Séance 07 :

→ DJANGO – Les API REST, GraphQL & Microservices

INF37407 – Technologie de l'inforoute

Prof. **Yacine YADDADEN**, Ph. D.

UQAR

Plan

1. Introduction et Contexte
2. Qu'est-ce qu'une API REST ?
3. Configuration Django pour API REST
4. Qu'est-ce que la sérialisation ?
5. Gestion des mots de passe sur Django
6. Implémentation de l'API REST sur Django
7. Authentification & gestion des permissions
8. Interface graphique avec Swagger
9. Initiation à GraphQL
    a. Introduction générale
    b. Les schémas et les types de données
    c. Principales opérations
    d. GraphQL & Django
    e. Utilisation de GraphQL dans Django
10. Les *microservices*
11. Démonstration
12. Questions & Discussion

UQAR

Introduction et Contexte

- **Contexte :**
  - Plusieurs types de système accédant à une même et unique source de données :
    - Site ou application Web, application mobile, application de bureau, …
  - Fournir de manière *sécurisé* un *service* avec une *couche d'abstraction* avec les données.
- **Problématique :**
  - Une approche classique (*tel que nous l'avons vu sur Django*) ne le permet pas,
  - Il faut adopter une nouvelle approche afin d'avoir accès à ce genre de service.
- **Solution :**
  - Utilisation d'une API REST dont l'implémentation est possible en utilisant :
    - Java (Spring Boot), JavaScript (Express.js), C# (ASP.NET), **Python** (Flask et **Django**), …

UQAR

Qu’est-ce qu’une API REST ?

- **Définition I :** « *une API (Application Programming Interface) ou interface de programmation applicative est une solution sous forme d’un ensemble de règles permettant la communication entre différent types d’application afin d’échanger des données ou des services*. », On distingue deux styles :
  - API **REST** (***Re**presentational **S**tate **T**ransfer*) :
    - C’est un style d’architecture, les méthodes sont standardisées, moins de liaison client-serveur.
  - API **SOAP** (***S**imple **O**bject **A**ccess **P**rotocol*)
    - C’est un protocole, le serveur et le client sont étroitement liés.
- **Définition II :** « ***REST** a été défini en 2000 par Roy Fielding. C’est une suite de contraintes dont il faut tenir compte lors de la conception d’une API. L’objectif est l’optimisation et la facilité d’utilisation*. »,

Contraintes REST

Il y a six principales contraintes qui ont été définies :

