



Université Hassan II de Casablanca
Ecole Nationale Supérieure d'Arts et Métiers Casablanca

RAPPORT DU MINI-PROJET
Back-end DJANGO
Sujet : Compte bancaire

Réalisé par :

Abdelmoughit BOUDDINE

Encadré par :

M.AYAD Habib

Année universitaire : 2018/2019

PRÉREQUIS :

1. Introduction au Python

Le Python est un langage de programmation orientée objet similaire au Perl et Ruby. Sa force majeure est qu'il propose des milliers de modules qui vous feront gagner un temps précieux durant le développement de vos applications.

Pour ce qui est de l'apprentissage, le Python est très intuitif grâce à son typage dynamique fort et à sa communauté présente sur les forums. Entre autres, des sites sont disponibles pour apprendre rapidement ce langage avec des thématiques comme l'Open Data : dataquest.io.

2. Méthode REST

Pour rendre accessible des données via un site, il existe les méthodes dites REST (Representational State Transfer). Il s'agit d'un ensemble de conventions et de bonnes pratiques à respecter et non d'une technologie entière.

L'information de base, dans une architecture REST, est appelée ressource. Toute information qui peut être nommée est une ressource : la description d'un bâtiment, la liste des arrêts de bus ou n'importe quel concept. Dans un système hypermédia, une ressource est tout ce qui peut être référencé par un lien.

L'interface entre les composants est simple et uniforme. En HTTP, cette interface est implantée par les verbes GET, PUT, POST, DELETE, . . . qui permettent aux composants de manipuler les ressources de manière simple. Par exemple quand un agent voudra récupérer la liste des arrêts de bus depuis l'application, il passera par la méthode GET qui lui retournera les ressources voulues.

3. Django

Django est un Framework web Python de haut niveau qui encourage le développement rapide et propre. Gratuit et open source, Django vous permet d'éviter de réinventer la roue grâce à toutes les librairies disponibles en Python, mais aussi de tout ce que ce Framework offre dès son installation.

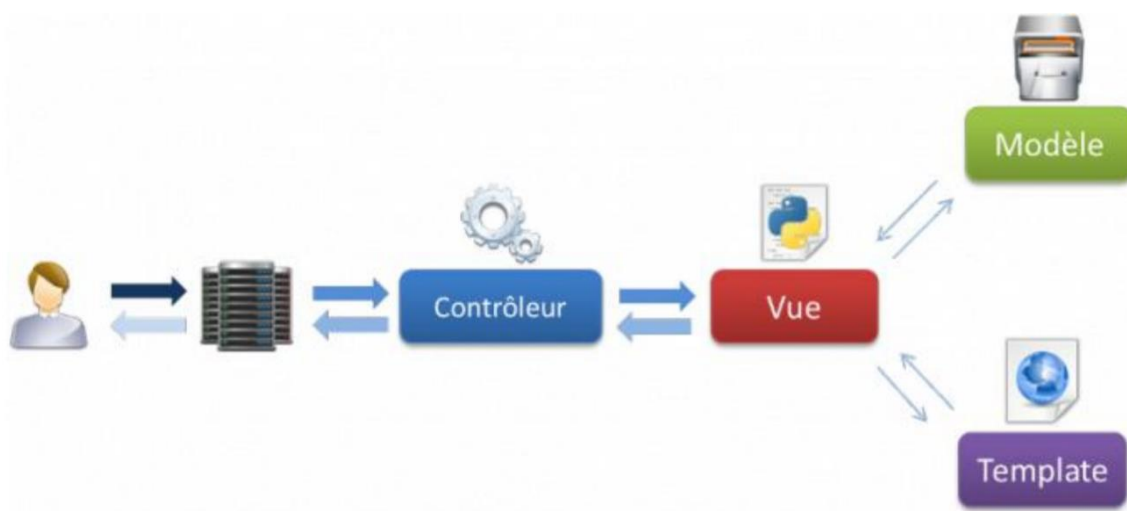
Le fonctionnement d'une application Django se divise en 2 parties.

La première section a pour but de préparer l'étape entre l'utilisateur et l'application en elle-même. Cette section gère la relation entre les bases de données et les applications Django, mais aussi s'occupe du routage via des règles URL.

La seconde section est l'application. Organisée en modèles, vues et templates, l'application se trouve au cœur du projet Django. Utilisé par la NASA ou le Washington Post, Django prouve sa fiabilité et sa stabilité à travers ce type d'organisme.

4. Le modèle MVT

Django se base sur le modèle MVT, légèrement différent du modèle MVC, le Framework gère lui-même le contrôleur et laisse place au Template.



