[Kevin Wallentin Carlsen]

04/03/2024

Space Inventor A/S

Produktrapport

Teknisk dokumentation af ”Server Sentry”

Titelblad

Techcollege Aalborg,
Struervej 70,
9220 Aalborg

Elev:
Kevin Wallentin Carlsen

Firma:
Space Inventor A/S

Projekt:
Produktrapport for ”Server Sentry”

Uddannelse:
Datatekniker med speciale i programmering

Projektperiode:
18/01/2024 – 04/03/2024

Afleveringsdato:
04/03/2024

Fremlæggelsesdato:
11/03/2024

Vejledere:
Simon Hoxer Bønding

Læsevejledning

- ORM (Object Relational Mapping)
- API (Application Programming Interface)
- SQL (Structured Query Language)
- Repo ((GitHub) repository)
- DRF (Django-REST-Framework)

Forord

Denne produktrapport er skrevet i sammenhæng med det tværfaglige projekt på H5PD010124.

Produktrapporten dokumenterer implementation af serveradministrationsværktøj "Server Sentry".

Indholdsfortegnelse

Indledning

Det har fra starten været indtænkt at produktet skal implementeres på min server derhjemme, men samtidigt har det også været vigtigt, for mig, at implementationen er nem i flest mulige passende sammenhænge.

Den réele implementation skal som efter planen bruges af en vennegruppe på 5 mennesker, som skal have mulighed for lettere administration af vores spilservere, mens jeg selv er på arbejde eller ligeledes henholdt fra serveradministration.

Kravspecifikation

Krav kategorier

Tabellen herunder fortæller hvilke kategorier de forskellige krav kan tilhøre, og dermed også hvilken funktion de tjener projektet.

Kategori ID	Navn	Beskrivelse
KN1	Kernefunktionalitet	Funktionaliteter som ikke kan undværes, da de er kritiske for projektets helhed
KN2	Brugervenlighed	Funktionaliteter rettet mod slutbrugere som gør det nemmere for dem at anvende software-pakken
KN3	Telemetri/Logning	Mulighed for at tilgå historisk data for handlinger og statistikker
KN4	Udvidet funktionalitet	Funktionaliteter som er mindre vigtige for projektets helhed. Dermed kan de også nedprioriteres og undværes under tidspres.

Kravspecifikation

Krav ID	Kategori	Beskrivelse	Prioritering
K1	KN1	Administrationsværktøjet kan åbne/lukke de understøttede spilservere	1
K2	KN2	Software-pakken indeholder en app, der som minimum tillader brugere at åbne/lukke servere	2
K3	KN3	Når servere åbnes/lukkes skal administrationsværktøjet kunne logge, af hvem, og hvornår det skete.	2
K4	KN3	Administrationsværktøjet kan vise og logge hvor mange/hvilke spillere der er online på serverne	3

K5	KN2	Appen opdaterer live for at vise aktive spillere/servere	3
K6	KN2	Administrationsværktøjet giver skrive/læse adgang til servernes eventuelle konsoller	4
K7	KN4	Administrationsværktøjet kan indstilles til at tage automatiske backups af save-filer	4
K8	KN4	Administrationsværktøjet inkluderer en web-frontend, udover dets web-api	5
K9	KN2	Appen kan sende push notifikationer når servere åbnes/lukkes og eller når spillere forbinder dertil	5
K10	KN4	Brugere kan tildeles rettigheder til administration (åbne/lukke, osv.) af specifikke servere på administrationsværktøjet	5
K11	KN4	Administrationsværktøjet kan tildele spillere til whitelist/blacklist og administratorlister for de forskellige servere. Hermed menes der om den givne spiller kan forbinde til serveren, og om de har administratorrettigheder deri.	5

Krav matrice

Matrixens celler indeholder ”Krav ID” fra kravspecifikationen, mens deres placering fortæller vigtighed kontra kompleksitet for projetet.

				K2	
^			K5		
|		K11	K4		
|		K8, K9	K3, K7	K6	K1
Vurderet Kompleksitet		K10			
	Vigtighed	-->			

Definition af produktet

Server Sentry lægger kompleksiteterne ved opsætningen og administration af dedikerede spilservere hos dets udviklere, og gør det derfor nemt for brugere.
Software pakken gør udelukkende brug af cross platform teknologier, for sikre at du køre værktøjet hvor det passer dig bedst, og tilgå brugerfladen fra din foretrukne platform.