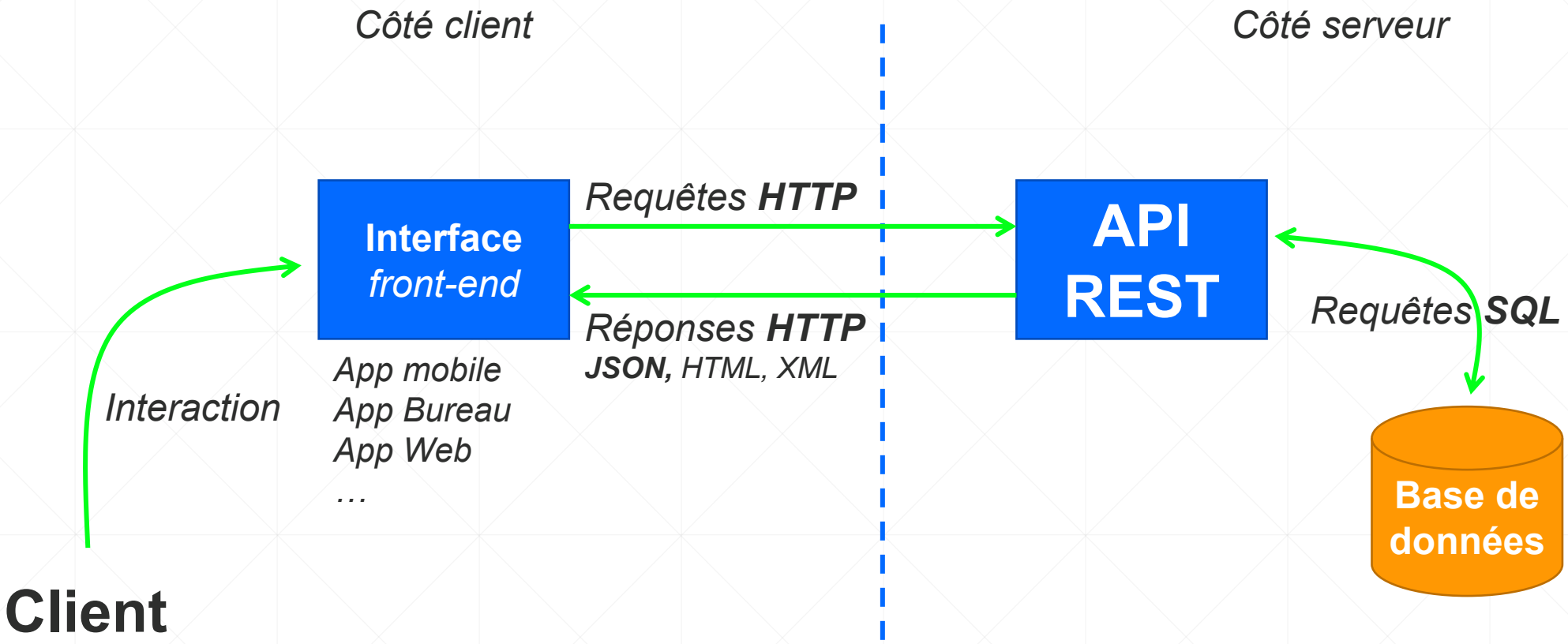
1. **Client-Server :**
   - *Il y a une séparation des rôles entre le client et le serveur,*
2. **Stateless Server :**
   - *Les requêtes contiennent l'ensemble des informations nécessaires au traitement,*
   - *Il n'y a pas de notion de session côté serveur.*
3. **Cache :**
   - *Les réponses obtenues depuis le serveur doivent être cacheable côté client (sauvegarde en local).*

UQAR

Contraintes REST (*suite*)

4. **Uniform interface :**
   - *La méthode de communication entre le client et le serveur doit être uniforme.*
5. **Layered System :**
   - *Possibilité d'ajouter des couches intermédiaires (serveur proxy, pare-feu, …).*
6. **Code-on-Demande Architecture (*optionnelle*) :**
   - *L'architecture doit permettre l'exécution du code côté client.*

Diagramme de fonctionnement



Anatomie d'une requête HTTP

Afin d'envoyer une requête HTTP, il y a quatre composants à considérer :

1. **Point de terminaison** (*endpoint*) : ***URI*** = *URL + URN*
   - *C'est l'URI (domaine, port et chemin) utilisé pour accéder à une ressource.*
2. **Méthode** (*method*) :
   - *Méthode HTTP utilisée pour accéder à la ressource (CRUD).*
3. **En-têtes** (*headers*) :
   - *Informations utilisées pour définir le format des données ou bien l'authentification.*
4. **Corps/données** (*body/data*) :
   - *Données transmises (chaîne de caractères : **JSON** ou XML).*

***U**nified **R**esource (**I**dentifier, **L**ocator et **N**ame)*

UQAR

Les différentes méthodes utilisées

Méthode HTTP	CRUD	Description
GET	Lecture	*Retourne des données*
POST	Création	*Crée un nouvel enregistrement*
PUT ou **PATCH**	Mise à jour ou modification	*Modification d'un enregistrement existant*
DELETE	Surpression	*Suppression d'un enregistrement existant*

UQAR

Les types de statut

Code de statut	Description
2xx *Success*	*La requête s'est effectuée avec succès*
3xx *Redirection*	*L'emplacement de l'URL demandé a changé*
4xx *Client error*	*Une erreur est survenue lors de l'envoi de la requête (URL)*
5xx *Server error*	*Une erreur est survenue lors du traitement de la requête*

UQAR

Configuration Django pour API REST

Afin de pouvoir utiliser la conception d'**API REST** sur Django, il faut :

1. Installer un paquet supplémentaire dans l'environnement virtuel *venv_django* :
   - Commande : `python -m pip install djangorestframework`
2. Déclarer l'utilisation du nouveau paquet dans le *project_name*/**settings.py** :
   - Aller dans la section **INSTALLED_APPS**,
   - Ajouter la ligne : *"rest_framework"*.