Lors d'une requête venant de l'utilisateur, le Framework Django gère lui-même, via les règles de routage défini par le développeur, de charger la bonne vue correspondante au résultat voulu.

Une Template est un fichier HTML qui sera récupéré par la vue pour être envoyé à l'utilisateur, mais entre cette étape, Django va exécuter la Template comme si c'était un fichier de code.

Inclus dans les templates, le Framework propose l'utilisation des structures conditionnelles, des boucles, des variables... afin d'avoir une grande liberté de développement.

5. Rest Framework

Le Framework Django Rest offre la possibilité de mettre en place une API qui a pour but d'autoriser l'accès aux ressources d'une ou plusieurs bases de données. Utiliser comme une application, il faudra lui d'écrire les modèles, vues et routage pour être fonctionnel.

Partie théorique :

- Énoncé:

Une banque désire posséder un SGBD pour suivre ses *clients*. Elle désire ainsi stocker les coordonnées de chaque client (nom, prénom adresse), et les *comptes* dont elle dispose ainsi que leur solde (sachant par ailleurs que certains comptes ont plusieurs bénéficiaires). On stockera également les *opérations* relatives à ces comptes (retrait et dépôt, avec leur date et le montant).

- Analyse :

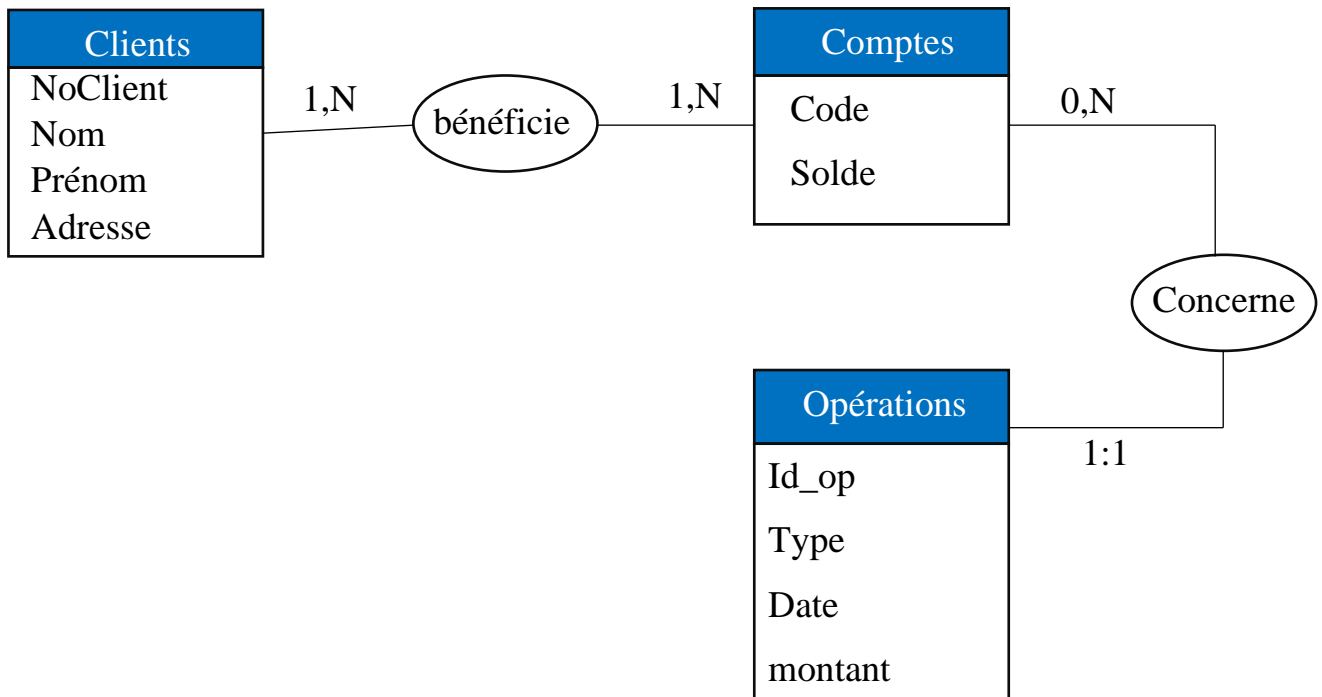
✓ Entités : Clients – Comptes – Opérations

✓ Associations :

❖ Clients – Comptes : Association binaire de cardinalité (1,N)-(1,N)

❖ Comptes – Opérations : Association binaire de cardinalité (0,N)-(1,1)

➤ Conception en MCD :



Partie pratique :

➤ Étapes initiales :