Software pakken inkluderer både en Flutter og en Django applikation, herudover også en database.

En administrativ brugerflade kan tilgås på Django applikationen, som tilbyder lignende funktionalitet som den sikrere/simplere Flutter frontend.

Begrænsning

Programmet begrænser sig til specifikke spil, for at gøre brugeroplevelsen så problemfri som muligt.
Det er muligt at implementation kunne laves mere generisk i fremtiden, men det nuværende fokus er på bedst mulig understøttelse af de valgte spil.

Testkonditioner

Krav ID	Beskrivelse	Test
K1	Administrationsværktøjet kan åbne/lukke de understøttede spilservere	Opsætningen startes med ”docker compose up –build” hvorefter brugeren kan tilgå Django’s administratorside på ”http://localhost:8000/admin/”. Herefter kan servere oprettes på ”http://localhost:8000/admin/api/gameserver/add/” og åbnes/lukkes vha ”admin actions” på ”http://localhost:8000/admin/api/gameserver/”
K2	Software-pakken indeholder en app, der som minimum tillader brugere at åbne/lukke servere	Opsætningen startes med ”docker compose up –build” hvorefter brugeren kan logge ind og åbne/lukke allerede eksisterende serevere på ”http://localhost:8080/” ved at trykke på ’play’ eller ’pause’ knappen.
K3	Når servere åbnes/lukkes skal administrationsværktøjet kunne logge, af hvem, og hvornår det skete	Når servere åbnes/lukkes (både via Flutter og Django admin), oprettes et nyt event på listen set her: ”http://localhost:8000/admin/api/serverevent/”
K4	Administrationsværktøjet kan vise og logge hvor mange/hvilke spillere der er online på serverne	Under Django Admin (og eller Flutter applikationen) vises som minimum et antal af spillere på hver server i serverlisten (gerne udvidet med navne).
K5	Appen opdaterer live for at vise aktive spillere/servere	Listen af servere i Flutter applikation opdaterer live når ændringer fortages (enten via andre instanser af Flutter, eller Django admin), eller spillere opretter/afslutter forbindelse til serverne
K6	Administrationsværktøjet giver skrive/læse adgang til servernes eventuelle konsoller	Under Django Admin (og eller Flutter applikationen) kan brugere tilgå et

		terminal-lignende vindue, som giver adgang til stdin, stdout og stderr
K7	Administrationsværktøjet kan indstilles til at tage automatiske backups af save-filer	Under Django Admin (og eller Flutter applikationen) forekommer en visning som giver mulighed for at indestille hyppighed for sikkerhedskopiering i et vilkårligt format
K8	Administrationsværktøjet inkluderer en web-frontend, udover dets web-api	Django applikationen inkluderer sin egen brugerflade, uafhængig af Flutter, som yder samme funktionalitet (omfanget af denne alternative brugerflade må afvige fra den normale)
K9	Appen kan sende push notifikationer når servere åbnes/lukkes og eller når spillere forbinder dertil	Afhængigt af platform vises enten snackbar eller anden besked uden for appen når servere åbnes/lukkes og eller når spillere forbinder dertil
K10	Brugere kan tildeles rettigheder til administration (åbne/lukke, osv.) af specifikke servere på administrationsværktøjet	Under Django Admin kan specifikke rettiheder oprettes på ”http://localhost:8000/admin/api/serverpermission/add/”, som respekters brugere forsøger at se/åbne/lukke servere hvor muligt
K11	Administrationsværktøjet kan tildele spillere til whitelist/blacklist og administratorlister for de forskellige servere. Hermed menes der om den givne spiller kan forbinde til serveren, og om de har administratorrettigheder deri	Under Django Admin (og eller Flutter applikationen) kan en side ligende ”http://localhost:8000/admin/api/serverpermission/add/” tilgås, hvorpå brugere kan tillades eller forbydes fra at forbinde til valgte server. Hermed håndteres implementationsdetaljer vedrørende whitelists/blacklists til de forskellige spil

Brugervejledning

Installation

Pakker anbefalet til installation og opsætning

- Docker (version ^25.0.3, som bør inkludere Docker Compose)
- Git
- Python 3 (version ^3.10)
- Flutter / Dart

Docker Compose anbefales til opsætningen af systemet, hvilket tillader at alle tjenester startes med én enkelt kommando.
Installationproceduren for Docker varirer afhængt af installationsdestinationens platform, derfor bedes administratoren følge Dockers officielle guide, fundet her: https://docs.docker.com/engine/install/

Projektets afhængigheder afhænger af tilpas nye "Docker Compose" funktionaliter, som kun kan findes når kommandoen køres som "docker compose" og **ikke** "docker-compose".