UQAR

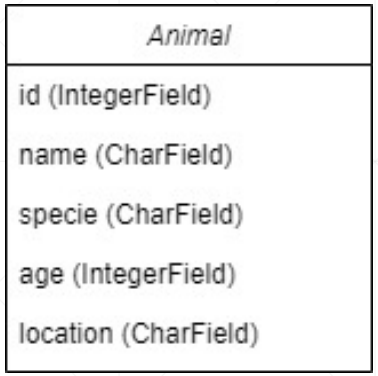
Qu’est-ce que la sérialisation ?

- **Définition :** « *C’est l’opération qui permet de convertir ou traduire les données dans un format facilement exploitable, généralement en JSON*. »,
- Pour chacune des *entités* utilisées par une API REST, il faut créer une *classe* permettant la *sérialisation* des données,
- Le Framework Django fournit une *classe de base pour la sérialisation* :
  1. Un fichier doit être créé dans *app_name*/**`serializers.py`**
  2. Ensuite à l’intérieur, il faut procéder à la création des fonctions de sérialisation :
     - Example : **`HistorySerializer`** qui est lié à l’*entité* `History`

UQAR

Qu’est-ce que la sérialisation ?

Structure du modèle



Classe de sérialisation

```python
from rest_framework import serializers
from .models import Animal


class AnimalSerializer(serializers.ModelSerializer):
    class Meta:
        model = Animal
        fields = ("name", "specie", "age", "location")
```

UQAR

Comment ça marche ?

Avant de renvoyer une réponse HTTP avec des donnés, il faut :

1. Récupérer les données en passant par le *modèle* :

```
animals = Animal.objects.all()
```

2. Appliquer la fonction de sérialisation pour générer du **JSON** :

```
animals_serializer = AnimalSerializer(animals, many=True)
```

3. Renvoyer la réponse HTTP avec un *code de statut* adéquat :

```
return Response(animals_serializer.data, status=status.HTTP_200_OK)
```

Important : *il existe également des fonctions de sérialisation inverse permettant de convertir du contenu JSON dans un format exploitable par le modèle.*

UQAR

Gestion des mots de passe sur Django

- Il n'est pas recommandé de *stocker un mot de passe en clair* dans la base de données, il faut le ***crypter*** ou ***chiffrer***,
- Le Framework Django fournit plusieurs moyens de le faire, notamment en appliquant une *fonction de hachage cryptographique* (**MD5**, **SHA-1**, **PBKDF2**, …),
- **Principe :**
  1. Un nouveau mot de passe est haché avant d'être stocké,
  2. Lors de la vérification, le mot de passe entré est également haché (*même façon*),
  3. Les deux sont ainsi comparé pour valider ou non l'authentification.
- Pour plus de sécurité, il est possible d'ajouter un **SALT** (*sel*) : « *Chaîne de caractères générée aléatoirement et stockée qui sera concaténée au mot de passe avant le l'opération de hachage (préfix, suffixe ou les deux).* ».

UQAR

Hacheurs disponibles sur Django

Déclarer l'utilisation des *hacheurs* dans le *project_name*/**settings.py** :

- Ajouter une section **PASSWORD_HASHERS**,
- Déclarer les *hacheurs* suivants, le premier sera utilisé par défaut :

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.BCryptPasswordHasher',
    'django.contrib.auth.hashers.SHA1PasswordHasher',
    'django.contrib.auth.hashers.MD5PasswordHasher',
    'django.contrib.auth.hashers.UnsaltedSHA1PasswordHasher',
    'django.contrib.auth.hashers.UnsaltedMD5PasswordHasher',
    'django.contrib.auth.hashers.CryptPasswordHasher',
]
```

Effectuer un hach de mot de passe

Dans le processus d'ajout d'un mot de passe :

1. Importer la méthode à utiliser :

