Ce fichier Readme contiendra une explication de l'Infrastructure Réseaux, la structure de la base de données, les schémas des Urls de mon proet et les réponses aux questions Django et Docker.

Plan :

1. Infrastructure Réseaux :

- **Réseau "back"** : Un réseau interne pour les services qui ne sont pas directement accessibles depuis l'extérieur (API et base de données).
    - Services :
        - **API** : est un service backend construit avec Django REST Framework. Elle expose des endpoints qui permettent aux utilisateurs ou à d'autres services d'interagir avec les données du système via des requêtes http et au **Serializers** qui servent à convertir les objets Django en formats comme JSON, afin de permettre l'échange de données entre le frontend et le backend.
        - **Base de données**
- **Réseau "front"** : Un réseau destiné aux services accessibles depuis l'extérieur (public, proxy).
    - Services :
        - **Public** : Le service frontend qui est accessible au public via le proxy.
        - **Proxy** : Le reverse proxy (Nginx) qui gère les connexions entre le frontend public et l'API backend.
        - **pgAdmin** : Interface de gestion de la base de données, accessible via un navigateur.
        - **Base de données (db)** : La base de données est partagée entre les réseaux **back** et **front** pour permettre l'accès des services backend et frontend.

Interactions avec la Machine Hôte

- Le trafic externe est géré par le **proxy**, qui est accessible sur le port 80 de la machine hôte.
- L'interface d'administration **pgAdmin** est accessible via le port 5050 depuis la machine hôte.

Schéma de l'infrastructure réseau :

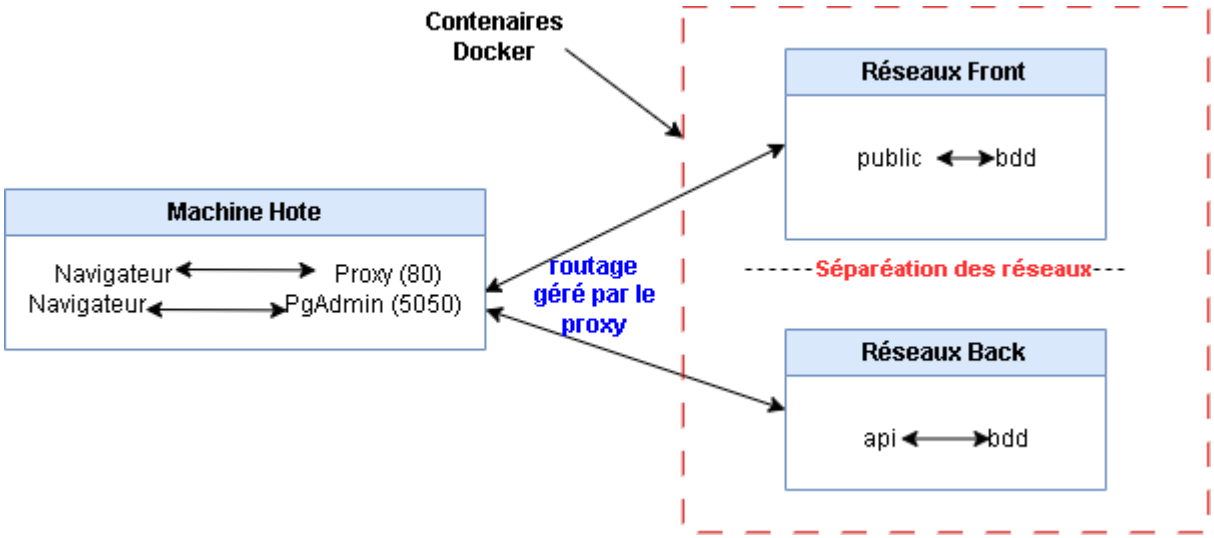
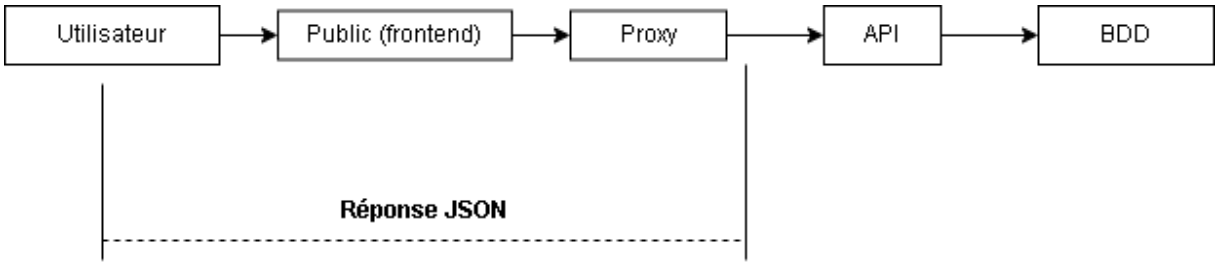
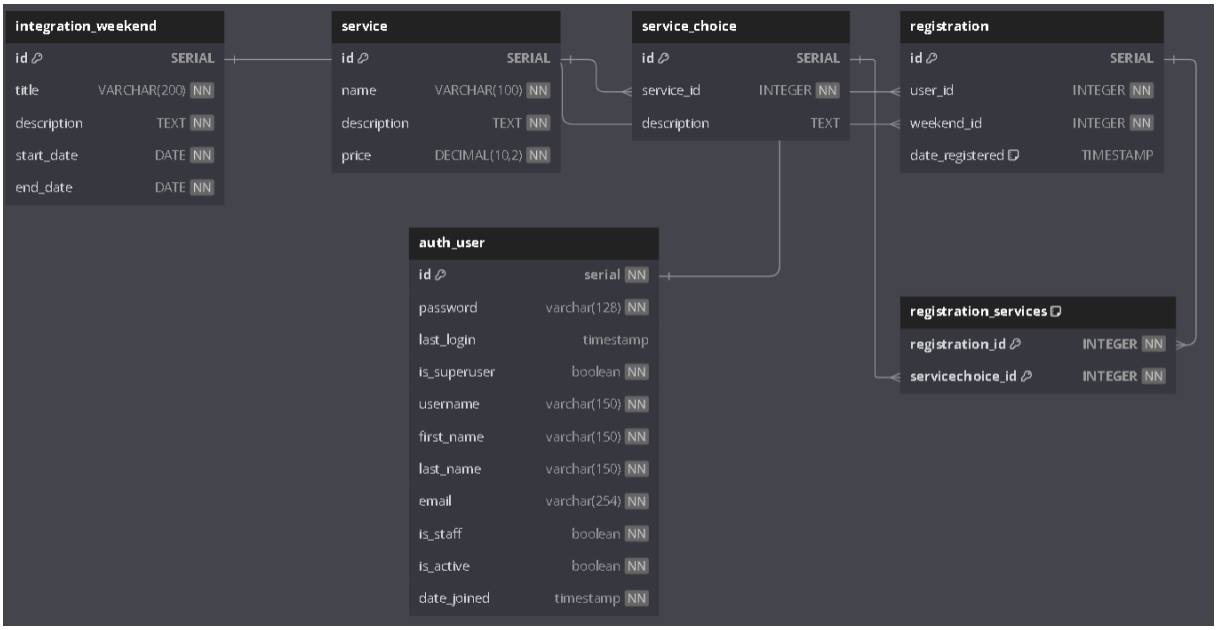