Opsættes systemet på Ubuntu, kan følgende kommandoer som udgangspunkt anvendes til installationen af afhængigheder:

```
    # Docker

    # Add Docker's official GPG key:
    sudo apt-get update
    sudo apt-get install ca-certificates curl
    sudo install -m 0755 -d /etc/apt/keyrings
    sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
/etc/apt/keyrings/docker.asc
    sudo chmod a+r /etc/apt/keyrings/docker.asc

    # Add the repository to Apt sources:
    echo \
    "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/ubuntu \
    $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
    sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
    sudo apt-get update

    sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin
docker-compose-plugin


    # Git

    sudo apt install git
```

Herefter anvendes følgende kommandoer til at hente og starte programmet:

```
git clone https://github.com/kivkiv12345/tverfagligt_projekt.git

cd tverfagligt_projekt

# Her forventes det at start af Flutter webserveren kan tage lang tid.
docker compose up
```

Anvendelse

Oprettelse af brugere

Alemene brugere af projektet applikationer tillades _ikke_ at registrere sig selv, hermed ville de kunne tildele sig selv administrationsrettiheder som ellers _kun_ udstedes til bestemte brugere.
I stedet er administratorbrugere ansvarlige for oprettelsen af yderligere brugere (administratorer eller ej).
Køres Django applikationen med "docker compose up" kan programmets første administratorbruger kan oprettes med følgende kommandoer:

```
docker exec -it tverfagligt_projekt-api-1 /bin/bash
python3 manage.py createsuperuser
```

Herefter bedes administratorbrugeren at besvare opfordringerne med deres ønskede legimitationsoplysninger, hvorefter de ville kunne tilgå både "Django Admin" og Flutter (se tilsvarende afsnit).

Flutter

Flutter applikationen kan nemmest tilgås vha. dets web-applikation.
Køres Flutter applikationen med "docker compose up" kan brugerfladen tilgås på 127.0.0.1:8080

På det overstående link bliver brugeren først mødt af en 'login' formular. Udfyldes og indsendes denne formular med brugerens legimitationsoplysninger (se "oprettelse af brugere"), vil de derefter videresendes til listen af servere som denne bruger er bevilget at se/administrere.

På denne side kan brugeren se hvilke servere som er aktive/inaktive.
Vises serveren med et "pause" ikon (firkant), er den på nuværende tidspunkt aktiv.
Trykker brugeren på ikonet deaktiverer de serveren.
Modsat set vises inaktive servere med et "afspil" ikon (trekant), og kan aktiveres med et tryk på ikonet.

Når brugeren engang er færdige med at starte/stoppe servere, kan de 'logge ud' af applikationen ved først at tykke på burgermenuen (set øverst i venstre hjørne), og derefter "Log Out". Herefter slettes brugerens gemte legimitationsoplysninger, hvorefter de tilbagesendes til 'login' formularen.

Burgermenuen tillader også brugeren at ændre mellem lys/mørk tilstand af applikationen.
Standardtilstanden følger brugerens valg fra operativsystemet.

Django Admin

Django kommer med en nem tilpasselig administrationsside som en del af deres contrib pakke.
Django applikationen implementerer derfor denne administrationsside, for at afprøve og understøtte udviddet funktionalitet som endnu ikke findes i Flutter applikationen, såsom oprettelse af nye servere.

Køres projektet lokalt kan administrationssiden findes på 127.0.0.1:8000/admin

På det overstående link bliver brugeren mødt af en side som viser de mest væsentlige modeller fra projektets database.
Herpå fører linket "Game servers" til "http://localhost:8000/admin/api/gameserver/" som er en liste af spilservere som kan sammenlignes med den fundet i Flutter applikationen. Dog adskiller denne liste sig med tilføjelsen af knappen "Add game server", som fører til http://localhost:8000/admin/api/gameserver/add/ hvor nye servere kan tilføjes. På denne side udfylder brugeren en formular hvor server navn, standardtilladelser og spil specificeres. Klikker brugeren herefter på én af de adskillige "save" knapper oprettes en ny server.
Brugeren kan herefter starter serveren ved at navigere tilbage til "http://localhost:8000/admin/api/gameserver/", hvor de kan markere deres server i listen og vælge "Start servers" i "Action" rullemenuen.