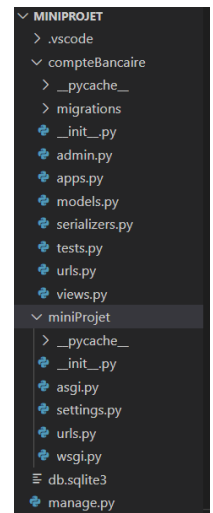
- ✚ Créer le dossier mini projet : en utilisant la commande « django-admin startproject 'nom_projet' »
`D:\Djangoprojects>django-admin startproject miniProjet`

- ✚ Créer l'app banque : en utilisant la commande « python manage.py startapp 'nom_app' » dans le terminal du Visual Studio Code

(ensam) D:\Djangoprojects\miniProjet>python manage.py startapp compteBancaire

N.B : Pour vérifier que l'application est bien créée, check la barre « Explorer » sous le dossier 'MINIPROJET', vous trouverez le dossier « compteBancaire » est présent.

- Installer le Framework REST : en utilisant la commande « conda install rest_framework »



➤ Développement d'une application :

- **Réglages « miniProjet » - settings.py :** C'est un module Python tout à fait normal, avec des variables de module qui représentent des réglages de Django.
- ✓ Le 1e réglage important à modifier est « **INSTALLED APPS** ». Cette variable contient le nom des applications Django qui sont actives dans cette instance de Django. Les applications peuvent être utilisées dans des projets différents, et vous pouvez emballer et distribuer les vôtres pour que d'autres les utilisent dans leurs projets.

Par défaut, INSTALLED_APPS contient les applications suivantes, qui sont toutes contenues dans Django :

- **django.contrib.admin** - Le site d'administration. Vous l'utiliserez très bientôt.
- **django.contrib.auth** - Un système d'authentification.
- **django.contrib.contenttypes** - Une structure pour les types de contenu (content types).
- **django.contrib.sessions** - Un cadre pour les sessions.
- **django.contrib.messages** - Un cadre pour l'envoi de messages.
- **django.contrib.staticfiles** - Une structure pour la prise en charge des fichiers statiques.

Dans notre projet, nous allons ajouter les applications 'rest_framework' et 'compteBancaire.apps.CompteBancaireConfig'.

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'rest_framework',  
    'compteBancaire.apps.ComptebancaireConfig',  
]
```

- ✓ Le 2^e réglage à modifier est **« DATABASES »** : Un dictionnaire contenant les réglages de toutes les bases de données à utiliser avec Django. C'est un dictionnaire imbriqué dont les contenus font correspondre l'alias de base de données avec un dictionnaire contenant les options de chacune des bases de données.

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.mysql',  
        'NAME': 'bank',  
        'USER': 'root',  
        'PASSWORD': '',  
        'HOST': 'localhost',  
        'PORT': '3306',  
        'OPTIONS': {  
            'init_command': "SET sql_mode='STRICT_TRANS_TABLES'",  
        },  
    },  
}
```

- **Routage du « miniProjet » - urls.py** : Le fichier url.py a pour but de gérer le routage de l'application. Pour faire simple, lorsqu'un client appelle une ressource via un URI, ce fichier permet de faire la liaison entre l'extension de l'adresse et la vue adéquate.

```
from django.contrib import admin  
from django.urls import *  
from compteBancaire.views import *  
  
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('', include('compteBancaire.urls'))  
]
```

- **Modèle – models.py** : Le modèle représente la structure de la table. Lié à une base de données, le modèle doit permettre à Django d'interpréter la structure des ressources.

1) **Table « Accounts »** :

```
class Account(models.Model):
    class Meta:
        db_table = 'accounts'

    code = models.AutoField(primary_key=True, db_column='code')
    balance = models.FloatField(default=0, db_column='solde')
    clients = models.ManyToManyField(Client, related_name='comptes')

    def __str__(self):
        return "{}".format(self.code)
```

+ **db_table** : nom de la table des comptes dans la base de données « Bank ».

+ **models.AutoField(primary_key=True, db_column='code')** : le champ « code » est une clé primaire, auto incrémentée, dont son nom dans la base est 'code'.

+ **models.ManyToManyField('Client')** : Pour définir une relation plusieurs-à-plusieurs entre la classe « Account » et la classe « Client ».

+ **__str__()** : method just tells Django what to print when it needs to print out an instance of any model.