Schéma d'Interaction API <-> Frontend :



2. BDD :

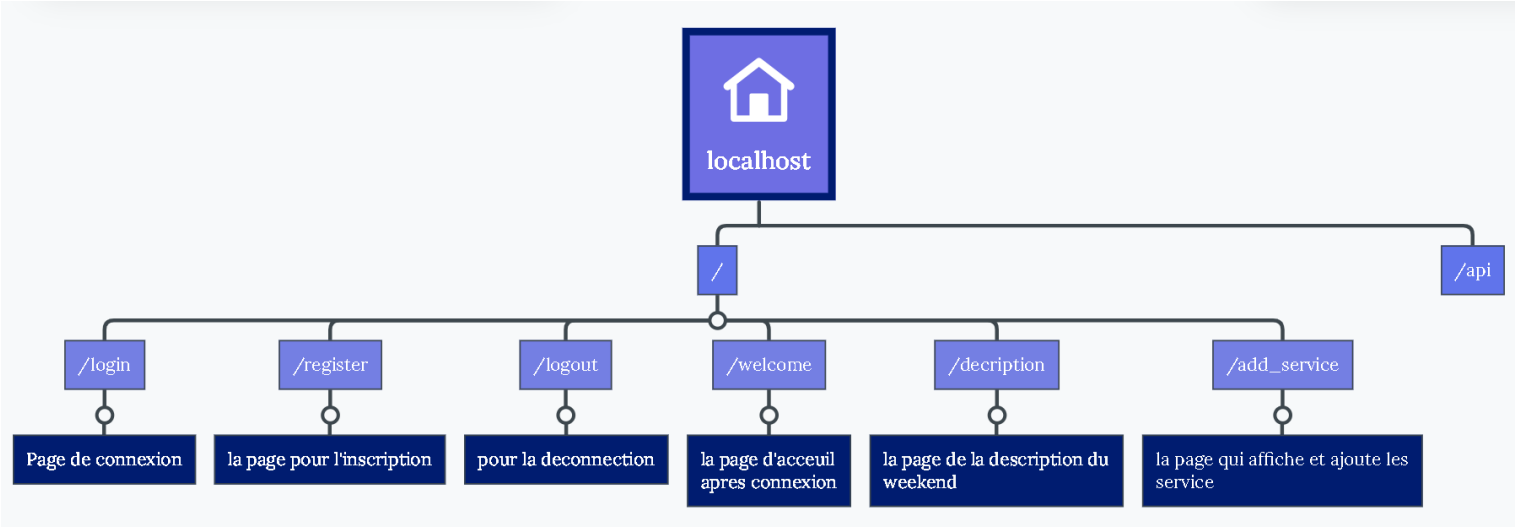


PS : la Table « auth_user » c'est un modèle importé de Django.

