste element, scrumboardet.
- *Scrumboard*
Et scrumboard er et organisatorisk værktøj der anvendes til at gøre opgaverne i et udviklingsprojekt overskuelige. Jeg har anvendt Trello som scrumboard tilpasset mine behov, i dette forløb (se figur 1). Ved at føre kontinuerligt opdaterede prioriterede lister over krav og tasks er dette et yderst nyttigt redskab til at strukturere og overskue arbejdet.
- *Daily Scrum meeting*
Jeg har ikke holdt daglige standup møder med mig selv, men jeg har sammen med en anden KEA studerende, der også laver sit projekt alene, holdt et kort møde mange morgener i forløbet, hvor vi på skift har berettet om hvad vi har opnået, hvad vores plan for dagen er, og eventuelt hvilke udfordringer der eventuelt er opstået. Dette har vi begge haft stor glæde af. Bare det at få sat ord på ens arbejde og der er en anden person der forholder sig til det har for mig været meget givende.

² Pressman & Maxim, Chapter 3, p. 42.

³ <https://www.teknologisk.dk/kurser/scrum-er-hvad-du-goer-det-til/41615>

Figur 1



Figur 1 er et udsnit af mit scrum board under udviklingen af Meta Data Abonnement Systemet.

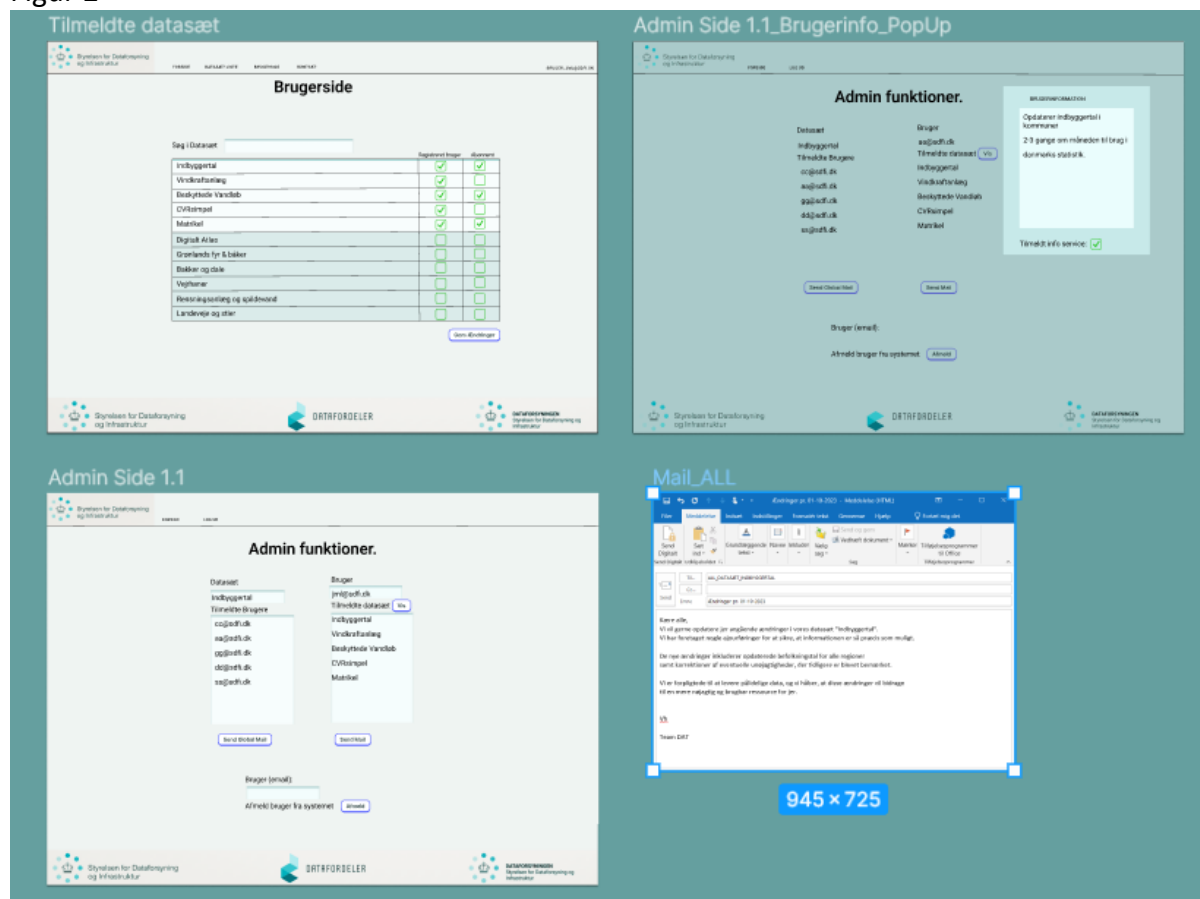
2.3 Meta Data Abonnement Systemets udviklingsforløb.

Den overordnede vision for og ønskerne til denne applikation blev præsenteret mundtligt for mig i et møde med min nærmeste leder i databanken, Peter, til at begynde med. Ud fra de informationer, blev jeg bedt om selv at definere kravene til projektet yderligere, og udarbejde en prototype af applikationen i design værktøjet Figma, med den hensigt at kunne præsentere denne for de aktuelle interessenter i kontoret og derefter modtage feedback til brug for yderligere specifikation af krav og ønsker til applikationen. Hermed er den overordnede proces model der skal anvendes til udviklingen af projektet også defineret: *Prototyping process Model*.⁴

Da projektet fra starten kun er baseret på ideér og ønsker og der ikke findes noget definitivt design, viser denne model sig at være et glimrende redskab til at få defineret og raffineret kravene til applikationen for hver version af prototypen der præsenteres. De to første prototype versioner er rene design modeller hvor der er lavet en idé baseret model af alle skærbilleder i design værktøjet Figma. I dette værktøj er det muligt at lave en interaktiv model der giver meget realistisk oplevelse af de ønskede funktioner. Ud fra denne oplevelse kan de relevante interresenter levere feedback af høj kvalitet.

⁴ Pressman & Maxim, Chapter 2, p. 26.

Figur 2



Figur 2 er et screenshot af en del af Figma prototypen der blev produceret og anvendt til at udvikle det endelige produkt. Jeg har prøvet at visualisere ønskerne til applikationen velvidende om at det endelige produkt kommer til at se anderledes ud. Men ideér og ønsker til funktionalitet er med så det kan præsenteres for at skabe dialog.