```
from django.contrib.auth.hashers import make_password
```

2. Ensuite lors de la récupération du mot de passe :

```
username = request.POST.get("username")
password = request.POST.get("password")

user = User(username=username.lower(), password=make_password(password))
user.save()
```

3. Dans la base de données, on trouve :

	id	username	password
▶	1	yacine	pbkdf2_sha256$216000$NNGnogFPqaGq$S9/Tckj+nBGtbMVrVOu/n2E/MO8xJXoxKlH47RUFd48=
	NULL	NULL	NULL

UQAR

Vérification de mot de passe

Afin de faire une vérification de mot de passe :

1. Importer la méthode à utiliser :

```
from django.contrib.auth.hashers import check_password
```

2. Lors de l'appel de la méthode :

```
check_password(plain_password_to_verify, hashed_password)
```

démonstration

```
>>> from fourth_django_app.models import User
>>> from django.contrib.auth.hashers import check_password
>>> user = User.objects.first()
>>> user.username
'yacine'
>>> user.password
'pbkdf2_sha256$216000$NNGnogFPgaGq$S9/Tckj+nBGtbMVrVOu/n2E/MO8xJXoxKlH47RUFd48='
>>> check_password("Yacine", user.password)
True
>>> check_password("ahmed", user.password)
False
```

UQAR

Implémentation de l'API REST sur Django

Il y a plusieurs moyens de le faire (*views.py*) parmi lesquels figurent :

- En utilisant des **`APIView`** (*classe*),
- En passant par des *décorateurs* (*fonctions*) : **`@api_view`**,
- En utilisant des **`ViewSets`** (*classe*).

Il est également nécessaire de définir les routes adéquates en passant (*urls.py*) par :

- En utilisant **`DefaultRouter`**,
- Ou l'alternative : **`SimpleRouter`**.

API REST sur Django – Les routes

Dans le fichier *urls.py* il faut :

1. Il faut importer les paquets nécessaires :

```
from django.urls import path, include
from rest_framework import routers
```

2. Selon le cas, on utilisera :
   - **ViewSets** :

```
router = routers.DefaultRouter()
router.register(r'animals_viewset', AnimalViewSet, basename='/')
```

   - Pour les autres :

```
urlpatterns = [
    path(r'animals_decorator', apiViewDecorator, name='animals_decorator'),
    path(r'animals_apiview', AnimalAPIView.as_view(), name='animals_apiview'),
    path(r'', include(router.urls)),
]
```

Définition des actions – ViewSets

Elle est sous forme d'une *classe* qui contient différentes *méthodes* selon la requête :

- **GET** : `list()` et `retrieve()`
- **PUT** : `update()`,
- **PATCH** : `partial_update()`,
- **DELETE** : `destroy()`,
- **POST** : `create()`.

1. Importer les dépendances :

```
from rest_framework import viewsets, status
```

2. Classe et méthode :

```
class AnimalViewSet(viewsets.ViewSet):
    def list(self, request):
        animals = Animal.objects.all()
        animals_serializer = AnimalSerializer(animals, many=True)
        return Response(animals_serializer.data, status=status.HTTP_200_OK)
```

Définition des actions – `APIView`

C'est également sous forme de classe qui contient différentes *méthodes* selon la requête :

- **GET** : `.get()` ,
- **PUT** : `.put()`,
- **PATCH** : `.patch()`,
- **DELETE** : `.delete()`,
- **POST** : `.post()`.

1. Importer les dépendances :

```
from rest_framework.views import APIView
```

2. Classe et méthode :

```
class AnimalAPIView(APIView):
    def get(self, request, format=None):
        animals = Animal.objects.all()
        animals_serializer = AnimalSerializer(animals, many=True)
        return Response(animals_serializer.data, status=status.HTTP_200_OK)
```

UQAR

Définition des actions – *Les décorateurs*

- Contrairement aux deux méthodes précédentes, c'est sous forme de fonctions,
- Il est possible d'y déclarer les différentes méthodes : **GET**, **PUT**, **POST**, …
- Afin des les définir, il faut suivre :
  1. Importer les dépendances :

