# TD Gestion de projet - 02

## 1. Développement

### Introduction

À l'aide du cahier des charges précédemment défini, nous allons développer l'application en simulant des sprints et en utilisant un framework Git. Cette approche nous permettra de structurer le développement en cycles itératifs, chacun se concentrant sur des fonctionnalités spécifiques. En utilisant Git, nous assurerons un suivi rigoureux des versions, facilitant ainsi la collaboration entre les membres de l'équipe et garantissant une intégration continue des modifications.

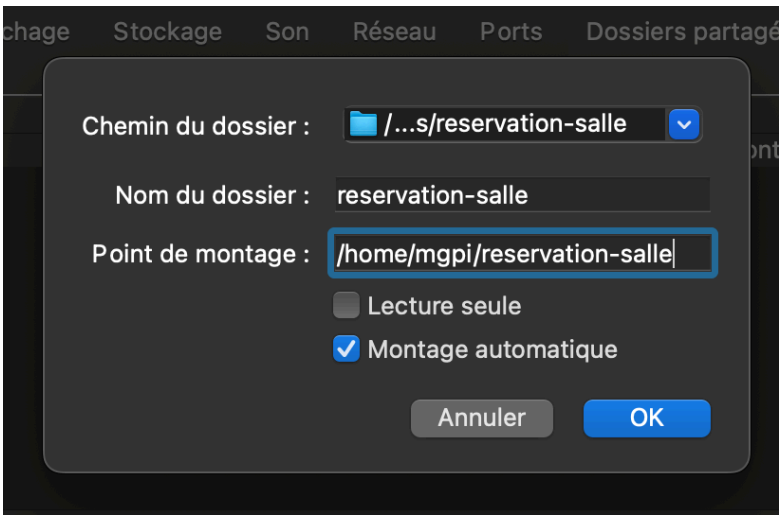
### Objectif

1. Développer une application répondant aux besoins défini dans le cahier des charges
2. Appliquer le workflow git flow lors des développements
3. Ecrire les tests unitaires associés
4. Documenter le code de l’application

### Étapes de développement

#### Prérequis

1. Télécharger l’image VirtualBox MGPI sur nuage (lien) dans le répertoire /tmp et décompressez le.
2. Ouvrir VirtualBox et ajouter une nouvelle machine virtuelle et sélectionner le fichier MGPI.vbox
3. Ouvrir la configuration et vérifier que les ports sont bien exposés (ssh / django)
4. Configurer la machine virtuelle dans l’onglet dossier partagés monter le répertoire de l’application qui va être développée




5. Démarrer la machine virtuelle
6. Ouvrir un terminal dans votre host et se connecter à la machine virtuelle (mdp: mgpi)

```
ssh -p 3022 mgpi@127.0.0.1
```

7. Ajouter les dépendances

```
poetry add django
poetry add --group dev pytest pytest-django pytest-html sphinx
sphinx-rtd-theme
```

## Sprint 1

1. Créer une branche de feature en suivant le principe de git-flow donc en partant de la branche develop

```
git checkout develop
git checkout -b features/001-initialize-tests
```

2. Ajouter les fichiers suivants (présents sur moodle TD 02 - Ressources > tests.zip) dans le répertoire tests

- test_models.py
- test_reservation_views.py
- test_salle_views.py

3. Faire un commit
4. Pousser la branche et le commit sur le repository distant
5. Faire une review du code et merger la branche de feature dans la branche **develop**
6. Créer une branche de feature en suivant le principe de git-flow et implémenter la création des entitées afin que les tests puissent passer

```
git checkout develop
```

```
git checkout -b features/002-entities
```

a. Les utilisateurs seront gérés par le mécanisme d'authentification de django, donc il n'y a pas besoin d'en créer