Service

Se afsnittet "Regristrering af spil / Manager klasser" for understøttelse af flere spil.

Teknisk produktdokumentation

Dette afsnit dækker væsentlige elementer for systemets nuværende implementation, samt mulige udvidelser og refaktoring dertil.

Docker

Docker er et meget veletableret virtualiserings værktøj.
Docker adskiller sig fra typiske virtualiserings værktøjer ved ikke at bruge virtuelle maskiner, i virkeligheden separerer Docker bare processer ved at bruge namespaces.

Docker anvendes til at forenkle opsætningen af system, og samtidigt også gøre det nemt at slette bagefter.
Django applikationen er bygget således at spilservere skulle kunne drives af vilkårlige teknologier, men på nuværende tidspunkt forkommer "Docker Compose" som den eneste konkrete implementation.
Hermed kræves det at Docker er installeret på serveren.

Django

Som backend framework bruger projektet Django, som er et Python web framework med fokus på hurtig udvikling med indbygget sikkerhed.
Det primære formål med projektets Django backend er at tilbyde et web-API som muliggør at administrere spilserverne fra Flutter applikationen. Derfor skal backenden også tilbyde et fælles interface til diverse implementationer af spilservere.

Rodmappen for Django applikationen (i forhold til projektets rodmappe) findes i ./backend/
Stier forklaret i dette afsnit, tager derfor udgangspunkt i denne mappe.

Når man laver et Django projekt, laver men deri efterfulgt en (eller flere) "app(s)", hvori alt implementation forgår. Der findes ingen fast afgrænsning for hvad én app må indeholde, og hvornår projektet skal implementers på tværs af flere apps.

Regristrering af spil / Manager klasser

Projektets Django applikation adskiller sig fra typiske Django eksempler med Manager klassernes arvehierarki. Her registreres AbstractGameServerManager konkrete underklasser automatisk, hvorefter det nye spil, også automatisk, bliver en valgmulighed for brugere når de laver deres servere.

```
Bemærk dog at nye valgmuligheder for spil (konkrete AbstractGameServerManager
underklasser) kræver en opdatering til databasen!
For at udføre denne opdatering manuelt benyttes følgende kommandoer:

python3 ./manage.py makemigrations
python3 ./manage.py migrate

Benytter man Docker eller "Docker Compose" udføres disse kommandoer automatisk, som set
nederst i ./Dockerfile
```

Manager klasserne er derfor ansvarlige for at implementere den endegyldige forretningslogik angående håndtering af spilservere.
I et typisk Django projekt ligger den endegyldige forretningslogik direkte på databasemodellerne (hvor meget typisk ligger i Model.save()).
Den fordeling er ulempelig her, da den kræver kobling til konkrete nedarvede spilklasser (mht. forespørgsler til databasen).
Alternativt kan man i fremtiden refaktorere databasemodellerne til at bruge pakken Django-Polymorphism, i stedet for brugen af Manager klasserne. Brugen Django-Polymorphism tillader forespørgsler på abstrakte superklasser, som dermed tillader løs kobling. Bemærk dog at dette kan medføre værere ydeevne, da hver konkret spilklasse kræver én yderligere SQL JOIN i forespørgslerne.

Som den første konkrete spilklasse understøtter projektet servere til spillet Vintage Story, som i baggrunden kører vha. Docker-Compose. Men med Manager klassernes arvehierarki, kan udvikleren nemt udskifte hvordan spilserveren i virkeligheden køres. Som eksempel på hvordan én konkret underklasse kan se ud, vises VintageStoryManager nedenunder:

```
class VintageStoryManager(GitHubVersionedDockerComposeManager):
    compose_file = 'repos/docker-vintagestory/docker-compose.yml'
    version_commit_map = {
        'newest': 'master',
        '1.19.3': '5ce36b8a75e909fa30bfdca9f20a2ac46000fdbf',
        '1.19.2': '71fab0d025eecbd73884be8dedced5c65226298d',
        '1.18.15': 'dd4818b90f74786442f7d1985bcb644915d884c1',
        '1.18.1': '558e4aa364a38f1ea8af1caf32c4f94240bfba1a',
    }
    repo = 'https://github.com/Devidian/docker-vintagestory.git'
    services = ['vsserver-local',]

    game_versions = (
        'newest',
        '1.19.3',
        '1.19.2',
        '1.18.15',
        '1.18.1',
    )
```