Détails des relations :

- **IntegrationWeekend :** Contient les informations sur le week-end d'intégration.
- **Service :** Définit les services proposés lors du week-end (Tournois/ soirées…).
- **ServiceChoice :** Permet aux utilisateurs de choisir des services spécifiques.
- **Registration :** Contient l'inscription des utilisateurs à un week-end d'intégration et les services qu'ils ont choisis.

3. Structure Chemins URL :



4. Questions :

Django :

1. **Vous disposez d'un projet Django dans lequel une application public a été créée. Décrivez la suite de requêtes et d'exécutions permettant l'affichage d'une page HTML index.html à l'URL global / via une application public, ne nécessitant pas de contexte de données. Vous décrirez la position exacte dans l'arborescence des répertoires des différents fichiers utiles à cette exécution.**

Pour afficher une page html index.html sur l'url /, nous allons procéder en plusieurs étapes :

- Définir la vue dans l'application public dans «views.py» :

```
3  def index(request):
4      return render(request, 'public/index.html')
5
```

- Définir les URLs dans l'application public dans « public/urls.py» :

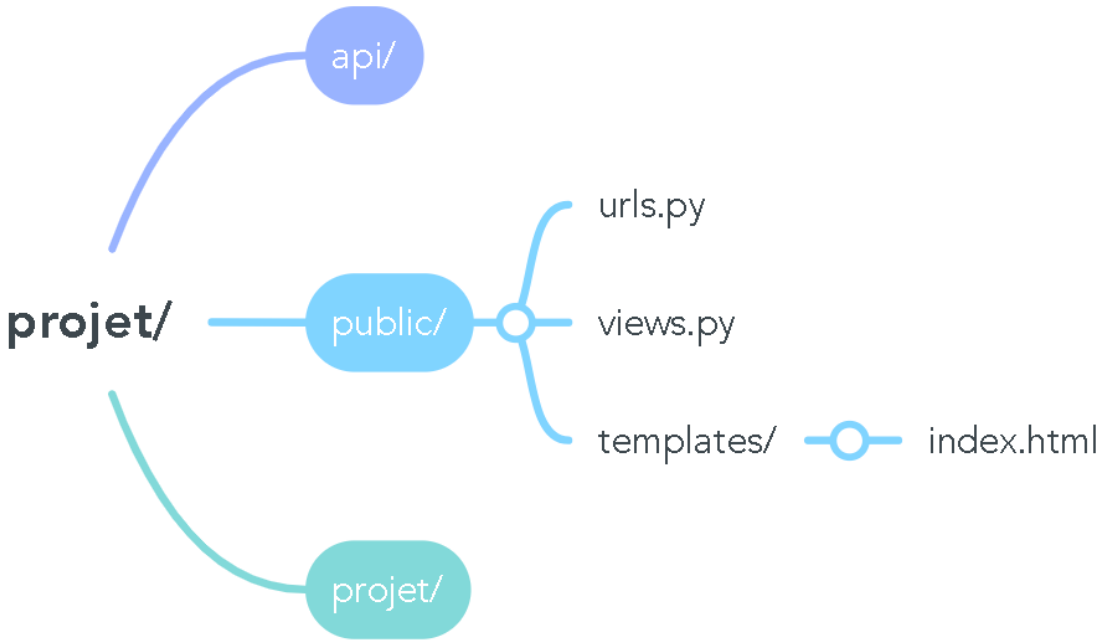
```
6  urlpatterns = [
7      path('', views.index, name='index'),
8  ]
```

- Inclure les URLs de l'application public dans les URLs globales (projet/urls.py):

```
10  urlpatterns = [
11      path('admin/', admin.site.urls),
12      path('', include('public.urls')),
13  ]
```

- Créer le fichier HTML index.html sous le dossier « templates/»

- Arborescence finale :



PS : Ne faut pas oublier de configurer l'emplacement des « TEMPLATES » dans le fichier de configuration (settings.py) car dans mon cas j'avais un problème avec l'emplacement par défaut.

2. **Dans quelle(s) section(s) de quel(s) fichier(s) peut-on configurer la base de données que l'on souhaite utiliser pour un projet Django ?**

   La configuration de la base de données se fait dans le fichier « settings.py» de l'application principale (« projet/ » pour mon exemple). La configuration de la base de données se trouve dans la section DATABASES de ce fichier.