Efter præsentationen og evalueringen af design prototyperne kunne jeg bryde mine userstories ned i tasks og påbegynde implementeringen af dem.

De to næste prototyper var fungerende versioner af applikationen. Det var op til mig at afgøre hvornår der var nok fungerende funktionalitet til en ny præsentation og behov for ydeligere specificering af features samt svar på spørgsmål der opstod løbende med implementeringen.

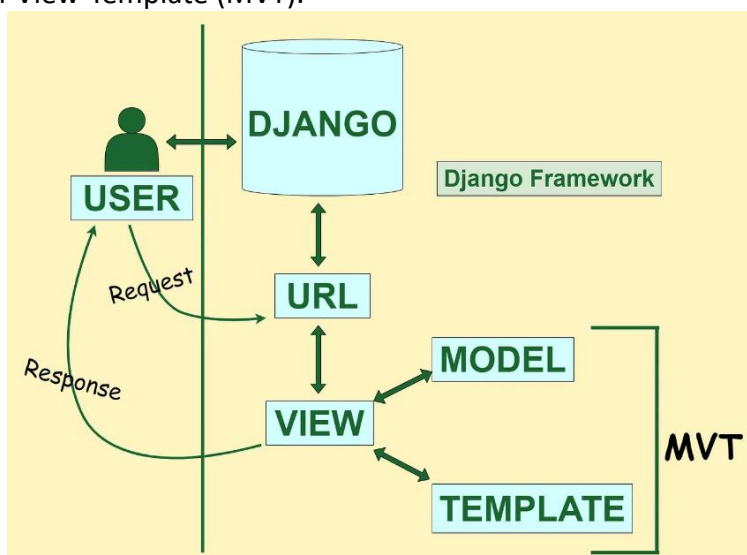
Det blev til to yderligere store møder med præsentation, feedback og afklaring af spørgsmål. Sideløbende har jeg holdt talrige individuelle møder med relevante personer i flere af kontorerne i SDFI, blot til sparring eller om mere specifikke tekniske spørgsmål og hvad der ellers opstod af udfordringer. Der vil ultimativt blive et afsluttende møde med Databank kontoret med præsentation og overlevering af applikationen og udlånte materialer, efter afsluttet eksamen.

3 Software konstruktion.

I dette kapitel vil jeg fokusere på konstruktionen af applikationen, herunder hvilke software aspekter der er anvendt i de forskellige lag. Jeg vil fremstille en præsentation med eksempler der tager udgangspunkt i en brugers interaktion med systemet og rejsen hele vejen gennem stacken. Denne detaljerede gennemgang af en brugerinteraktion giver et indblik i, hvordan Django's templates, URL-routing, views, forms og modeller arbejder sammen for at skabe en brugervenlig og funktionel webapplikation.

3.1 Django.

Valget af Django blev besluttet efter drøftelser med udviklerne internt i Databank kontoret, da dette framework indeholder mere end rigeligt af funktionalitet og integrationer til at kunne konstruere Meta Data Abonnement Systemet. Da projektet skal konstrueres med Django frameworket som skelet, bliver det anvendte kodesprog naturligvis Python. Django er et gratis og open source, Python-baseret web framework, der følger det arkitektoniske mønster Model-View-Template (MVT).⁵



Django MVT: Når brugeren sender et URL request, aktiveres det tilsvarende VIEW for den pågældende URL. VIEW'et interagerer med databasen via MODEL, hvis det er en del af funktionaliteten i det specifikke view. Viewet sørger for at sende et response tilbage, f.eks i form af et HTML-dokument, som er defineret i en TEMPLATE-fil.

Mens Python udgør rygraden i Django med hvilket server-side funktionalitet og databasemanipulation er defineret, bruges HTML, CSS og JavaScript til at udvikle webapplikationens brugergrænseflade ovenpå Djangos template system. I afsnittet længere fremme i rapporten, hvor jeg gennemgår en brugers interaktion med systemet, vil jeg nærmere beskrive flere af de dele af Django frameworket der er anvendt i projektet.

⁵ <https://www.educative.io/answers/what-is-mvt-structure-in-django>

3.2 PostgreSQL.

PostgreSQL er et open source objekt-relational database management system, der understøttes af 30 års udvikling, hvilket gør den til en af de mest etablerede tilgængelige relationsdatabaser ⁶. Valget af PostgreSQL som databasen der er anvendt i udviklingen af dette projekt er et valg der er taget af kontoret, IT-udvikling, i SDFI. Det er den database der anvendes til udviklings projekter i styrelsen. Senere, når projektet jeg beskriver, skal ud i produktion, skal det integreres med en Oracle Enterprise database hvorfra der skal hentes live opdateringer om datasæt.

Her er en beskrivelse af hvordan PostgreSQL til at begynde med, er integreret med Django i dette projekt: I **settings.py**, som er hoved konfigurations filen i et Django projekt, defineres forbindelsen til PostgreSQL-databasen således:

```
80 DATABASES = {
81     'default': {
82         'ENGINE': 'django.db.backends.postgresql',
83         'NAME': config('DATABASE_NAME'),
84         'USER': config('DATABASE_USER'),
85         'PASSWORD': config('DATABASE_PASSWORD'),
86         'HOST': config('DATABASE_HOST'),
87         'PORT': config('DATABASE_PORT', 5432),
88     }
89 }
```

Denne konfiguration viser, hvordan Django's indbyggede support til PostgreSQL gør det muligt at forbinde til databasen ved at angive parametre som databasenavn, brugernavn, adgangskode, host og port. Brugen af **config** er p.g.af at oplysningerne hentes fra en .env fil. Senere er databasen og applikationen sat op til at køre i docker containere og da bruger man direkte referencer til miljøvariabler med \$-notationen, hvilket er standard for Docker-compose og forskelligt fra, hvordan **config**-funktionen anvendes i Django's **settings.py**.

⁶ <https://www.postgresql.org/about/>

3.2.1 Databasemodellering med migrations og ORM

I Django er **models.py** en konventionel fil som man definerer databasemodeller i. Disse modeller er Django's abstrakte repræsentationer af, hvordan databasetabellerne vil se ud. Når man har defineret modellerne, genererer man migrationsfiler med kommandoen **python manage.py makemigrations**. Disse migrationsfiler er scripts, som Django opretter for at ændre databasestrukturen (f.eks., oprette en ny tabel, ændre en eksisterende tabel osv.). For at udføre ændringerne i databasen, skal man køre kommandoen **python manage.py migrate**

Her er et eksempel fra **models.py** i Meta Data Abonnement Systemet:
Dataset model:

```
3 class Dataset(models.Model):
4     OBJECTID = models.IntegerField(unique=True)
5     ID_LOKALID = models.CharField(max_length=255, unique=True)
6     TITEL = models.CharField(max_length=255)
7
8     def __str__(self):
9         return self.TITEL
```

Denne model definerer en **Dataset** tabel med felterne **OBJECTID**, **ID_LOKALID** og **TITEL**.