```
from rest_framework.decorators import api_view
```

  2. Définition de la fonction avec décorateur :

```
@api_view(['GET'])
def apiViewDecorator(request):
    animals = Animal.objects.all()
    animals_serializer = AnimalSerializer(animals, many=True)
    return Response(animals_serializer.data, status=status.HTTP_200_OK)
```

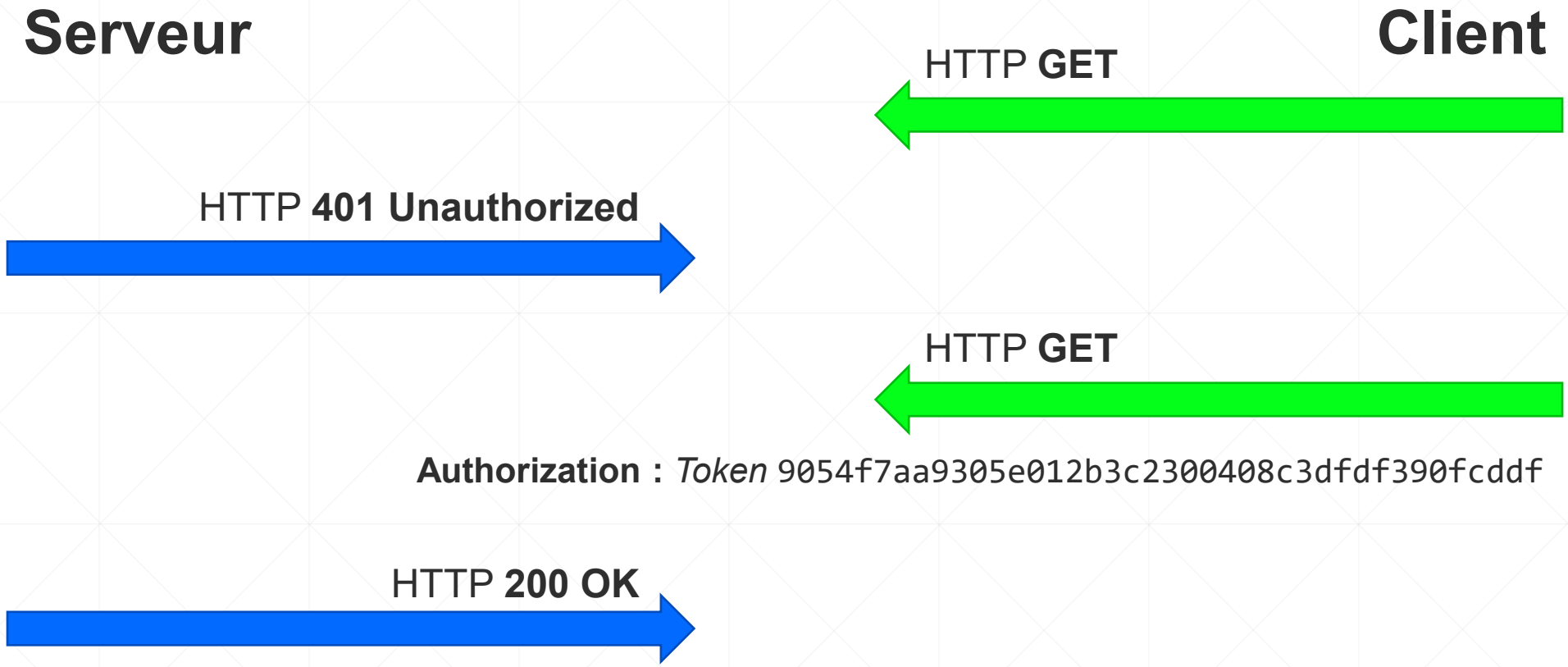
UQAR

Authentification sur Django

Django vient avec plusieurs *méthodes d'authentification* :

- **Authentification de base :**
  - *C'est la solution la plus simple, il suffit qu'avant l'exécution de chaque requête, que l'utilisateur envoie son identifiant et mot de passe pour validation. Le problème, c'est le fait que ça soit répétitif ce qui rend la tâche plus lourde.*
- **Authentification par session :**
  - *L'utilisateur envoie son identifiant et mot de passe et un ID session lui est renvoyé. Ce dernier est stocké dans les cookies. Il est envoyé dans les headers pour assurer l'authentification. Il est détruit après déconnexion de l'utilisateur. Cette méthode comprend des failles de sécurité qui peuvent être exploitées.*
- **Authentification par jeton (***voir dans la diapo suivante***).**

Authentification par Jeton (*Token*)



UQAR

Authentification par Jeton (*Token*) – Configuration

Afin d'implémenter l'authentification par jeton, il faut suivre les étapes :

1. Éditer le fichier *project_name*/**settings.py** :
   - Aller dans la section **INSTALLED_APPS**,
   - Ajouter la ligne : *"rest_framework.authtoken"*.
2. Dans le même fichier, il faut définir l'authentification par défaut :