Klassen defineres udelukkende med klassevariabler som kræves af superklasserne, som hermed håndterer registrering og metoder.
Set her er kravene som følgende:

- VersionedGameServerManager
  - game_versions: Sequence[str]
- GitHubVersionedManager
  - version_commit_map: dict[str, str]
  - repo: str
- AbstractDockerComposeGameServerManager
  - compose_file: str
  - services: list[str] | str

Bemærk at konkrete underklasser skal defineres i filer som indlæses af Django under opstart, derfor er VintageStoryManager defineret i ./api/apps.py

Oprettelse- og vedligeholdelse af web-API endpoints

Endpoints defineres i filen *./api/views.py* og regristeres med dekoratoren *@api_view()*, hertil angives hvilke HTTP forspørgelesemetoder endepunktet skal acceptere.

Endepunktet synliggøres ved at specificere et, eller flere, URL som skal bruge det.
Det gør man i urls.py filen tilsvarende til sin app, til projektet her er det ./api/urls.py

Denne fil skal indeholde en variablen navngivet "urlpatterns" af typen list[path]:

```
from django.urls import path
from api.views import stop_server

urlpatterns = [
    # ...
    path(f"stop-server/", stop_server, name="stop-server"),
    # ...
]
```

Bemærk at "path"s constructor første argument er stien som skal føre til ens endepunkt/view, og derefter funktionen/klassen selv (class based view eksisterer også, men er ikke brugt i projektet her).
Typisk angiver man også et "name=" til sit URL/view. Herefter kan man omdiregere til navnet på sit URL/view, og derfor flytte dets placering efter behov.

Argumenter til endepunktet håndteres typisk forskelligt afhængigt af om man laver et GET eller POST endpoint:

- POST: Argumenter hertil modtages typisk som JSON i kroppen af HTTP forespørgslen.
Følgende signature viser hvordan argumenter til endepunktet kan findes i request.data:

```
    @api_view(['GET'])
    def is_server_running(request: Request) -> Response:
        print(request.data)
```

- GET: Til GET endepunkter laves argumenter typisk i URLet (som forspørgelsesargumenter).
Følgende path viser hvordan forspørgelsesargumenter registreres samtidigt med selve URLet til ens endepunkt i ./api/urls.py, hvor argumentet er angivet som \<str:ident>:

```
path(f"get-server-version/<str:ident>", get_server_version, name="get-server-version")
```

Hermed genererer (DRF) Django-REST-Framework en webside hvor man kan afprøve sit endpoint.
Denne webside kan tænkes som en simpel version af Swagger, da den ikke har kendskab til endepunktet accepterede argumenter (det er man selv ansvarlig for at dokumentere).

Heldigvis præsenterer DRF endepunkt funktionernes dokumentationstreng som dokumentation, hvilket vil sige at hvis man laver en multilinje streng straks efter funktionens signatur, bliver den præsenteret til brugere af web-APIet:

```
    @api_view(['GET'])
    def get_server_version(request: Request, ident: str) -> Response:
        """

        Expected JSON:
        {
            "server_ident": int | str,
        }
        """
```

Overordnet struktur på endpoints

Den typiske struktur til Django applikationens web-API er som følgende:

1. Validér argumenter
2. Find efterspurgte server
3. Tjek tilladelser
4. Kald efterspurgte GameServerManager funktion

Hermed menes der at typiske spilserver forespørgsler er implementeret som selvstændige funktioner på Manager klasserne.

Django Admin

Django's administrationsside kan nemt tilpasses med udviddet funktionalitet.

Admin actions

Som et simpelt eksempel kan man lave "action"s, som tillader udførelse at komplekse på mange server (model instanser) samtidigt. Definerer man den funktion man vil køre (brugen af dekoratoren "@admin.action()" anbefales, men er ikke nødvendig):

```
@admin.action(description="Start servers")
def start_servers(modeladmin: ModelAdmin, request: Request, queryset:
QuerySet[GameServer]):
    for server in queryset:
        server.manager.start()
```

Argumentet "queryset" er de valgte instanser på listen af sine modeller (servere), som brugeren ønsker handlingen udført på.