Efter at have defineret modellerne og kørt migrations, benytter Django's ORM (Object Relational Mapping) sig af disse modeller til at interagere med databasen på et højere og mere abstrakt niveau end direkte SQL. ORM'en tillader at man kan manipulere databasen ved at bruge objekter i stedet for at skrive ren SQL-kode. Dette forenkler udviklingsprocessen betydeligt, da det reducerer behovet for at kende specifikke database-queries og SQL-syntaks.⁷

Django's ORM kan også beskytte mod SQL-injection angreb gennem brugen af query parameterisering, hvilket betyder, at SQL-koden for en database forespørgsel defineres separat fra dens parametre. Dette sikrer, at parametre der er konstrueret af brugerinput, som kan være usikre, bliver korrekt escaped af den underliggende database-driver. Så når et brugerinput integreres i en databaseforespørgsel, sikrer Django, at det behandles på en sikker måde.⁸

Object Relational Mapping er ikke unikt for Django. Af andre eksempler kan nævnes Hibernate til java og Microsofts Entity Framework til brug med .Net.

⁷ <https://www.fullstackpython.com/object-relational-mappers-orms.html>

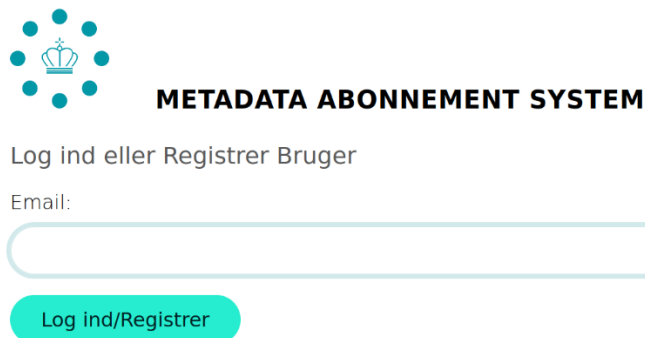
⁸ <https://docs.djangoproject.com/en/4.2/topics/security/#sql-injection-protection>

3.3 The Fullstack Process

Introduktion: I dette kapitel vil jeg gennemgå en brugerinteraktion i Meta Data Abonnement Systemet, fra det øjeblik brugeren interagerer med systemet på forsiden 'homepage.html' og efterfølgende navigerer til 'personal_page.html'. Jeg vil beskrive den proces der iværksættes og følge den gennem hele stacken, fra frontend til backend, og illustrere, hvordan de komponenter i Django-frameworket som er anvendt i denne applikation, samarbejder for at levere en simpel og effektiv brugeroplevelse.

3.3.1 Brugerinteraktion

Brugeren navigerer til forsiden, **homepage.html**, og præsenteres for registreringsformen som er et centralt element i brugeroplevelsen.



METADATA ABONNEMENT SYSTEM

Log ind eller Registrer Bruger

Email:

Log ind/Registrer

Formen er defineret i **forms.py** og indlejres i templateen ved hjælp af Django's Template Language (DTL).⁹ Django's templatesystem er et effektivt værktøj, der tillader udviklere at skabe dynamiske HTML-sider. Templates adskiller præsentationslaget fra backend-logikken, hvilket gør det nemmere at vedligeholde og opdatere applikationens brugergrænseflade.

```
5 <h3>Log ind eller Registrer Bruger</h3>
6 <form method="post">
7     {% csrf_token %}
8     {{ user_form.as_p }}
9     <button type="submit" name="user_login">Log ind/Registrer</button>
10 </form>
```

Her i homepage templateen importeres formen med **{% csrf_token %}** og **{{ user_form.as_p }}**.

as_p-metoden konverterer formfelterne til HTML-paragraffer, hvilket giver en struktureret og letlæselig layout.

{% csrf_token %} er en vigtig Django-template tag, der beskytter mod Cross-Site Request Forgery-angreb ved at inkludere et token i form input dataen.¹⁰

⁹ <https://docs.djangoproject.com/en/5.0/ref/templates/language/>

¹⁰ <https://docs.djangoproject.com/en/5.0/ref/csrf/>

3.3.2 URL Routing

URL-routing er en vigtig komponent i enhver moderne webapplikation. Routingen definerer hvordan en applikation reagerer på requests fra en client. I Django sker dette gennem URL-dispatcher ¹¹, der tolker URL'er og dirigerer dem til de view-funktioner der er mappet til URL'en.

URL-dispatcheren i Django fungerer som en slags vejviser, der læser de URL-patterns, som er defineret i **urls.py**-filerne, og matcher disse patterns med indkommende web-requests. Når en bruger interagerer med applikationen – for eksempel ved at trykke på en 'submit'-knap på en formular – genereres et HTTP-request, som sendes til serveren. URL-dispatcheren analyserer dette request og afgør, hvilket view der skal aktiveres, baseret på det aktuelle URL-pattern.

Eksempel fra Meta Data Abonnement System `urls.py`:

```
5 | path('homepage/', views.home_page_view, name='home_page_view'),
```

Denne linje definerer et URL-pattern, tilknytter det til en specifik view-funktion, som håndterer anmodninger til denne URL og navngiver det til brug for reference andre steder i applikationen.

I tilfældet med Meta Data Abonnement Systemet, når en bruger besøger 'homepage/' og indsender formdata, sker følgende proces:

- Request Modtaget: Serveren modtager et request til URL'en 'homepage/'.
- Sammenligner URL Pattern: URL-dispatcheren kigger igennem urlpatterns i urls.py for at finde et matchende pattern.
- Forbinder til View: Når et match er fundet, forbindes det indkommende request til home_page_view i views.py.
- Udførelse af View: home_page_view-funktionen udføres, hvilket resulterer i behandling af brugerdata eller generering af et respons, såsom rendering af en ny side eller et redirect.