2) **Table « Operations »** :

```
class Operations(models.Model):
    class Meta:
        db_table = 'operations'

    DEBIT = 'D'
    CREDIT = 'C'
    TYPE = [(DEBIT, 'D'), (CREDIT, 'C'),]
    id_op = models.AutoField(primary_key=True, db_column='id')
    type = models.CharField(max_length=1, choices=TYPE, default=DEBIT)
    date = models.DateField(auto_now=True)
    amount = models.FloatField(default=0)
    account = models.ForeignKey(Account, default=0, on_delete=models.CASCADE, related_name='operations')

    def __str__(self):
        return "{} - {}".format(self.id_op, self.type)
```

+ **choices=TYPE** : énumérer les valeurs du type des opérations bancaires - soit D=Débit ou C=Crédit -

+ **models.DateField(auto_now=True)** : remplir le champ automatiquement avec le temps d'insertion d'une donnée.

+ **models.ForeignKey(Account, default=0, on_delete=models.CASCADE)** : Pour définir une relation plusieurs-à-un entre la classe « Operations » et « Account », utilisez ForeignKey qui adopte un comportement quand un objet référencé est supprimé (e.g : quand on supprime un compte, il sera supprimé aussi dans le champ « accounts » dans la table 'Operations').

3) **Table « Client »** :

```
class Client(models.Model):
    class Meta:
        db_table = 'clients'

    NoClient = models.CharField(primary_key=True, max_length=8, db_column='CIN')
    lastname = models.CharField(max_length=100)
    firstName = models.CharField(max_length=100, db_column='prenom')
    address = models.TextField(null=True)

    def __str__(self):
        return "{} - {}".format(self.NoClient, self.lastname)
```

- ❖ Créer les tables des modèles dans la base de données en utilisant les deux commandes : « python manage.py makemigrations » et « python manage.py migrate ».
- ❖ Ajouter la représentation des modèles dans l'interface d'administration via le fichier « admin.py » :

```
from django.contrib import admin
from .models import Account, Operations, Client

# Register your models here.
admin.site.register(Account)
admin.site.register(Operations)
admin.site.register(Client)
```

- **Sérialisation – serializers.py** : La sérialisation permet aux données, complexes telles que les modèles, d'être convertis en type de données natives pour Python. Ils peuvent ensuite être facilement rendus dans les formats JSON ou XML. Dans le cas présent, la sérialisation permet aussi de sélectionner les champs à afficher pour limiter la vue des données.

```
from rest_framework import serializers
from .models import Account, Client, Operations

class OperationsSerializer(serializers.ModelSerializer):
    class Meta:
        model = Operations
        fields = '__all__'
        extra_kwargs = {'id_op': {'required': False}}
        depth = 1

class AccountSerializer(serializers.ModelSerializer):
    operations = serializers.SlugRelatedField(many=True, queryset=Operations.objects.all(),
        slug_field='id_op')
    class Meta:
        model = Account
        fields = '__all__'
        extra_kwargs = {'code': {'required': False}}

class ClientSerializer(serializers.ModelSerializer):
    accounts = AccountSerializer(many=True)
    class Meta:
        model = Client
        fields = '__all__'
        depth = 1
```

+ **depth=1** : le nombre indique la profondeur que vous voulez pour aller dans les relations, pour notre cas avec la valeur 1 il suffit d'obtenir ce résultat.

+ **extra_kwargs** :
{'champ' : {'required' : False}} : champ optionnel.

- + **serializers.SlugRelatedField(many=True, slug_field='id_op')** : utilisé pour représenter la cible de la relation en utilisant un champ sur la cible « id_op » qui identifie de manière unique l'instance de « Operations » et « many=True » vu qu'il est appliqué à une relation à plusieurs.

- **Vue – views.py :** La vue, dans son ensemble, affiche les données depuis une ressource. L'utilisation de "viewset" permet de passer par un Template du Framework et nécessitera que l'indication du modèle correspondant aux ressources de la vue.

- **GenericAPIView**

Cette classe étend la APIView classe du framework REST , en ajoutant le comportement couramment requis pour les vues de liste et de détail standard.

Chacune des vues génériques concrètes fournies est construite en combinant GenericAPIView, avec une ou plusieurs classes **mixin**.

- **Les attributs :**

Paramètres de base :

Les attributs suivants contrôlent le comportement de base de la vue.

1. **queryset**- Le jeu de requêtes à utiliser pour renvoyer des objets à partir de cette vue. En règle générale, vous devez définir cet attribut ou remplacer la **get_queryset()** méthode. Si vous remplacez une méthode de vue, il est important que vous appeliez **get_queryset()** au lieu d'accéder directement à cette propriété, car elle queryset sera évaluée une fois, et ces résultats seront mis en cache pour toutes les demandes suivantes.
2. **serializer_class**- La classe de sérialiseur à utiliser pour valider et désérialiser l'entrée et pour sérialiser la sortie. En règle générale, vous devez définir cet attribut ou remplacer la **get_serializer_class()** méthode.
3. **lookup_field**- Le champ de modèle qui doit être utilisé pour effectuer la recherche d'objet d'instances de modèle individuelles. La valeur par défaut est 'pk'. Notez que lors de l'utilisation des API , vous aurez besoin des liens hypertexte pour faire en sorte que les vues de l' API et les classes sérialiseur définissent les champs de recherche si vous avez besoin d'utiliser une valeur personnalisée.