```
REST_FRAMEWORK = {
    "DEFAULT_AUTHENTICATION_CLASSES": [
        "rest_framework.authentication.TokenAuthentication"
    ]
}
```

3. Une nouvelle table est créée pour les *jetons*, il faut appliquer les migrations :
   - Commandes : `python manage.py` **`makemigrations`** et `python manage.py` **`migrate`**

UQAR

Authentification par Jeton (*Token*) – Utilisateur

Il faut ensuite créer un nouvel utilisateur dans le modèle par défaut de Django :

1. Importer les paquets nécessaires :

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

2. Différents champs possibles[1], mais ceux obligatoires : **`username`** et **`password`**,

3. Pour créer l'utilisateur, il faut faire comme suit :

```
user = User(username="yacine", password="mdp$yacine")
user.save()
```

[1] https://docs.djangoproject.com/fr/3.1/ref/contrib/auth/

UQAR

Authentification par Jeton (*Token*) – Jeton

Maintenant que l'utilisateur est créé, il faut lui assigner un *jeton*, pour cela :

1. Importer les paquets nécessaires :

```
from rest_framework.authtoken.models import Token
```

2. Ensuite, il faut générer le *jeton* associé à l'utilisateur :

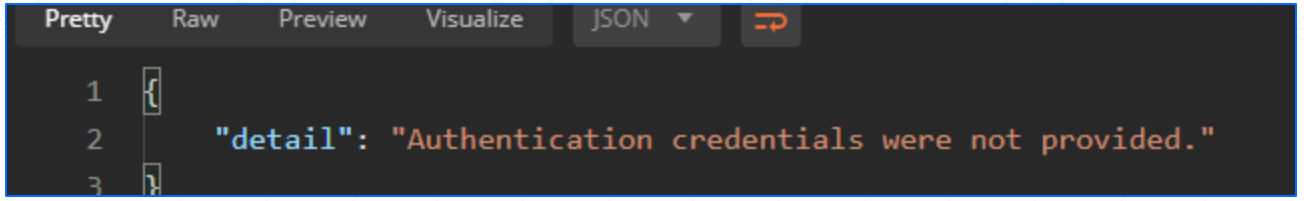
```
token = Token.objects.create(user=user)
```

- Les deux *modèles* **`User`** et **`Token`** sont liés, c'est important lors de l'*authentification*,
- Il est également possible de supprimer un *jeton*, pour cela :

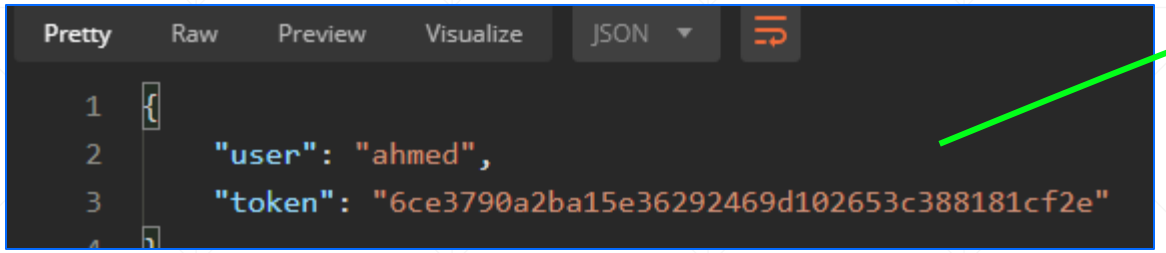
```
invalidate_token = Token.objects.filter(key="token_key")
invalidate_token.delete()
```

Authentification par Jeton (*Token*) – Test

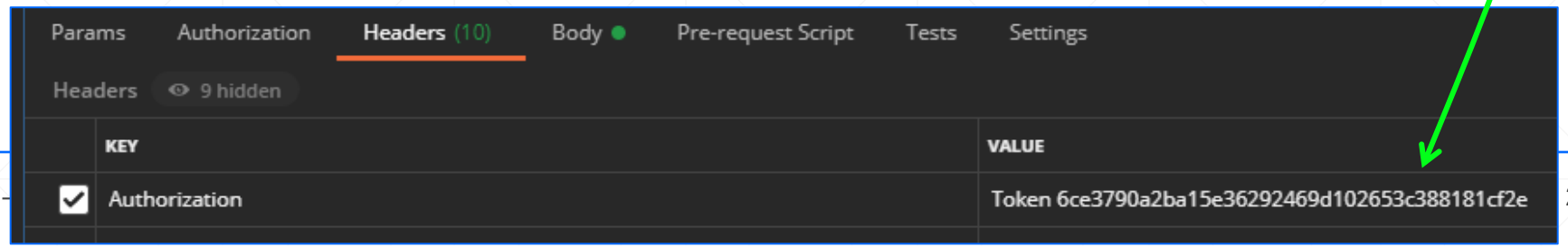
- Demander une ressource sans authentification :



- Envoyer les informations via un **POST**, ça retourne une clé :