Effektiv URL-routing er afgørende for at sikre, at en webapplikation er navigerbar og intuitiv for brugeren. Det giver mulighed for at opbygge komplekse, men strukturerede applikationer, hvor hver del af applikationen er let tilgængelig gennem en specifik URL. I Django gør dette system det nemt for udviklere at udvide og vedligeholde applikationer, da tilføjelse af nye funktioner ofte simpelthen indebærer tilføjelse af nye URL-patterns og views.

¹¹ <https://docs.djangoproject.com/en/5.0/topics/http/urls/>

3.3.3 View Processing

Views er en af de vitale dele i Django's MVT struktur. Et view er en Python-funktion, der tager et web request som parameter og returnerer et web response. Dette response kan f.eks være: HTML indholdet af en web side, et redirect, en exception error, et XML dokument, kort sagt, alt hvad en web browser kan vise. Hvert view skal tilknyttes et tilsvarende URL-pattern. I vores tilfælde behandler `home_page_view`-funktionen i `views.py` brugerens request og registreringsdata. Denne funktion håndterer både indlæsning af formularen og behandling af formularindsendelser.

```
8 def home_page_view(request):
9     user_form = UserLoginForm(request.POST or None)
10
11     if request.method == "POST":
12         if 'user_login' in request.POST and user_form.is_valid():
13             email = user_form.cleaned_data.get('email')
14             user = CustomUser.objects.filter(EMAIL=email).first()
15             if not user:
16                 user = CustomUser.objects.create(EMAIL=email)
17             request.session['user_id'] = user.USERID
18             return redirect('personal_page_view')
```

Når en bruger besøger forsiden, initialiseres `UserLoginForm`. Når brugeren sender et input med registreringsformen, valideres dataene, og en ny bruger oprettes i CustomUser tabellen i databasen, hvis den ikke allerede eksisterer. Uanset om brugeren eksisterer eller først bliver oprettet her, oprettes der en session, brugerens ID gemmes i denne, og brugeren omdirigeres til deres personlige side (**personal_page.html**). Denne session gør det muligt for systemet at holde styr på brugerens aktiviteter og sikre, at brugeren forbliver autentificeret, når de navigerer i applikationen.

Normalt ville man adskille bruger registrering og login men da det er besluttet i Databank kontoret at login kun skal foregå med email og ikke password, er login og registrering slået sammen. Dette begrundes med at det kun er interne brugere der allerede er autentificeret med login fra statens IT der har tilgang til systemet.

3.3.4 Formdata håndtering og validering

Django Forms er et kraftfuldt værktøj, der anvendes til at håndtere og validere brugerinput i webforms. Det indeholder gode metoder til at definere form felter med, udføre validering af data og præsentere forms på en ensartet måde. I dette tilfælde bruger vi **UserLoginForm** til at indsamle og validere brugerens email.

Eksempel fra `'forms.py'`:

UserLoginForm-klassen definerer formularfeltet for brugerens email og indeholder en tilpasset valideringsmetode:

```
7  class UserLoginForm(forms.Form):
8      email = forms.EmailField(label="Email")
9
10 def clean_email(self):
11     email = self.cleaned_data.get('email')
12     if not email.endswith('@sd.fi.dk'):
13         raise forms.ValidationError("Email must end with '@sd.fi.dk'")
14     return email
```

UserLoginForm bruges til at opsamle brugerens email på en sikker og valideret måde. I **clean_email**-metoden udføres en tilpasset validering, som sikrer, at den indtastede email opfylder specifikke krav (i dette tilfælde, at den slutter med '@sd.fi.dk'). Hvis emailen ikke opfylder dette krav, udløses en valideringsfejl. Denne tilgang sikrer, at kun gyldige emailadresser accepteres, hvilket er afgørende for at opretholde dataintegriteten og sikkerheden i applikationen. Denne form for validering understreger vigtigheden af at have et pålideligt valideringssystem på plads for at beskytte webapplikationer mod uautoriserede eller skadelige input.

3.3.5 Databaseinteraktion

Efter at brugerens input er blevet valideret, er næste skridt at gemme disse data i databasen. Dette håndteres af **CustomUser**-modellen i **models.py**, som er en del af Django's ORM-system :

```
11 class CustomUser(models.Model):
12     USERID = models.AutoField(primary_key=True)
13     EMAIL = models.EmailField(unique=True)
14     datasets = models.ManyToManyField(Dataset, related_name='users')
15
16     def __str__(self):
17         return self.EMAIL
```

I denne model:

- **USERID** er defineret som en automatisk genereret primærnøgle, som unikt identificerer hver bruger i systemet.
- **EMAIL** er et felt, der gemmer brugerens emailadresse. Attributten **unique=True** sikrer, at hver email i systemet er unik.
- **datasets** er et ManyToMany-felt, der etablerer en relation mellem **CustomUser** og **Dataset**-modellen, hvilket gør det muligt for brugere at være tilknyttet flere datasæt.

CustomUser-modellen fungerer som grundlaget for at gemme brugeroplysninger i databasen.

Da jeg allerede tidligere i rapporten, har beskrevet hvordan modellerne fungerer med Django's databasemigrations- og ORM-system, henvises til afsnit "**3.2.1 Databasemodellering med migrations og ORM**". Dette afsnit giver en mere detaljeret forklaring på modellering, migrationsprocessen, og hvordan Django's ORM håndterer sikker og effektiv databehandling.

3.3.6 Response og navigering til personal_page

Efter at en bruger er blevet succesfuldt registreret eller logget ind, er det næste trin at omdirigere brugeren til brugersiden - `personal_page.html`, hvorfra de kan interagere med applikationen. Denne omdirigering er et vigtigt aspekt af brugeroplevelsen, da den bekræfter, at registreringen eller login-processen er fuldført, og giver brugeren adgang til deres personlige område i applikationen.

Bruger: peter@sdfi.dk

Log ud

Registrer brug af Datasæt

Dataset:

Beskriv anvendelse af Datasæt:

Registrer

Registrerede Datasæt

- Anonymiseret elforbrugsdata - it, sed diam nonummy nibh euismod tincidunt ut laoreet dolor quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat, vel illum dolore eu feugiat nulla facilisis at ve

Slet

Som beskrevet i afsnit 3.3.3 - View Processing, håndterer **home_page_view** funktionen, brugerens request, herunder validering og oprettelse af en session. Efter disse trin anvender funktionen Django's **redirect**-metode til at føre brugeren til en ny URL, specifikt til **personal_page_view**.

Ved at omdirigere til **personal_page_view**, præsenteres brugeren for en personlig side, som reflekterer deres unikke brugerstatus og -data. Denne side fungerer som en central hub for brugerens interaktion med applikationen, hvor de kan se og interagere med relevante data og funktioner.