- **View des comptes :** Définir les fonctions de la recherche « get », la création « post », la modification « put » et la suppression « destroy » -définie implicitement grâce à generics.DestroyAPIView- des comptes.

```
# Create your views here.
class AccountGenericApi(mixins.ListModelMixin,
mixins.RetrieveModelMixin,mixins.CreateModelMixin,
mixins.UpdateModelMixin,generics.DestroyAPIView):
    serializer_class = AccountSerializer
    queryset = Account.objects.all()
    lookup_field = 'code'
    permission_classes = [IsAuthenticated]

    def get(self,request,code=None):
        if code:
            return self.retrieve(request)
        else:
            return self.list(request)

    def post(self,request):
        return self.create(request)

    def put(self,request,code=None):
        return self.update(request)
```

Remarque : « permission_classes » est pour la sécurisation /Expliquée dans un paragraphe prochain/

- Création des mixins personnalisés : Nous créons une classe mixin si nous devons rechercher des objets en fonction de plusieurs champs dans la configuration d'URL au lieu du filtrage d'un champ unique par défaut.

```
class MultipleFieldLookupMixin:

    def get_object(self):
        queryset = self.get_queryset()          # Get the base queryset
        queryset = self.filter_queryset(queryset) # Apply any filter backends
        filter = {}
        for field in self.lookup_fields:
            if self.kwargs[field]: # Ignore empty fields.
                filter[field] = self.kwargs[field]
        obj = get_object_or_404(queryset, **filter) # Lookup the object
        self.check_object_permissions(self.request, obj)
        return obj
```

- View des clients : Appliquer le mixin personnalisé sur les clients en utilisant le nom et prénom comme lookup_fields.

```
class ClientGenericApi(mixins.ListModelMixin,
mixins.RetrieveModelMixin,mixins.CreateModelMixin,
mixins.UpdateModelMixin,generics.DestroyAPIView,MultipleFieldLookupMixin):
    serializer_class = ClientSerializer
    queryset = Client.objects.all()
    lookup_field = 'NoClient'
    lookup_fields = ['lastName','firstName']
    permission_classes = [IsAuthenticated]

    def get(self,request,NoClient=None):
        if NoClient:
            return self.retrieve(request)
        elif lastName!=None or firstName!=None:
            obj = self.get_object()
            serializer = self.get_serializer(instance=obj)
            return Response(serializer.data)
        else:
            return self.list(request)

    def post(self,request):
        return self.create(request)

    def put(self,request,NoClient=None):
        return self.update(request)
```

- [View des opérations](#) : Application du mixin personnalisé sur les opérations en utilisant le type et la date d'opération comme étant lookup_fields.

```
class OperationsGenericApi(generics.GenericAPIView, mixins.ListModelMixin,
mixins.RetrieveModelMixin, mixins.CreateModelMixin, MultipleFieldLookupMixin):
    serializer_class = OperationsSerializer
    queryset = Operations.objects.all()
    lookup_field = 'id_op'
    lookup_fields = ['type', 'date']
    permission_classes = [IsAuthenticated]

    def get(self, request, id_op=None, type=None, date=None):
        if id_op:
            return self.retrieve(request)
        elif type!=None or date!=None:
            obj = self.get_object()
            serializer = self.get_serializer(instance=obj)
            return Response(serializer.data)
        else:
            return self.list(request)

    def creditor(self, request):
        code = request.data['account']
        account = Account.objects.get(code = int(code))
        account.balance += float(request.data['amount'])
        account.save()
```

+ **créditer()** : ajouter un montant au solde.

```
def debiter(self, request):
    code = request.data['account']
    account = Account.objects.get(code = int(code))
    if(account.balance >= request.data['amount']):
        account.balance -= float(request.data['amount'])
    else:
        return Response('Opération impossible !!')
    account.save()

def post(self, request):
    if request.data['type'] == 'D':
        self.debiter(request)
    elif request.data['type'] == 'C':
        self.creditor(request)
    return self.create(request)
```

+ **débiter()** : retirer de l'argent d'un compte tout en vérifiant si le solde est suffisant.