3. **Dans quel(s) fichier(s) peut-on configurer le fichier de paramètres que l'on souhaite faire utiliser par le projet Django ? Si plusieurs fichiers sont à mentionner, expliquez le rôle de chaque fichier.**

Par défaut, nous utilisons le fichiers « settings.py» pour tous le projet (il contiendra les paramètres comme pour la bdd, les application à charger, les templates…). Mais dans le cas ou on veut avoir des paramétrages différents (l'utilisation des paramètres différents pour des centenaires docker différents par exemple), nous pouvons créer un fichier de settings commun qui contiendra les paramètres communs et d'autres pour spécifier les paramètres à changer.

- **settings.py** : Fichier commun, Ce fichier agit comme un point central contenant les paramètres partagés entre l'API et le front-end.
- **settings_api.py** : Fichier spécifique pour l'API**,** Ce fichier est dédié à l'API et inclut les configurations nécessaires pour gérer les requêtes backend, incluant des apps comme rest_framework.
- **settings_front.py** : Fichier spécifique pour le front, Spécifique à la partie front-end, il gère l'affichage des templates et les applications liées à l'interface utilisateur.

Ensuite, nous pouvons alors utiliser la variable d'environnement « DJANGO_SETTINGS_MODULE » pour spécifier quel fichier de configuration doit être utilisé.

- **Pour l'API :** export DJANGO_SETTINGS_MODULE=projet.settings_api
- **Pour le front :** export DJANGO_SETTINGS_MODULE=projet.settings_frontm

Cela nous permet de séparer proprement les configurations tout en partageant les paramètres communs nécessaires.

4. **Nous nous plaçons à la racine de votre projet Django. Quel effet a l'exécution python manage.py makemigrations ? Et l'exécution python manage.py migrate ? Quel(s) fichier(s) sont mis en oeuvre pendant ces exécutions ?**

python manage.py makemigrations :

- **Effet** : Cette commande examine les fichiers models.py pour détecter les modifications dans la structure de la base de données (comme l'ajout de nouveaux champs, la suppression de champs…). Ensuite, elle génère des fichiers de migration qui décrivent ces changements.

python manage.py migrate :

- **Effet** : Cette commande applique les fichiers de migration générés à la base de données. Cela signifie que Django va exécuter les instructions contenues dans les fichiers de migration pour créer ou modifier les tables et les colonnes afin de refléter les changements définis dans les modèles.

Docker :

1.Expliquez l'effet et la syntaxe de ces commandes, communément vues dans des fichiers Dockerfile : FROM, RUN, WORKDIR, EXPOSE, CMD.

FROM : La commande définit l'image de base à partir de laquelle le conteneur Docker sera construit. (version de l'image python par exemple)

RUN : La commande exécute une ou plusieurs commandes dans le conteneur lors de la construction de l'image. Elle est utilisée pour installer des dépendances par exemple.

WORKDIR : La commande définit le répertoire de travail dans le conteneur où toutes les commandes seront exécutées.

EXPOSE : La commande indique le ou les ports que le conteneur utilise pour écouter les connexions réseau.

PS : Cette commande n'ouvre pas réellement le port, mais sert juste d'indication.

CMD : c'est la commande par défaut à exécuter lorsque le conteneur est lancé. Contrairement à RUN qui est exécutée lors de la construction de l'image, CMD est exécutée lors du démarrage du conteneur.

2. Explications des champs du docker-compose.yml :

ports: Configure un mappage entre les ports de l'hôte et du conteneur. (Cela signifie que si le conteneur exécute une application, elle sera accessible via http://localhost:80)

build: Spécifie comment construire l'image Docker à partir d'un Dockerfile et ou le trouver.

depends_on: Définit les services qui doivent être démarrés avant ce service.

environment: Définit des variables d'environnement pour le service, souvent utilisées pour la configuration dynamique(par exemple : les informations de connexion à une base de données).

3. Citez une méthode pour définir des variables d'environnement dans un conteneur :

L'une des méthode pour définir des variables d'environnement est dans un **Dockerfile** ou on peut utiliser la commande **ENV** pour définir des variables d'environnement.

Cette méthodes servent à configurer le comportement du conteneur à l'exécution. (cf question 3 Django)

4.Config accès au public sans passé par l'adresse IP dans le proxy :

Pour permettre à un conteneur **nginx** de communiquer avec un conteneur **public** sans utiliser d'adresses IP, on peut utiliser le **nom du service Docker**. Docker Compose attribue automatiquement un nom DNS à chaque service, permettant aux conteneurs de se

communiquer par leurs noms de service. Dans le fichier de configuration Nginx, on peut alors utiliser le proxy_pass : « http://public:8000 » pour faire pointer vers le conteneur **web** via son nom de service.