Denne omdirigering er ikke kun en teknisk nødvendighed, men også en vigtig del af brugeroplevelsen. Her sker en overgang fra en simpel registreringsproces til en personliggjort brugeroplevelse. Den understøtter også Databank kontorets beslutning om at forenkle login-processen ved kun at anvende email, hvilket gør brugerflowet mere strømlinet og effektivt.

3.3.7 Kommentar

Samlet set illustrerer kapitel 3, hvordan Django's MVT-arkitektur og dens sikkerhedsfunktioner ikke blot forenkler udviklingsprocessen, men også skaber en sikker og brugervenlig platform. Ved at dele mekanismerne op, som det bliver gjort her, følges også software design princippet, *seperation of concerns* ¹², hvor man lægger vægt på at opdele et program i moduler og sektioner, der hver behandler specifikke aspekter af funktionalitet. Dette giver et bedre overblik og gør koden lettere at læse og vedligeholde. Og ved samtidig at integrere robuste sikkerhedsforanstaltninger, muliggør Django udviklingen af komplekse, men navigerbare webapplikationer, der kan tilpasses og opfylde brugernes behov effektivt.

¹² <https://deviq.com/principles/separation-of-concerns>

3.4 Testing

Testning er en vigtig del af softwareudvikling. Det er med til at sikre, at applikationen fungerer som forventet og fejlfrit i forskellige scenarier. I webudvikling med Django hjælper tests os med at validere, at vores views, modeller og formularer opfører sig korrekt. Ved at udføre tests, kan vi proaktivt identificere og rette fejl, øge kodekvaliteten og sikre en stabil brugeroplevelse.

Django leveres med et indbygget test framework ¹³, der gør det muligt for udviklere at skrive og køre tests for deres applikationer. Dette framework er baseret på Python's standard unittest bibliotek ¹⁴, men tilbyder også en række ekstra funktioner specifikt designet til at teste Django-applikationer.

Blandt disse er muligheden for at teste Django-modeller, views, forms og templates samt integrationen med Django's database og URL-routing system.

I **tests.py**-filen har vi implementeret flere tests, som tjekker forskellige aspekter af applikationen. Lad os se nærmere på nogle af disse tests:

CustomUserModelTest:

```
7  class CustomUserModelTest(TestCase):
8
9  def test_email_label(self):
10     user = CustomUser.objects.create(EMAIL="test@example.com")
11     field_label = user._meta.get_field('EMAIL').verbose_name
12     self.assertEqual(field_label, 'EMAIL')
```

Denne testklasse tester funktionaliteten af **CustomUser**-modellen. Test-funktionen **test_email_label**, validerer, at email-feltet i **CustomUser**-modellen er korrekt defineret.

HomePageViewTest:

Denne testklasse fokuserer på **home_page_view**-funktionen. Den indeholder tests såsom **test_home_page_view_status_code** :

```
20 def test_home_page_view_status_code(self):
21     response = self.client.get(reverse('home_page_view'))
22     self.assertEqual(response.status_code, 200)
```

Her testes for at hjemmesiden returnerer en HTTP 200 statuskode, hvilket indikerer, at siden indlæses korrekt.

¹³ <https://docs.djangoproject.com/en/5.0/topics/testing/>

¹⁴ https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Testing#what_does_django_provide_for_testing

UserLoginFormTest - klassen: Her tester vi **UserLoginForm**.

```
36     def test_valid_email(self):
37         form = UserLoginForm(data={'email': 'user@sdfi.dk'})
38         self.assertTrue(form.is_valid())
39
40     def test_invalid_email(self):
41         form = UserLoginForm(data={'email': 'user@example.com'})
42         self.assertFalse(form.is_valid())
43         self.assertIn('email', form.errors)
44         self.assertEqual(form.errors['email'], ['Email must end with \'@sdfi.dk\'])
```

Testene **test_valid_email** og **test_invalid_email** sikrer, at formvalideringen fungerer som forventet – accepterer gyldige emailadresser og afviser ugyldige.

For at køre disse tests, bruger man Django's indbyggede testrunner som er et af Django's standard tools. Django's testrunner er ansvarlig for at finde, køre og rapportere om tests. Når kommandoen **python manage.py test** køres, finder testrunneren alle filer i projektet, der starter med **test**, og udfører de tests, de indeholder. En af de vigtige funktioner ved Django's testrunner er, at den opretter en separat testdatabase for hver test session. Dette sikrer, at tests ikke forstyrrer hinanden og ikke ændrer på data i din udviklings- eller produktionsdatabase.

De tests, der er beskrevet her, er eksempler på unit testing ¹⁵. Unit testing fokuserer på at teste individuelle dele af softwaren - som f.eks. funktioner, metoder eller klasser - isoleret fra resten af systemet. Dette er særligt nyttigt for at sikre, at hver enkelt komponent fungerer korrekt uafhængigt af de andre dele af applikationen. I vores tilfælde tester vi specifikke dele af Django-applikationen, såsom modeller, views og formularer, for at sikre, at de hver især opfører sig som forventet. Man vil ofte kombinere unit testing med andre typer test, som f.eks. integrationstests eller funktionelle tests for at opnå en mere grundig validering af softwarens kvalitet og funktionalitet.

Testning er ikke kun et redskab til at opdage fejl; det er også en integreret del af en bæredygtig udviklingsproces. Ved at skrive og vedligeholde tests, kan udviklere sikre, at software opfører sig som forventet, selv når der foretages ændringer eller tilføjelser til koden. Dette giver en solid base for at udvikle pålidelige, robuste og brugervenlige applikationer.

¹⁵ <https://aws.amazon.com/what-is/unit-testing/>

4 Infrastruktur med Docker

Dette kapitel handler om brugen af Docker som en central teknologi i driften af Meta Data Abonnement Systemet. Docker spiller en vigtig rolle i projektets infrastruktur fordi det skaber et stabilt og isoleret miljø til vores webapplikation. Docker-teknologien er en populær og revolutionerende tilgang til at deploye og køre applikationer ved at bruge containere.

Containerisering med Docker indebærer, at hver del af systemet - fra de enkelte applikationskomponenter til de nødvendige dependencies - pakkes i selvstændige enheder, kendt som containere. Hver container fungerer som et isoleret miljø, der indeholder alt, hvad der er nødvendigt for at køre et bestemt stykke software, herunder applikationskoden, et runtime-miljø, system tools og libraries.¹⁶ Denne metode fremmer både portabilitet og skalerbarhed og sikrer, at applikationen kan deployes og køre effektivt uanset underliggende infrastruktur, så længe det er kompatibelt med docker.