- Sécurisation de l'application :

JWT est une norme ouverte pour le transfert de données en toute sécurité entre deux parties. Il est utilisé avec les systèmes d'authentification pour effectuer des requêtes authentifiées. Il se compose principalement de header, payload et signature. JWT est un mécanisme d'authentification sans état, c'est-à-dire qu'il maintient les sessions du côté client lui-même au lieu de les stocker sur le serveur.

- Comment fonctionne JWT?

- ✓ **Le JWT est juste un jeton d'autorisation qui doit être inclus dans toutes les demandes.**
- ✓ Le JWT est acquis en échangeant un nom d'utilisateur + un mot de passe pour un **jeton d'accès** et un **jeton d'actualisation**.

- Le **jeton d'accès** est généralement de courte durée (expire dans environ 5 minutes, peut être personnalisé). Le jeton d'accès représente l'autorisation d'une application spécifique d'accéder à des parties spécifiques des données d'un utilisateur. Les jetons d'accès doivent rester confidentiels en transit et en stockage.
- Le **jeton d'actualisation** dure un peu plus longtemps (expire dans 24 heures, également personnalisable). C'est comparable à une session d'authentification. Après son expiration, vous avez à nouveau besoin d'une connexion complète avec nom d'utilisateur + mot de passe. Il contient les informations nécessaires pour obtenir un nouveau jeton d'accès. En d'autres termes, chaque fois qu'un jeton d'accès est requis pour accéder à une ressource spécifique, un client peut utiliser un jeton d'actualisation pour obtenir un nouveau jeton d'accès émis par le serveur d'authentification.

- Installation et configuration

- Installation : **conda install django-rest-framework-simplejwt**
- Configuration :
- ✓ Ajouter le framework dans les applications du REST_FRAMEWORK dans le fichier « settings.py » :

```
REST_FRAMEWORK = {  
    'DEFAULT_AUTHENTICATION_CLASSES': [  
        'rest_framework_simplejwt.authentication.JWTAuthentication',  
    ],  
}
```

- ✓ Modifier les paramètres des jetons dans le fichier « settings.py » :

```
SIMPLE_JWT = {  
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=60),  
    'REFRESH_TOKEN_LIFETIME': timedelta(days=15),  
    'ROTATE_REFRESH_TOKENS': False,  
    'BLACKLIST_AFTER_ROTATION': True,  
    'UPDATE_LAST_LOGIN': False,
```

+ Access token : 1min → 60 min
+ Refresh token : 1 day → 15 days

- ✓ Créer les éléments nécessaires pour créer l'authentification JWT. Ajoutez les URL ci-dessous dans le « urls.py » fichier de niveau projet qui comprend toutes les vues JWT.

```
from django.contrib import admin  
from django.urls import *  
from compteBancaire.views import *  
from rest_framework_simplejwt.views import TokenObtainPairView,TokenRefreshView,TokenVerifyView  
  
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('',include('compteBancaire.urls')),  
    path('api/token/',TokenObtainPairView.as_view()),  
    path('api/token/refresh/',TokenRefreshView.as_view()),  
    path('api/token/verify/',TokenVerifyView.as_view()),  
]
```

+ Le 1e URL permet de récupérer un access token.
+ Le 2e URL permet de récupérer un nouveau access token à partir d'un refresh token.
+ Le 3e URL permet de vérifier si un token est

valide ou non ou expiré.

- ✓ Ajouter dans le fichier « views.py » une classe d'autorisation 'permission_classes' : IsAuthenticated - Autorise uniquement les utilisateurs authentifiés à accéder à la route protégée.
- ✓ Faisons maintenant une demande authentifiée. Pour ce faire, créons d'abord un utilisateur. Je crée un compte superutilisateur : en utilisant la commande « python manage.py createsuperuser »

```
(ensam) D:\Djangoprojects\miniProjet>python manage.py createsuperuser  
Username (leave blank to use 'admin'):  
Email address: admin@test.com  
Password:  
Password (again):  
The password is too similar to the username.  
This password is too short. It must contain at least 8 characters.  
This password is too common.  
Bypass password validation and create user anyway? [y/N]: y  
Superuser created successfully.
```

+ Username : admin
+ Password : admin
+ Mail : admin@test.com

✓ Connexion pour obtenir Access et Refresh Tokens :

POST http://127.0.0.1:8000/api/token/ Send

Params Authorization Headers (10) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL

KEY	VALUE	DESCRIPTION
username	admin	
password	admin	
Key	Value	Description

Body Cookies (2) Headers (9) Test Results Status: 200 OK Time: 419 ms Size: 707 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "refresh": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2t1b190eXB1IjoicmVmcVzaCisImV4cCI6MTYxNTA2NTA0NSwianRpIjo1ZDQ3ZjRhNWU4NGYzNDQzN2FmMDF1NWU4Zj1jYyY0NWU1LCJ1c2VyX2lkIjoifQ.eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2t1b190eXB1IjoicmVmcVzaCisImV4cCI6MTYxNTA2NTA0NSwianRpIjo1ZDQ3ZjRhNWU4NGYzNDQzN2FmMDF1NWU4Zj1jYyY0NWU1LCJ1c2VyX2lkIjoifQ.eyJ0b2t1b190eXB1IjoicmVmcVzaCisImV4cCI6MTYxNTA2NTA0NSwianRpIjo1ZDQ3ZjRhNWU4NGYzNDQzN2FmMDF1NWU4Zj1jYyY0NWU1LCJ1c2VyX2lkIjoifQ.eyJ0b2t1b190eXB1IjoicmVmcVzaCisImV4cCI6MTYxNTA2NTA0NSwianRpIjo1ZDQ3ZjRhNWU4NGYzNDQzN2FmMDF1NWU4Zj1jYyY0NWU1LCJ1c2VyX2lkIjoifQ",
3   "access": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2t1b190eXB1IjoicmVmcVzaCisImV4cCI6MTYxNTA2NTA0NSwianRpIjo1ZDQ3ZjRhNWU4NGYzNDQzN2FmMDF1NWU4Zj1jYyY0NWU1LCJ1c2VyX2lkIjoifQ.eyJ0b2t1b190eXB1IjoicmVmcVzaCisImV4cCI6MTYxNTA2NTA0NSwianRpIjo1ZDQ3ZjRhNWU4NGYzNDQzN2FmMDF1NWU4Zj1jYyY0NWU1LCJ1c2VyX2lkIjoifQ",
4 }
```

✓ Ajouter dans l'entête d'autorisation le « Access Token » avec un préfixe Bearer. Puis la ressource demandée sera visible.

GET http://127.0.0.1:8000/operations/ Send

Params Authorization Headers (10) Body Pre-request Script Tests Settings Cookies

Type Bearer Token

The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. [Learn more about variables](#)

Token

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2t1b190eXB1IjoicmVmcVzaCisImV4cCI6MTYxNTA2NTA0NSwianRpIjo1ZDQ3ZjRhNWU4NGYzNDQzN2FmMDF1NWU4Zj1jYyY0NWU1LCJ1c2VyX2lkIjoifQ.eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2t1b190eXB1IjoicmVmcVzaCisImV4cCI6MTYxNTA2NTA0NSwianRpIjo1ZDQ3ZjRhNWU4NGYzNDQzN2FmMDF1NWU4Zj1jYyY0NWU1LCJ1c2VyX2lkIjoifQ.eyJ0b2t1b190eXB1IjoicmVmcVzaCisImV4cCI6MTYxNTA2NTA0NSwianRpIjo1ZDQ3ZjRhNWU4NGYzNDQzN2FmMDF1NWU4Zj1jYyY0NWU1LCJ1c2VyX2lkIjoifQ
```

Body Cookies (2) Headers (9) Test Results Status: 200 OK Time: 49 ms Size: 542 B Save Response

Pretty Raw Preview Visualize JSON

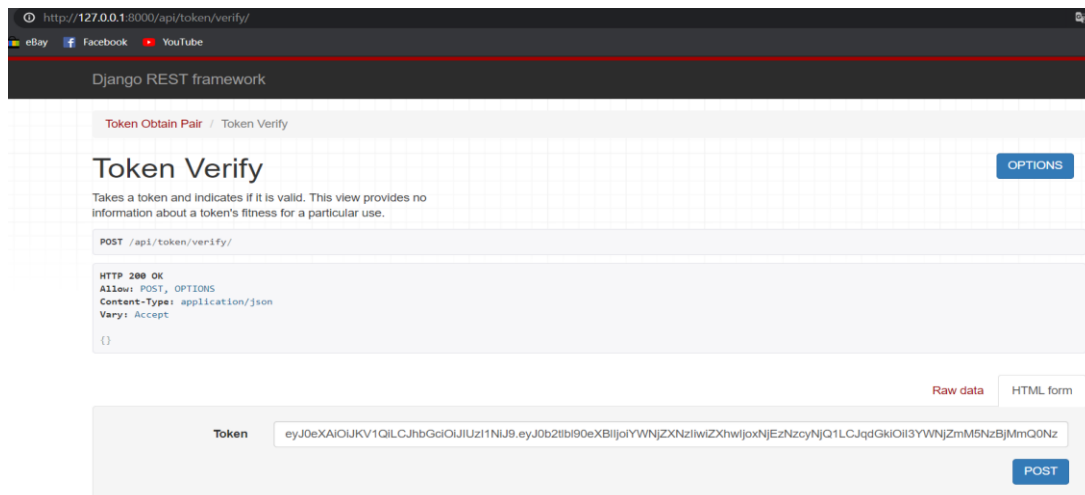
```
1 {
2   "id_op": 3,
3   "type": "c",
4   "date": "2021-02-17",
5   "amount": 442.0,
6   "account": {
7     "code": 3,
8     "balance": 0.89999999999999986,
9     "clients": [
10  ]
11 }
```