Herefter registrerer ens "action" på admin klassen tilsvarende ens model:

```
class GameServerAdmin(ModelAdmin):
    model = GameServer
    ...
    actions = (..., start_servers, ..., )
    ...
```

Med følgende eksempel kan man herefter starte mange servere samtidigt på administratorsiden.

Simple ændringer af administratorsiden

Administratorsiden er opbygget med serverbaseret rendering, hvormed brugerændringer som udgangspunkt forekommer i Python, fremfor HTML.

Mærkværdigt er det faktum at det ikke nødvendigvis er alle spilserverklasser som kan håndtere versionering, derfor håndteres dette på administratorsiden med betinget rendering af ændringsformen. Metoden GameServerAdmin.get_form() viser hvordan spilklasser som ikke nedarver fra VersionedGameServerManager præsenteres uden versionering:

```
def get_form(self, request, obj: GameServer = None, change=False, **kwargs):

    ...

    class GameServerAdminForm(ModelForm):
        if isinstance(obj.manager, VersionedGameServerManager):
            CHOICES = ((game_version, game_version) for game_version in
obj.manager.available_versions)
            server_version = forms.ChoiceField(choices=CHOICES,
initial=obj.server_version)
        server_running = forms.BooleanField(initial=obj.manager.server_running(),
required=False)

        ...

    return GameServerAdminForm
```

Flutter

Programmet præsenteres af Flutter, som er et cross-platform frontend framework, hvormed den samme kode derfor kan præstere på: mobil, computere, web, mm.
Kører man systemet med Docker, kan Flutter tilgås som en webapp på port 8080.

```
Rodmappen for Flutter applikationen (i forhold til projektets rodmappe) findes i
./gameserver_frontend/
Stier forklaret i dette afsnit, tager derfor udgangspunkt i denne mappe.
```

Login proceduren fungerer ved at Flutter sender de indtastede legitimationsoplysninger (brugernavn og kodeord) til Django applikationen, som derefter verificerer oplysningerne.
Herefter svarer Django applikationen med en "Token" som Flutter applikationen bruger til at verificere brugerens identitet ved efterfølgende forespørgsler.

Flutter applikationen bruger pakken shared_preferences til at persitere brugernavn og token til disken på tværs af sessioner. Hermed behøver brugere, som adgangspunkt, kun at logge ind én enkelt gang.

Flutter applikationen gør stort brug af pakken Bloc til at håndtere opdateringer af brugerfladen.
Disse opdateringer indebærer på nuværnde tidspunkt:
- Bemyndigelse af brugere ved login/logout.
- Ændring mellem mørkt og lyst tema.
- Ønskede og reelle opdateringer af servere (start/stop osv.)

En 'Bloc' er en kombineret state maskine og observer pattern. Brugere af en Bloc kan skabe 'Events' som kan påvirke den aktuelle Bloc instans til at skifte tilstand ("state"). Denne tilstand udsendes herefter til relevante lyttere (BlocBuilder instanser), som genbygger dele af brugerfladen baseret på informationen inkluderet i den udsendte tilstand instans.
Brugerfladen mest simple eksempel ses i form af ThemeBloc:

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

class ThemeBloc extends Bloc<ThemeMode, ThemeMode> {

  ThemeBloc(super.initialState) {
    on<ThemeMode>(themeChange);
  }

  void themeChange(ThemeMode theme, Emitter<ThemeMode> emit) {
    emit(theme);
  }
}
```

Denne Bloc klasse bruger både ThemeMode klassen som 'events' og 'states', hermed behøver Bloc'en blot videresende det modtagede 'event' til lyttende BlocBuilder instanser:

```
BlocBuilder<ThemeBloc, ThemeMode>(
    builder: (context, themeMode) {
        return MaterialApp(
        ...
        themeMode: themeMode,
        home: const MyHomePage(title: 'Flutter Demo Home Page'),
        );
    },
),
```

I det overstående eksempel er themeMode argumentet den videresendte ThemeMode fra vores ThemeBloc. Hermed bygges resten af applikationen med det signalerede tema. Temaet signaleres når brugeren trykker på følgende ListTile:

```
ListTile(
    title: Text('Dark Mode'),
    trailing: Switch(
        ...,
        onChanged: (value) {
            // Toggle dark mode
            ThemeMode newThemeMode =
                value ? ThemeMode.dark : ThemeMode.light;
            ...
            BlocProvider.of<ThemeBloc>(context).add(newThemeMode);
        },
    ),
),
```

Switch.onChanged() finder den nærmeste overstående ThemeBloc i widget træet, og giver den et event om at ændre temaet til det modsatte af den nuværnde.

Database

Systemet bruger på nuværende tidspunkt én enkelt SQLite database, for at mindske kompleksiteten.
Takket været Django's ORM, kan databasen nemt ændres til et mere skalérbart alternativt efter behov.
Ønskes en PostgreSQL database, kan følgende ændreringer anvendes:

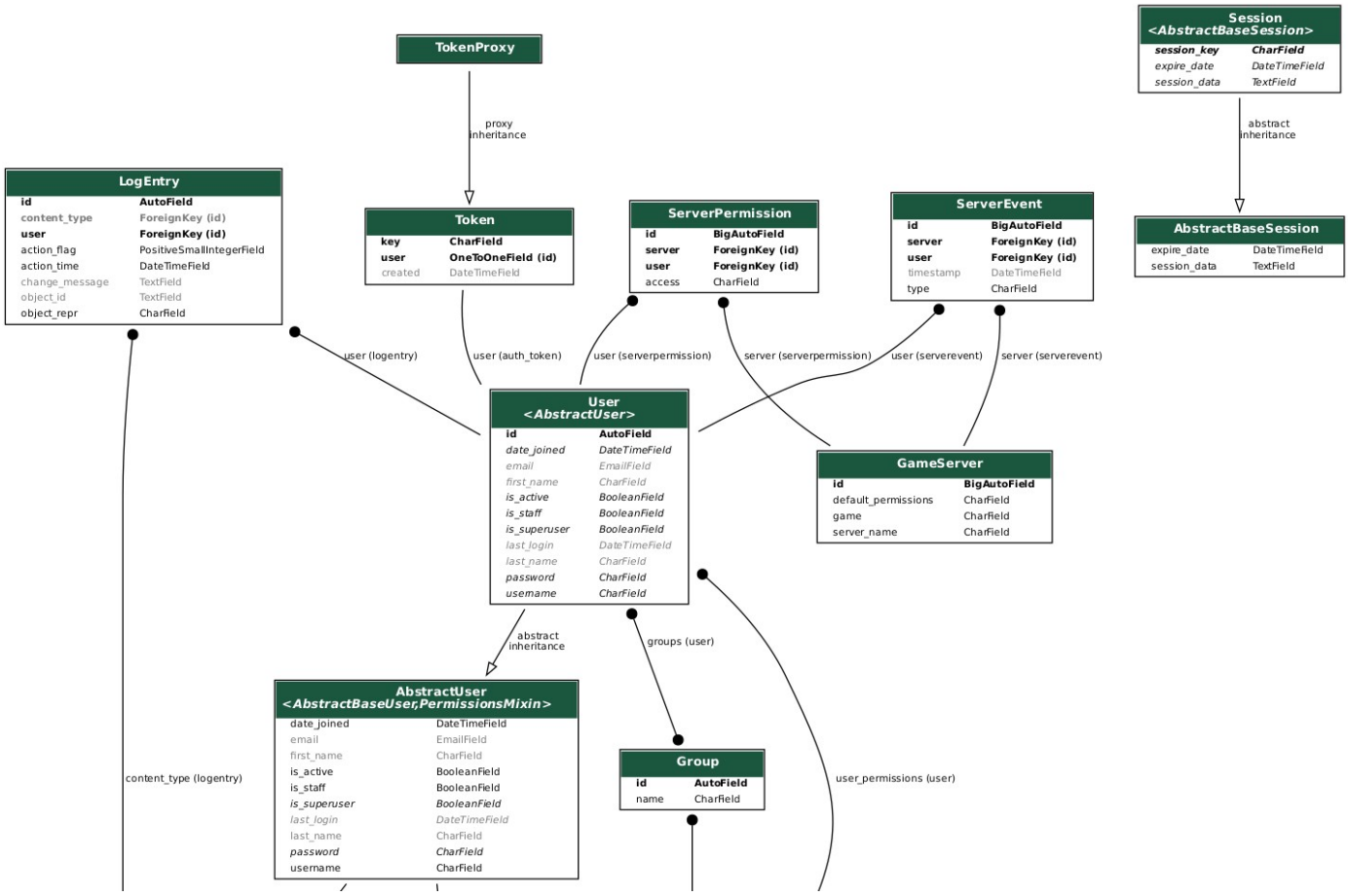
- Ændre "DATABASES" i ./backend/settings.py til:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.environ.get('POSTGRES_NAME'),
        'USER': os.environ.get('POSTGRES_USER'),
        'PASSWORD': os.environ.get('POSTGRES_PASSWORD'),
        'HOST': 'db',
        'PORT': 5432,
    }
}
```