Denne containerisering sikrer, at hver del af systemet kan køre uafhængigt af de andre, hvilket reducerer risikoen for konflikter mellem forskellige applikationskomponenter. Det betyder, at udviklere og systemadministratorer kan være sikre på, at softwaren kører ensartet og forudsigeligt, uanset hvor den deployes, hvad enten det er på en lokal udviklermaskine, i et testmiljø eller i en produktionsindstilling.

Jeg vil i kapitlet se nærmere på de specifikke Docker-filer, der udgør rygraden i vores Docker-setup: Dockerfile, entrypoint.sh, og docker-compose filerne. Gennem disse filer vil jeg illustrere, hvordan jeg har bygget et robust system, der understøtter en effektiv udvikling og deployment-proces. Til sidst vil jeg gå over hvordan applikationen er deployed på styrelsens servere ved hjælp af Docker Swarm og Jenkins.

¹⁶ <https://www.docker.com/resources/what-container/>

4.1 Dockerfile

I dette afsnit vil jeg gå i detaljer med Dockerfile, som er kernen i vores Django container-byggeproces for Meta Data Abonnement Systemet. Dockerfile'en er en instruktionssæt, der fortæller Docker, hvordan det skal opbygge Django containerens image.

```
1 FROM python:3.11.5-slim
2
3 ARG USERNAME=niksk
4 ARG UID=2022
5 ARG GID=4004
```

Vores Dockerfile starter med at definere et base-image, **python:3.11.5-slim**, som er en letvægtsversion af et officielt Python-image. Dette valg er gjort for at minimere containerens størrelse og sikre en hurtig opsætning og deployment.

Herefter opretter Dockerfile'en en ny bruger i containeren med specifikke UID (User ID) og GID (Group ID) for at øge sikkerheden.

```
11 WORKDIR /app
12
13 COPY mysite /app
14 COPY entrypoint.sh /
```

Applikationskoden og entrypoint-scriptet (**entrypoint.sh**) kopieres til containeren, og arbejdsområdet (**WORKDIR**) sættes til **/app**. Dette sikrer, at alle efterfølgende kommandoer kører i **/app**.

```
16 RUN apt-get update \
17     && apt-get upgrade -y \
18     && rm -rf /var/lib/apt/lists/* \
19     && pip install --no-cache-dir -r /app/requirements.txt \
20     && pip install --upgrade pip django
```

Dockerfile'en indeholder også kommandoer til at opdatere package lister, opgradere eksisterende packages, og fjerne unødvendige filer. Disse skridt sikrer, at containeren har de nyeste sikkerhedsopdateringer og at unødvendige filer fjernes for at reducere containerens størrelse. Endelig installeres de nødvendige Python-pakker defineret i **requirements.txt**, og både Python og Django opgraderes. Dette sikrer, at alle dependencies er på plads og opdateret, hvilket er nødvendigt for at applikationen kan køre problemfrit.

```
22 USER ${USERNAME}
23
24 ENTRYPOINT ["/entrypoint.sh"]
```

Med **USER**-kommandoen skiftes til den nyoprettede bruger, og **ENTRYPOINT** sættes til at køre **entrypoint.sh**-scriptet. Dette betyder, at når containeren starter, vil dette script blive kørt for at initialisere applikationen.

4.2 entrypt.sh

entrypt.sh spiller en central rolle i vores Docker container-setup for Meta Data Abonnement Systemet. Dette script initialiserer og kører Django-applikationen, når containeren starter.

```
1  #!/bin/bash
2
3  python manage.py makemigrations
4  python manage.py migrate
5  python manage.py runserver 0.0.0.0:3000
```

Linje 1 : **#!/bin/bash** er en shebang, der angiver, at scriptet skal køres i Bash-shell.

Linje 3 : **python manage.py makemigrations** tjekker for ændringer i Django-modellerne og genererer nye migrationsfiler, hvis der er blevet foretaget ændringer, der endnu ikke er reflekteret i databasen.

Linje 4 : **python manage.py migrate** udfører ændringerne i databasen. Dette skridt er afgørende for at sikre, at databasens skema matcher modellernes aktuelle tilstand. Hvis der ikke er nogen nye migrations, vil denne kommando ikke foretage sig noget, men den sikrer, at alle eksisterende migrations er udført.

Linje 5 : **python manage.py runserver 0.0.0.0:3000** starter Django's indbyggede udviklingsserver, hvilket gør applikationen tilgængelig på port **3000**. Ved at angive **0.0.0.0** som host-adresse, sikres det, at serveren er tilgængelig uden for Docker-containeren.

Med disse steps automatiseres processen for at synkronisere databasen og starte applikationen, hver gang containeren startes. Dette bidrager til at sikre en problemfri og konsistent opsætning, uanset hvor ofte containeren genstartes eller deployes på nyt. Dette script sikrer, at applikationen altid kører med den nyeste databasekonfiguration og er klar til at modtage requests, så snart containeren er i drift.

4.3 Docker compose

Docker Compose er et redskab man bruger til at definere og køre multi-container Docker-applikationer. I Docker Compose bruger man en YAML-fil til at konfigurere applikationens services, netværk og volumes. Dette gør det muligt at oprette og starte alle services fra konfigurationen, med en enkelt kommando.¹⁷

Docker Compose er særligt nyttig i udviklings-, test- og produktionsmiljøer, da det sikrer konsistens mellem miljøer og forenkler processen med at administrere og forbinde flere containers.

Ved at bruge Docker Compose kan udviklere og systemadministratorer definere, hvordan Docker-containers interagerer med hinanden, håndtere dependencies, og konfigurere komplekse applikationer. Dette omfatter også håndteringen af netværksindstillinger og hukommelse.

Jeg vil herefter gå i detaljer med to specifikke Docker Compose-filer, som er brugt i Meta Data Abonnement Systemet: **compose-init.yaml** og **compose.nikolaj.yaml**. Disse filer illustrerer, hvordan vi har brugt Docker Compose til at håndtere forskellige aspekter af vores applikations livscyklus.