Quel est le point du jeton de rafraîchissement?

À première vue, le **jeton d'actualisation** peut sembler inutile, mais en fait, il est nécessaire de s'assurer que l'utilisateur dispose toujours des autorisations appropriées. Si votre **jeton d'accès** a une longue période d'expiration, la mise à jour des informations associées au jeton peut prendre plus de temps. En effet, la vérification d'authentification est effectuée par des moyens cryptographiques, au lieu d'interroger la base de données et de vérifier les données. Donc, certaines informations sont en quelque sorte mises en cache.

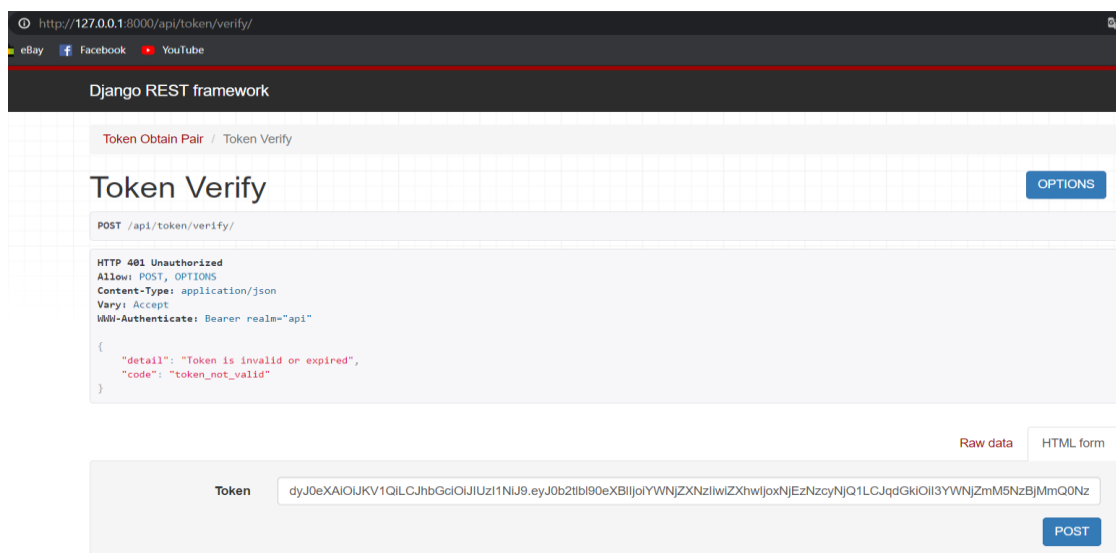
Il y a aussi un aspect de sécurité, en ce sens que le **jeton d'actualisation** ne voyage que dans les données POST. Et le **jeton d'accès** est envoyé via l'en-tête HTTP, qui peut être enregistré en cours de route. Donc, cela donne également une courte fenêtre, si votre **jeton d'accès** est compromis.

✓ Vérification du Token :



The screenshot shows the Django REST framework interface for the 'Token Verify' endpoint. The URL is `http://127.0.0.1:8000/api/token/verify/`. The response is a 200 OK status with the following headers: `Allow: POST, OPTIONS`, `Content-Type: application/json`, and `Vary: Accept`. The response body is an empty JSON object `{}`. The 'Token' field contains a long alphanumeric string.

*Dictionnaire
vide : Valid
Token.*



The screenshot shows the Django REST framework interface for the 'Token Verify' endpoint. The URL is `http://127.0.0.1:8000/api/token/verify/`. The response is a 401 Unauthorized status with the following headers: `Allow: POST, OPTIONS`, `Content-Type: application/json`, `Vary: Accept`, and `WWW-Authenticate: Bearer realm="api"`. The response body is a JSON object: `{ "detail": "Token is invalid or expired", "code": "token_not_valid" }`. The 'Token' field contains a long alphanumeric string.

*Cas d'un
Token
not valid.*