- Tilføj service til docker-compose.yml:

```
services:
    ...
    db:
    image: postgres
    volumes:
    - ./db_volume/db:/var/lib/postgresql/data
    environment:
    - POSTGRES_DB=postgres
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=postgres
    ports:
    - "5432:5432"  # Forward the port, so we can use the container when running Django locally for debug purposes.
    …
```

Herunder ses databasen som genereret fra Django's ORM. Herpå skal sammenspillet mellem: "User", "ServerPermission" og "GameServer" ses som vigtigst. Disse modeller muliggør kernefunktionaliterne ved programmet.



Testrapport

Krav ID	Beskrivelse	Gennemført	Note
K1	Administrationsværktøjet kan åbne/lukke de understøttede spilservere	100%	Både muligt gennem Flutter og Django-Admin
K2	Software-pakken indeholder en app, der som minimum tillader brugere at åbne/lukke servere	100%	Krav gennemført med Flutter applikationen
K3	Når servere åbnes/lukkes skal administrationsværktøjet kunne logge, af hvem, og hvornår det skete	100%	Implementeret på Manager klasserne, og virker derfor både med Django-Admin og Flutter
K4	Administrationsværktøjet kan vise og logge hvor mange/hvilke spillere der er online på serverne	30%	PlayerEventStreamReader kan udlæse antallet af spillere fra server logs
K5	Appen opdaterer live for at vise aktive spillere/servere	70%	Websockets er understøttet, men forrige krav mangler før de kan bruges til at vise spillere
K6	Administrationsværktøjet giver skrive/læse adgang til servernes eventuelle konsoller	0%	Nedprioriteret og unladt grundet kompleksitet
K7	Administrationsværktøjet kan indstilles til at tage automatiske backups af save-filer	20%	Web-API og domænemodel inkluderer backup metoder, men uden konret implementation
K8	Administrationsværktøjet inkluderer en web-frontend, udover dets web-api	100%	Django Admin er konfiguret til almen brug som administratorværktøj
K9	Appen kan sende push notifikationer når servere åbnes/lukkes og eller når spillere forbinder dertil	0%	
K10	Brugere kan tildeles rettigheder til administration (åbne/lukke, osv.) af specifikke servere på administrationsværktøjet	100%	Servere skjules for brugere med manglende rettigheder i Flutter applikationen.
K11	Administrationsværktøjet kan tildele spillere til whitelist/blacklist og administratorlister for de forskellige servere. Hermed menes der om den givne spiller kan forbinde til serveren, og om de har administratorrettigheder deri	0%	

Referencer

Inspiration til “ServerEvent Enum” som “TextChoices felt” I Django applikationen:
https://stackoverflow.com/questions/54802616/how-can-one-use-enums-as-a-choice-field-in-a-django-model
Her kunne “django-enum” pakken også være brugt

https://stackoverflow.com/questions/60869395/python-covert-enum-to-django-models-charfield-choices-tuple?noredirect=1&lq=1

Docker SDK i container
https://stackoverflow.com/questions/60054378/i-want-to-use-docker-sdk-inside-a-running-docker-container-and-want-to-build-an

At GameServer.server_name skal være ufølsomme for store og små bogstaver og unik:
https://stackoverflow.com/questions/7773341/case-insensitive-unique-model-fields-in-django

Flutter mørk tilstand:
https://stackoverflow.com/questions/60232070/how-to-implement-dark-mode-and-light-mode-in-flutter

Brug og opsætning af Django Channels til websockets:
https://channels.readthedocs.io/en/latest/installation.html

Brug og opsætning af Flutter/Dart channels:
https://pub.dev/packages/web_socket_channel/example