¹⁷ <https://docs.docker.com/compose/>

4.3.1 compose-init.yaml

compose-init.yaml-filen spiller en central rolle i den indledende opsætning af vores Django-projekt i Docker-miljøet. Denne Docker Compose-konfigurationsfil er designet til at simplificere processen med at starte et nyt Django-projekt med Docker.

```
1  version: "3"
2  services:
3    django-init:
4      build: .
5      working_dir: /home/${UNAME}/${PROJECT_DIR}
6      command: python -m django startproject ${PROJECT_NAME} .
7      volumes:
8        - ./${PROJECT_DIR}:/home/${UNAME}/${PROJECT_DIR}
9      networks:
10       default:
11
12  networks:
13    default:
14      name: meta_subscribe_init
```

1. **Version:** Det angiver versionen af Docker Compose-filformatet, som vi bruger. Her er det version 3.
2. **Services:**
 - **Django-init:** Denne service er ansvarlig for at oprette selve Django-projektet.
 - **build:** . angiver, at Docker skal bygge et image ud fra Dockerfile'en i den aktuelle mappe.
 - **working_dir** indstiller arbejdsområdet i containeren, hvor kommandoerne vil blive udført.
 - **command:** **python -m django startproject \${PROJECT_NAME} .** er den kommando, der kører, når containeren starter. Den bruger Django's CLI til at oprette et nyt Django-projekt.
 - **volumes** sikrer, at projektets mappe på host systemet er synkroniseret med mappen i containeren, hvilket gør det muligt for udviklere at arbejde på filerne direkte fra deres lokale system.
3. **Networks:** Konfigurationen definerer et standard netværk for services, der tillader interne forbindelser mellem containers, hvis det er nødvendigt.

compose-init.yaml-filen bruges typisk kun én gang for at initialisere projektstrukturen ved projektets start. Den opretter grundlæggende Django-projektfiler og konfigurationer, som er essentielt for den indledende udviklingsfase. Efter denne indledende opsætning, er der normalt ikke brug for at køre **compose-init.yaml** igen i det daglige udviklings- eller produktionsmiljø. I stedet vil man bruge en anden Docker Compose-fil til at håndtere den løbende drift og udvikling af applikationen.

4.3.2 compose.nikolaj.yaml

compose.nikolaj.yaml-filen er central nerven for opsætningen af Docker-miljøet for Meta Data Abonnement Systemet. Denne Docker Compose-fil indeholder konfigurationer for både applikations- og databaselaget, hvilket sikrer en smidig integration og drift af begge komponenter.

```
1  services:
2
3      django:
4          build: .
5          restart: always
6          ports:
7              - "${PORT}:3000"
8          working_dir: /app
9          environment:
10             TZ: Europe/Copenhagen
11             DATABASE_NAME: ${DATABASE_NAME}
12             DATABASE_USER: ${DATABASE_USER}
13             DATABASE_PASSWORD: ${DATABASE_PASSWORD}
14             DATABASE_HOST: ${DATABASE_HOST}
15             DATABASE_PORT: ${DATABASE_PORT}
16             ADMIN_EMAIL: ${ADMIN_EMAIL}
17             ADMIN_PASSWORD: ${ADMIN_PASSWORD}
18          volumes:
19              - ./${PROJECT_DIR}:/app/
20          networks:
21              - default
22
23      db:
24          image: postgres
25          restart: always
26          environment:
27              POSTGRES_DB: ${DATABASE_NAME}
28              POSTGRES_USER: ${DATABASE_USER}
29              POSTGRES_PASSWORD: ${DATABASE_PASSWORD}
30          networks:
31              - default
32
33
34  networks:
35      default:
36          name: meta_subscribe
```

Lad os se på de vigtigste aspekter:

Services :

1 Django :

- **Build og Restart:** - **build:** . betyder, at Docker Compose vil bygge Django-containerens image ved hjælp af Dockerfile i den aktuelle mappe.
- **restart: always** sikrer, at servicen automatisk genstarter, hvis den af en eller anden grund stopper uventet.
- **ports:** Denne indstilling sørger for, at porten, som Django-applikationen lytter på inden i containeren, bliver eksponeret til den angivne port på værtssystemet.
- **working_dir** her indstilles arbejdsområdet i containeren.
- **environment** : her defineres environment variabler.

2 DB(postgreSQL) :

- **Image:** Servicen anvender det officielle PostgreSQL Docker-image.
- **Restart og environment:** Ligesom i Django-servicen genstarter denne service automatisk, hvis den stopper uventet. Environment variablerne konfigurerer databaseforbindelsen, herunder databasenavn, bruger og adgangskode.

Networks :

Default network: I compose filen er både Django- og PostgreSQL-services konfigureret til at være en del af det samme netværk, som er defineret under **networks**. Dette er afgørende for at sikre, at de to containers kan kommunikere med hinanden og det giver også flere lag af sikkerhed.

Inter-Container Kommunikation: Når containere placeres på det samme netværk, kan de finde og kommunikere med hinanden. For eksempel, Django-applikationen skal kunne connecte til PostgreSQL-databasen. Ved at være på samme netværk kan Django-applikationen referere til databasen ved hjælp af dens service-navn (i dette tilfælde **db**), som fungerer som et internt DNS-navn inden for Docker-netværket. ¹⁸

Sikkerhedsaspekter:

- **Isolering:** Ved at oprette forskellige netværk kan man isolere grupper af containers fra hinanden. Dette betyder, at kun de containers, der er nødvendige for en bestemt funktion eller et bestemt lag i din applikation, kan kommunikere med hinanden. Det reducerer risikoen for utilsigtet eksponering af services og minimerer overfladen for potentielle angreb. ¹⁹

¹⁸ <https://docs.docker.com/network/>

¹⁹ <https://spacelift.io/blog/docker-security>

- **Begrænset Adgang:** For eksempel kan man have en container, der kører en front-end service, på et adskilt netværk fra en database-container for at begrænse adgangen til databasen. Kun de services, der kræver direkte adgang til databasen (f.eks en backend komponent), vil være på samme netværk som databasen.
- **Netværkspolitisikker:** Avanceret netværkskonfiguration og -policies kan yderligere forbedre sikkerheden. Docker og Docker Compose tillader indstilling af netværks policies, som kan kontrollere og begrænse, hvordan containers inden for et bestemt netværk interagerer med hinanden og med eksterne netværk.²⁰

Ved strategisk brug af disse netværkskonfigurationer og practices i Docker Compose-filen, kan man skabe et balanceret miljø, hvor funktionalitet og sikkerhed går hånd i hånd.