b. Les modèles peuvent être construit de cette manière dans le fichier models.py:

```
from django.db import models
from django.contrib.auth.models import User

class Etudiant(models.Model):
    nom = models.CharField(max_length=100)
    numero = models.IntegerField()
    STATUT_CHOICES = [
        ('valide', 'Validé'),
        ('rattrapge', 'Rattrapage'),
        ('echoue', 'Echoué'),
    ]
    statut = models.CharField(max_length=20,
choices=STATUT_CHOICES)

class Compte(models.Model):
    etudiant = models.ForeignKey(Etudiant,
on_delete=models.CASCADE)
    compte = models.ForeignKey(User, on_delete=models.CASCADE)
    date_debut = models.DateTimeField()
    date_fin = models.DateTimeField()

    def __str__(self):
        return f"Compte de {self.etudiant.nom} par
{self.compte.username} du {self.date_debut} au {self.date_fin}"
```

7. Une fois les entitées créées générer les migrations de la base de données

```
poetry run python reservation_salle/manage.py makemigrations
```

8. Exécuter le migrations

```
poetry run python reservation_salle/manage.py migrate
```

9. Commit, push, review, merge

10. Créer une branche de feature en suivant le principe de git-flow

```
git checkout -b features/003-create-permissions
```

11. Ajouter un fichier **000X_create_office_manager_group.py** dans le dossier migrations (remplacer le X en incrémentant le nombre de 1 et remplacer le nom dans dependencies par le fichier précédent)

```
from django.contrib.auth.models import Group, Permission, User
```

```
from django.contrib.contenttypes.models import ContentType
from django.db import migrations


def create_office_manager_group(apps, schema_editor) -> None:
    Salle = apps.get_model('reservation_salle', 'Salle')
    Reservation = apps.get_model('reservation_salle',
'Reservation')

    content_type_salle = ContentType.objects.get_for_model(Salle)
    content_type_reservation =
ContentType.objects.get_for_model(Reservation)

    can_ajouter_une_salle =
Permission.objects.create(codename='add_salle',
                                                    name='Peut
ajouter une salle',

content_type=content_type_salle)
    can_supprimer_une_salle =
Permission.objects.create(codename='delete_salle',
                                                      name='Peut
supprimer une salle',

content_type=content_type_salle)
    can_changer_une_salle =
Permission.objects.create(codename='change_salle',
                                                    name='Peut
modifier une salle',

content_type=content_type_salle)
    can_annuler_toutes_reservation =
Permission.objects.create(codename='delete_all_reservation',

name='Peut annuler toutes les réservations',

content_type=content_type_reservation)

    office_manager_group =
Group.objects.create(name='office_manager')
    office_manager_group.permissions.set([can_ajouter_une_salle,
can_supprimer_une_salle, can_changer_une_salle,
can_annuler_toutes_reservation])

    office_manager_group.save()
    user = User.objects.create_user(username='office',
password='12345')
    user.groups.add(office_manager_group)
```

```
    user.save()

class Migration(migrations.Migration):
    dependencies = [("reservation_salle", "0001_initial")]

    operations =
[migrations.RunPython(create_office_manager_group)]
```

