Documentation technique

Introduction

L'application a été créée pour les Jeux Olympiques de 2024 et permet aux utilisateurs de visualiser les événements et les différentes épreuves qui se dérouleront pendant cette période. L'objectif principal de l'application est de fournir une plateforme conviviale pour les spectateurs, leur permettant de consulter le programme des compétitions et d'acheter des billets en ligne. Le front-end de l'application a été développé avec HTML, CSS, Bootstrap et JavaScript pour offrir une interface utilisateur dynamique et responsive. Le back-end repose sur le framework Django, utilisant Python pour gérer la logique serveur et les bases de données. L'application est hébergée sur un serveur Heroku et utilise l'add-on JAWSDB MariaDB pour la gestion des bases de données, situé en Europe, garantissant ainsi une haute disponibilité et de bonnes performances.

Installation et configuration

1. Vérification de Python

Assurez-vous d'abord que Python est installé sur votre machine. Vous pouvez vérifier cela avec la commande suivante :

```powershell
python --version
```

Il est recommandé d'utiliser Python 3.x. Si Python n'est pas installé, vous pouvez le télécharger depuis le site officiel.

2. Création d'un environnement virtuel

Avant d'installer les dépendances du projet, il est important de créer un environnement virtuel afin de bien isoler les dépendances. Exécutez les commandes suivantes dans votre terminal :

```powershell
python -m venv venv
venv\Scripts\activate
```

Une fois l'environnement virtuel activé, vous pouvez installer les dépendances à l'aide de pip.

3. Installation de Django

Installez la version 4.2 de Django via le gestionnaire de paquets pip :

```powershell
pip install django==4.2
```

Vous pouvez aussi installer toutes les dépendances spécifiées dans le fichier requirements.txt pour assurer la compatibilité des versions. Ce fichier est disponible dans le projet.

4. Création du projet Django

Après l'installation de Django, créez un nouveau projet Django avec la commande suivante :

```powershell
django-admin startproject nom_projet
```

Ce projet contiendra 5 applications :

• jo : projet principal qui contient la configuration générale et les templates de l'application.

• connexion, panier, sports, ticket : applications pour les différentes pages et fonctionnalités de l'application.

5. Organisation des fichiers de configuration

Dans l'application principale (jo), vous aurez trois fichiers de configuration pour gérer les différents environnements :

• base.py : contient la configuration commune à tous les environnements (local et production).

• local.py : contient les configurations locales comme les identifiants à la base de données, les hôtes autorisés, et le mode débogage activé.

• production.py : contient la configuration spécifique à l'environnement de production, notamment la gestion des fichiers statiques et des middlewares de sécurité.

6. Création des applications et des dossiers

Chaque application contiendra un dossier templates pour les fichiers HTML et un dossier static pour les fichiers CSS, JavaScript, images, etc. Vous devrez également définir les vues et les URL pour chaque fonctionnalité (connexion, panier, etc.).

7. Configuration du fichier manage.py Dans le fichier manage.py, vous devrez définir la variable d'environnement `DJANGO_SETTINGS_MODULE` pour pointer vers le fichier de configuration approprié (local ou production). Ensuite, vous pourrez lancer le serveur local avec la commande suivante :

```powershell
python manage.py runserver
```

8. Fichiers d'environnement et Git

• Ajoutez un fichier .env à la racine du projet pour y stocker toutes les variables d'environnement sensibles comme les clés API, les identifiants de base de données, etc.

• Assurez-vous d'ajouter un fichier .gitignore pour exclure certains fichiers du suivi Git, tels que :

- .env
- venv/
- *.pyc
- pycache/

9. Initialisation du dépôt Git

N'oubliez pas de créer un repository GitHub et d'initialiser un dépôt Git dans votre projet actuel pour versionner et suivre les changements :

```powershell
git init
git add .
git commit -m "Initial commit"
git remote add origin <url_du_repository_github>
git push -u origin master
```

10. Base de données et migrations

Une fois que votre projet est configuré, vous pouvez effectuer les migrations pour créer les tables dans votre base de données :

```powershell
python manage.py makemigrations
python manage.py migrate
```

Architecture

L'architecture de Django repose sur le modèle MTV (Modèle - Template - Vue), qui facilite le développement et la maintenabilité des applications web.

1. Modèle (Model)

Le modèle définit la structure de la base de données et les relations entre les différentes entités. Dans Django, un modèle est généralement une classe Python qui hérite de django.db.models.Model. Chaque modèle correspond à une table en base de données et peut contenir des champs représentant les colonnes.

2. Template

Les templates permettent de générer l'interface utilisateur. Ce sont des fichiers HTML qui intègrent des balises spécifiques à Django pour insérer des variables dynamiques et utiliser des structures de contrôle (if, for, etc.).

3. Vue (View)

La vue est responsable de la logique de traitement des requêtes. Elle :

- Reçoit une requête HTTP.
- Traite cette requête (récupération de données, application de logique métier).
- Retourne une réponse au format HTML, JSON, ou autre.

Dans cette application, nous utilisons la bibliothèque django.http et l'objet JsonResponse pour formater certaines réponses en JSON.

4. Gestion des URL (Routage)

Le système de routage de Django permet d'associer les requêtes HTTP aux vues appropriées, via les fichiers urls.py. Chaque route doit être correctement définie pour éviter des erreurs. Exemple :

```powershell
path('<str:sport_name>/events/', sport_events, name='sport_events')
```

Ici, sport_name est un paramètre de type str (chaîne de caractères). Si une autre valeur est fournie (ex: un entier), une erreur 500 peut être déclenchée. Il est donc important de bien typer les paramètres et de lier chaque route à une vue correspondante.