Compose-filen er afgørende for at sikre en problemfri og effektiv drift af Meta Data Abonnement Systemet. Ved at definere og konfigurere både Django-applikationen og PostgreSQL-databasen i en samlet Docker Compose-konfiguration, opnår man en høj grad af kontrol. Denne konfiguration gør det muligt at styre, hvordan hver del af systemet interagerer med resten, og samtidig opretholde en klar struktur og opdeling af ansvar inden for arkitekturen.

²⁰ <https://docs.docker.com/compose/networking/>

4.4 Deployment

Indtil videre er den udvikling der er beskrevet i rapporten foregået på en lokal device men, da det hele tiden har været meningen at applikationen skal i production, engang når den er klar, har ITU kontoret i SDFI hosted applikationen og databasen på styrelsens servere med docker swarm. Dette er dog stadig i et udviklings miljø, da der er yderligere planer for flere lag der skal udvikles til applikationen, før den skal ud i production.

Docker Swarm er brugt til at hoste og administrere applikationen i produktion.

Docker Swarm vs. Docker Compose: Docker Compose bruges til at konfigurere og starte flere Docker-containere på den samme host – så man ikke behøver starte hver container separat. Docker Swarm er et container styrings værktøj, der giver mulighed for at køre og forbinde containere der kører på flere forskellige hosts, som tilsammen udgør en "swarm". Dette er nyttigt i større, distribuerede systemer, hvor applikationer kræver at køre på tværs af flere servere for øget tilgængelighed, belastningsfordeling og skalerbarhed.

For at gøre det muligt for mig at deploye til serverne selv, er der oprettet en CI/CD pipeline med Jenkins, der automatisere build- og deploy processerne.

Jenkins er en Java-baseret open source automatiseringsplatform, der muliggør Continuous integration and Delivery (CI/CD) og automatiserer de forskellige stadier af softwareudvikling såsom test, build og implementering.²¹

Hver gang der er tilføjet nye dele til projektets kodebase i det tilhørende version control repository, iværksættes, manuelt, et nyt build fra jenkins klienten. Denne handling iværksætter en automatiseret Build og Deployment proces:

- Jenkins henter den seneste version af koden, bygger automatisk den nyeste version af applikationen ved hjælp af *docker build* kommandoen. Dette indkapsler de nyeste ændringer i kodebasen sammen med nødvendige dependencies i et Docker-image.
- Dernæst udfører Jenkins en række tests med det formål at verificere funktionaliteten og integriteten af applikationen.
- Når build-processen er gennemført og testene er bestået, bliver det nye Docker-image uploadet til styrelsens eget Docker registry.
- Herefter deployer Jenkins automatisk koden til containeren på serveren via Docker Swarm. Dette skridt inkluderer opdatering af den kørende container med det nyeste Docker-image og dermed den seneste version af applikationen.

²¹ <https://www.spiceworks.com/tech/devops/articles/what-is-jenkins/>

5 Konklusion

Her ved slutningen af det afsluttende projekt forløb kan jeg konkludere, at det er lykkedes mig med at opfylde databank kontorets ønsker - Produktets funktionalitet stemmer godt overens med de oprindelige mål og krav. Jeg har konstrueret en velfungerende web applikation til internt brug, inden for de tekniske rammer der blev sat op for mig.

Her vil jeg blandt andet fremhæve at det har været en success oplevelse at bygge en fungerende applikation med Django frameworket, som jeg ikke havde nogen kendskab til i forvejen. Selvom det har været spændende og lærerigt at arbejde Django skal det siges at det kun er en mindre del af frameworket der er kommet i brug i dette projekt – derfor kan man argumentere for at man måske kunne have anvendt et mindre framework som f.eks Flask der har en noget mindre stejl indlæringskurve. Eller frameworks som Node.js eller Java Spring Boot, som jeg har erfaring med fra tidligere i uddannelsesforløbet, erfaring jeg kunne have bygget videre på. Men jeg synes alligevel at det har været værdifuld erfaring og god øvelse i at sætte sig ind i og arbejde med, nye teknologier. Jeg vil også sige, at jeg i høj grad har kunnet anvende principper fra arbejdet med de andre teknologier tidligere i uddannelsesforløbet, da mange af principperne går igen i fullstack udvikling med Django. Og så vil jeg tilføje at der har været, og fortsat er, masser af kød på den teknologi stack jeg har bygget op, og på projektet i det hele taget, i forhold til uddannelsens læringsmål og IT-udvikling generelt.

Software konstruktionen er det der har taget mest tid, men det har også været spændende og lærerigt at arbejde i det infrastruktur setup med docker teknologier, som styrelsen har stillet for mig.

Om udviklings- og arbejds processerne vil jeg retrospektivt konkludere at, selvom det er lettere at tage beslutninger når man arbejder alene, vil jeg højst sandsynligt fremover foretrække at arbejde i teams, da den menneskelige interaktion for mig er et vigtigt element i et domæne der er domineret af maskiner og teknologi.

Overordnet set har jeg fået masser af brugbar erfaring ud af udviklingsforløbet, som jeg kan tage med videre ud på arbejdsmarkedet. Jeg har fået indsigt i hvordan jeg selv arbejder og ideér til hvordan jeg kan blive bedre og mere effektiv.

6 Litteraturliste

Pressman, Roger S. and Maxim, Bruce R. 2020. Software Engineering: A Practitioner's Approach. New York: NY. McGraw-Hill Education.

<https://www.teknologisk.dk/kurser/scrum-er-hvad-du-goer-det-til/41615>

<https://www.educative.io/answers/what-is-mvt-structure-in-django>

<https://www.postgresql.org/about/>

<https://www.fullstackpython.com/object-relational-mappers-orms.html>

<https://docs.djangoproject.com/en/4.2/topics/security/#sql-injection-protection>

<https://docs.djangoproject.com/en/5.0/ref/templates/language/>

<https://docs.djangoproject.com/en/5.0/ref/csrf/>

<https://docs.djangoproject.com/en/5.0/topics/http/urls/>

<https://deviq.com/principles/separation-of-concerns>

<https://docs.djangoproject.com/en/5.0/topics/testing/>

[https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Testing#what does django provide for testing](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Testing#what_does_django_provide_for_testing)

<https://aws.amazon.com/what-is/unit-testing/>

<https://www.docker.com/resources/what-container/>

<https://docs.docker.com/compose/>

<https://docs.docker.com/network/>

<https://spacelift.io/blog/docker-security>

<https://docs.docker.com/compose/networking/>

<https://www.spiceworks.com/tech/devops/articles/what-is-jenkins/>