12. Ajouter un fichier **000X_create_collaborateur_group.py** dans le dossier migrations (remplacer le X en incrémentant le nombre de 1 et remplacer le nom dans dependencies par le fichier précédent)

```
from django.contrib.auth.backends import UserModel
from django.contrib.auth.models import Group, Permission, User
from django.contrib.contenttypes.models import ContentType
from django.db import migrations


def create_collaborateur_group(apps, schema_editor) -> None:
    """
    Fonction utilisée pour créer le groupe 'Collaborateur' et lui
attribuer des permissions spécifiques.

    Cette fonction crée un groupe nommé 'Collaborateur' et lui
attribue deux permissions :
    - 'can_lister_salles' : Permet de lister les salles.
    - 'can_faire_une_reservation' : Permet de réserver une salle.
    - 'can_annuler_reservation' : Permet d'annuler une réservation.

    :param apps: L'objet de registre des applications Django.
    :param schema_editor: L'objet d'édition de schéma Django.
    :return: None
    """
    Salle = apps.get_model('reservation_salle', 'Salle')
    Reservation = apps.get_model('reservation_salle',
'Reservation')

    content_type_salle = ContentType.objects.get_for_model(Salle)
    content_type_reservation =
ContentType.objects.get_for_model(Reservation)

    can_lister_salles =
Permission.objects.create(codename='view_salles',
                                                name='Peut lister
les salles',

content_type=content_type_salle)
```

```
    can_lister_reservations =
Permission.objects.create(codename='view_reservation',
                                                        name='Peut
lister les réservations',

content_type=content_type_reservation)
    can_faire_une_reservation =
Permission.objects.create(codename='create_reservation',

name='Peut réserver une salle',

content_type=content_type_reservation)
    can_annuler_reservation =
Permission.objects.create(codename='delete_reservation',
                                                        name='Peut
annuler une réservation',

content_type=content_type_reservation)

    collaborateur_group =
Group.objects.create(name='collaborateur')
    collaborateur_group.permissions.set(
        [can_lister_salles, can_lister_reservations,
can_faire_une_reservation, can_annuler_reservation])

    collaborateur_group.save()
    collaborateur =
User.objects.create_user(username='collaborateur',
password='12345')
    collaborateur.groups.add(collaborateur_group)
    collaborateur.save()
    office_manager = User.objects.get(username='office')
    office_manager.groups.add(collaborateur_group)
    office_manager.save()


class Migration(migrations.Migration):
    dependencies = [("reservation_salle",
"0002_create_office_manager_group")]

    operations = [migrations.RunPython(create_collaborateur_group)]
```

13. Commit, push, review, merge

14. Créer une nouvelle branche pour ajouter les fichiers de template dans le répertoire **template** (Moodle TD 02 - Ressources > template.zip) et ajouter les url dans le fichier **urls.py**

```
from django.contrib import admin
```

```
from django.urls import path, include

from .views import salle_views, reservation_views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('django.contrib.auth.urls')),
    path('creer_salle/', salle_views.creer_salle,
name='creer_salle'),
    path('liste_salles/', salle_views.liste_salles,
name='liste_salles'),
    path('supprimer_salle/<int:salle_id>/',
salle_views.supprimer_salle, name='supprimer_salle'),
    path('modifier_salle/<int:salle_id>/',
salle_views.modifier_salle, name='modifier_salle'),
    path('liste_reservations/',
reservation_views.liste_reservations, name='liste_reservations'),
    path('supprimer_reservation/<int:reservation_id>/',
reservation_views.supprimer_reservation,
        name='supprimer_reservation'),
    path('creer_reservation/<int:salle_id>/',
reservation_views.creer_reservation, name='creer_reservation'),
    path('liste_reservations_salle/<int:salle_id>/',
reservation_views.liste_reservations_salle,
        name='liste_reservations_salle'),
    path('', salle_views.liste_salles, name='home'),
]
```

15. Commit, push, review, merge

## Première release

1. Changer le numéro de version de l'application pour la passer en 1.0.0 dans le fichier pyproject.toml
2. Merger la branche develop dans la branche master
3. Créer un tag 1.0.0 à partir de la branche master

## Sprint 2

1. Retourner sur la branche develop et créer une nouvelle branche pour implémenter les forms et views
2. Créer un dossier forms, ajouter un fichier __init__.py et copier le fichier salle_forms.py reservation_forms.py

```
from django import forms
from ..models import Salle
```

```python
class SalleForm(forms.ModelForm):
    """
    Formulaire Django pour la création et la modification d'une
salle.

    Ce formulaire est basé sur le modèle Salle et inclut les champs
    'nom', 'capacite', et 'statut'. Il permet de valider et de
sauvegarder
    les informations de la salle de manière structurée.

    Attributes:
        Meta: Classe interne qui définit les métadonnées du
formulaire.

    Example:
        form = SalleForm(data=request.POST)
        if form.is_valid():
            form.save()
    """
    class Meta:
        """
        Métadonnées du formulaire SalleForm.

        Attributes:
            model (Salle): Le modèle associé au formulaire.
            fields (list): La liste des champs du modèle à inclure
dans le formulaire.
        """
        model = Salle
        fields = ['nom', 'capacite', 'statut']
```

```python
from django import forms
from ..models import Reservation


class ReservationForm(forms.ModelForm):
    """
    Formulaire Django pour la création et la modification d'une
réservation.

    Ce formulaire est basé sur le modèle Reservation et inclut les
champs
    'date_debut' et 'date_fin'. Il permet de valider et de
sauvegarder les
    informations de réservation de manière structurée.

    Attributes:
```

```
        Meta: Classe interne qui définit les métadonnées du
formulaire.

    Example:
        form = ReservationForm(data=request.POST)
        if form.is_valid():
            form.save()
    """
    class Meta:
        """
        Métadonnées du formulaire ReservationForm.

        Attributes:
            model (Reservation): Le modèle associé au formulaire.
            fields (list): La liste des champs du modèle à inclure
dans le formulaire.
        """
        model = Reservation
        fields = ['date_debut', 'date_fin']
```