5. Middlewares

Les middlewares sont des composants intermédiaires qui interceptent les requêtes et les réponses avant qu'elles n'atteignent la vue ou après qu'une vue ait renvoyé une réponse.

Django fournit plusieurs middlewares par défaut :

```powershell
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',  # Sécurise les requêtes/réponses (ex: HTTPS, HSTS)
    'django.contrib.sessions.middleware.SessionMiddleware',  # Gère les sessions utilisateur
    'django.middleware.common.CommonMiddleware',  # Ajoute des fonctionnalités comme la redirection automatique
    'django.middleware.csrf.CsrfViewMiddleware',  # Protège contre les attaques CSRF
    'django.contrib.auth.middleware.AuthenticationMiddleware',  # Gère l'authentification des utilisateurs
    'django.contrib.messages.middleware.MessageMiddleware',  # Permet l'affichage de messages temporaires
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  # Protège contre le clickjacking
]
```

Ces middlewares améliorent la sécurité, la gestion des sessions et l'expérience utilisateur.

6. Tests

Dans chaque application, un dossier tests est présent et contient différents fichiers de test, chacun détaillant une partie spécifique du code à tester. Le nommage du dossier tests, des fichiers et des classes de test est crucial pour que Django puisse identifier et exécuter correctement les tests.

Chaque classe de test hérite de TestCase, permettant d'utiliser les outils de test intégrés à Django. À l'intérieur de chaque classe, une méthode setUp est définie pour créer des objets réutilisables dans chaque test, garantissant ainsi la cohérence des résultats.

Pour exécuter les tests, la commande suivante est utilisée :

```powershell
python manage.py test
```

Une fois tous les tests effectués, j'ai installé Coverage, un outil permettant de mesurer le taux de couverture des tests. Grâce à Coverage, un dossier htmlcov a été généré, fournissant un rapport détaillé sur la partie du code couverte par les tests.

7. Déploiement

L'application est hébergée sur Heroku, avec l'add-on JAWSDB Maria configuré en version Kitefin Shared. Cette version offre un espace de stockage suffisant pour répondre aux besoins de l'application.

1. Configuration du serveur d'application

Pour exécuter l'application sur Heroku, j'ai utilisé Gunicorn, un serveur WSGI performant et adapté aux applications Django. Django étant basé sur un modèle synchrone, et l'application ne nécessitant pas un grand nombre de requêtes asynchrones, cette configuration est suffisante pour assurer de bonnes performances sans nécessiter un serveur ASGI comme Daphne ou Uvicorn.

2. Gestion des fichiers statiques avec WhiteNoise

Heroku ne sert pas automatiquement les fichiers statiques (CSS, JavaScript, images) comme un serveur classique. Pour pallier cela, j'ai utilisé WhiteNoise, une bibliothèque permettant de gérer les fichiers statiques directement via Django sans avoir besoin d'un serveur externe comme Amazon S3 ou Cloudinary.

Voici les étapes suivies pour configurer WhiteNoise : **Installation de WhiteNoise**

```powershell
pip install whitenoise
```

Ajout de WhiteNoise au middleware Django

```powershell
'whitenoise.middleware.WhiteNoiseMiddleware'
```

Collecte des fichiers statiques

J'ai exécuté la commande suivante pour rassembler tous les fichiers statiques dans un dossier unique appelé staticfiles :

```powershell
python manage.py collectstatic
```

Ce dossier contient tous les fichiers CSS, JS et images utilisés par l'application.

Configuration de STATICFILES_STORAGE dans settings.py

```powershell
STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'
```

Cela permet de compresser les fichiers statiques et d'améliorer les performances en production.

Gestion des variables d'environnement

Avant le déploiement, j'ai configuré les variables d'environnement directement sur la plateforme Heroku, incluant :

- DATABASE_URL : Connexion à la base de données MariaDB.
- SECRET_KEY : Sécurisation de l'application.

8. Evolutions possibles

    1. Amélioration de l'envoi des tickets en ligne

Il serait intéressant d'améliorer l'affichage et la mise en page des tickets afin d'offrir une présentation plus soignée et professionnelle aux utilisateurs.

```text
2. Optimisation de l'achat groupé
```

Actuellement, lorsqu'un utilisateur achète plusieurs places (ex. 4 billets), il ne reçoit qu'un seul email de confirmation. Une amélioration possible serait :

- Permettre à l’utilisateur de renseigner le nom et prénom de chaque participant.
- Envoyer un email contenant ces informations ainsi que 4 QR codes distincts, un pour chaque billet acheté.

3. Migration vers ASGI pour une meilleure gestion des requêtes asynchrones

- J’ai initialement tenté de déployer l’application avec ASGI, mais j’ai rencontré des problèmes liés aux sessions utilisateurs que je n’ai pas pu résoudre à ce moment-là.
- Une évolution intéressante serait donc de passer progressivement à ASGI pour bénéficier d’une meilleure gestion des connexions en temps réel.
- L'intégration de WebSockets pourrait également améliorer l’expérience utilisateur, notamment pour des mises à jour en direct sans recharger la page.

4. Mise en place d’une véritable stratégie CI/CD

- Mon objectif initial était d'implémenter un processus CI/CD (Intégration et Déploiement Continus), mais le manque de temps et d’expertise m’a freiné.
- À l’avenir, il serait pertinent d'intégrer des outils comme GitHub Actions, GitLab CI/CD ou Jenkins pour automatiser les tests et le déploiement, garantissant ainsi une meilleure stabilité et un gain de temps.