3. Créer un dossier views, ajouter un fichier __init__.py et copier le fichier salle_views.py et écrire le fichier reservation_views.py

```
from django.contrib.auth.decorators import permission_required,
login_required
from django.shortcuts import render, redirect, get_object_or_404
from ..forms.salle_forms import SalleForm
from ..models import Salle

@login_required
@permission_required('reservation_salle.add_salle',
raise_exception=True)
def creer_salle(request):
    """
    Crée une nouvelle salle.

    Cette vue nécessite que l'utilisateur soit connecté et ait la
permission
    'add_salle' pour créer une nouvelle salle.

    Args:
        request (HttpRequest): L'objet de requête HTTP.

    Returns:
        HttpResponse: La réponse HTTP rendant le template
'creer_salle.html'
        avec le formulaire de création de salle ou redirigeant vers
la liste des salles.
    """
```

```
    if request.method == 'POST':
        form = SalleForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('liste_salles')
    else:
        form = SalleForm()
    return render(request, 'salle/creer_salle.html', {'form':
form})

@login_required
@permission_required('reservation_salle.change_salle',
raise_exception=True)
def modifier_salle(request, salle_id):
    """
    Modifie une salle existante.

    Cette vue nécessite que l'utilisateur soit connecté et ait la
permission
    'change_salle' pour modifier une salle.

    Args:
        request (HttpRequest): L'objet de requête HTTP.
        salle_id (int): L'identifiant de la salle à modifier.

    Returns:
        HttpResponse: La réponse HTTP rendant le template
'modifier_salle.html'
        avec le formulaire de modification de salle ou redirigeant
vers la liste des salles.
    """
    salle = get_object_or_404(Salle, id=salle_id)
    if request.method == 'POST':
        form = SalleForm(request.POST, instance=salle)
        if form.is_valid():
            form.save()
            return redirect('liste_salles')
    else:
        form = SalleForm(instance=salle)
    return render(request, 'salle/modifier_salle.html', {'form':
form, 'salle': salle})

@login_required
@permission_required('reservation_salle.view_salles',
raise_exception=True)
def liste_salles(request):
    """
    Affiche la liste de toutes les salles.
```

```
    Cette vue nécessite que l'utilisateur soit connecté et ait la
permission
    'view_salles' pour accéder à la liste des salles.

    Args:
        request (HttpRequest): L'objet de requête HTTP.

    Returns:
        HttpResponse: La réponse HTTP rendant le template
'liste_salles.html'
        avec la liste des salles.
    """
    salles = Salle.objects.all()
    return render(request, 'salle/liste_salles.html', {'salles':
salles})

@login_required
@permission_required('reservation_salle.delete_salle',
raise_exception=True)
def supprimer_salle(request, salle_id):
    """
    Supprime une salle spécifique.

    Cette vue nécessite que l'utilisateur soit connecté et ait la
permission
    'delete_salle' pour supprimer une salle.

    Args:
        request (HttpRequest): L'objet de requête HTTP.
        salle_id (int): L'identifiant de la salle à supprimer.

    Returns:
        HttpResponse: La réponse HTTP rendant le template
'supprimer_salle.html'
        ou redirigeant vers la liste des salles.
    """
    salle = get_object_or_404(Salle, id=salle_id)
    if request.method == 'POST':
        salle.delete()
        return redirect('liste_salles')
    return render(request, 'salle/supprimer_salle.html', {'salle':
salle})
```

4. Commit, push, review, merge
5. Effectuer la release de la nouvelle version

# 2. Documentation

## Introduction

Ce guide vous accompagnera dans la configuration et la génération de la documentation pour votre projet à l'aide de Sphinx. Sphinx est un générateur de documentation écrit en Python, conçu pour rendre la documentation simple et belle. Nous allons configurer Sphinx, ajouter les modules à documenter, et générer la documentation au format HTML.

## Objectif

1. Configurer Sphinx
2. Ajouter et indiquer les modules à documenter
3. Générer la documentation au format HTML

### Configuration

1. Initialiser la documentation avec sphinx et configurer basiquement les

```
poetry run sphinx-quickstart docs
```

2. Modifier le fichier conf.py dans le dossier docs
3. Générer les fichiers

```
poetry run sphinx-apidoc -F -o docs reservation_salle
```

4. Modifier le fichier conf.py dans le dossier docs pour ajouter ces informations:

```
import os
import sys
import django
sys.path.insert(0, os.path.abspath('../reservation_salle'))
os.environ['DJANGO_SETTINGS_MODULE'] =
'reservation_salle.settings'
django.setup()
extensions = [
    'sphinx.ext.autodoc',
    'sphinx.ext.viewcode',
    'sphinx.ext.napoleon',
]
html_theme = 'sphinx_rtd_theme'
```

### Génération

1. Se rendre dans le répertoire de la documentation

```
cd docs
```

2. Générer la documentation

```
make html
```

# 3. Containerisation de l’application

## Introduction

Ce guide vous accompagnera dans la création et la configuration d'un environnement Docker pour un projet Django utilisant Poetry pour la gestion des dépendances. Nous allons créer un fichier Dockerfile à la racine du projet, configurer les étapes nécessaires pour installer les dépendances, appliquer les migrations de la base de données, et démarrer le serveur de développement Django. Enfin, nous construirons et démarrerons le conteneur Docker depuis une machine virtuelle (VM).

## Objectifs

1. Créer un fichier Dockerfile
2. Configurer le Dockerfile pour utiliser Ubuntu comme image de base, installer Poetry, et configurer l'environnement de travail.
3. Copier les fichiers du projet dans le conteneur et installer les dépendances du projet.
4. Appliquer les migrations de la base de données pour le projet Django.
5. Définir la commande par défaut pour démarrer le serveur de développement Django.
6. Construire l’image Docker.
7. Démarrer le conteneur Docker et exposer le serveur de développement sur le port 8000.

## Étapes

1. Créer un fichier Dockerfile à la racine du projet et renseigner les informations suivantes:

```
# Utiliser l'image Ubuntu la plus récente comme base
FROM ubuntu:latest

# Mettre à jour les paquets existants, installer les mises à jour
et installer Poetry
RUN apt update && apt upgrade -y && apt install -y python3-poetry
&& apt-get autoremove

# Définir le répertoire de travail à l'intérieur du conteneur
```

```
WORKDIR /app

# Copier tous les fichiers du répertoire courant sur la machine
hôte vers le répertoire /app dans le conteneur
COPY . /app

# Installer les dépendances du projet définies dans le fichier
pyproject.toml
RUN poetry install

# Appliquer les migrations de la base de données pour le projet
Django
RUN poetry run reservation_salle/manage.py migrate

# Définir la commande par défaut à exécuter lorsque le conteneur
démarre
CMD ["poetry", "run", "reservation_salle/manage.py", "runserver",
"0.0.0.0:8000"]
```

2. Construire le container depuis la VM

```
docker build -t reservation_salle .
```

3. Démarrer le container

```
docker run --rm -it -p8000:8000 reservation_salle
```