- Appliquer les autorisations au niveau des entêtes (*headers*) de la requêtes **HTTP** :



UQAR

Gestion des permissions

Il y a différents types de permission, on utilisera la plus basique (*Si authentifié*) :

1. Importer les paquets nécessaires :

```
from rest_framework.permissions import IsAuthenticated
from rest_framework.decorators import permission_classes
```

2. Selon la façon dont a mise en place le traitement des requêtes :
   - Dans le cas de **`@api_view`** :
   - Dans le cas de **`APIView`** :
   - Dans le cas de **`ViewSets`** :

```
@permission_classes([IsAuthenticated])
def apiViewDecorator(request):
    pass
```

```
class AnimalAPIView(APIView):
    permission_classes = [IsAuthenticated]
```

```
class AnimalViewSet(viewsets.ViewSet):
    permission_classes = [IsAuthenticated]
```

UQAR

Interface graphique avec Swagger

Afin de pouvoir utiliser l'interface graphique de **Swagger**, il faut :

1. Installer un paquet supplémentaire dans l'environnement virtuel *venv_django* :
   - Commande : `python -m pip install drf-yasg`
2. Déclarer l'utilisation du nouveau paquet dans le *project_name*/**settings.py** :
   - Aller dans la section **INSTALLED_APPS**,
   - Ajouter la ligne : *"drf_yasg"*.

UQAR

Interface graphique avec Swagger (*Suite*)

Pour la configuration, il faut effectuer les étapes suivantes :

1. Rajouter les lignes suivantes au niveau du fichier *app_name*/**urls.py** :

Importer les paquets nécessaires

```
from drf_yasg.views import get_schema_view
from drf_yasg import openapi
```

Configuration de la vue avec Swagger

```
schema_view = get_schema_view(
    openapi.Info(
        title="Un titre",
        default_version="v1.0",
        description="Une petite description",
    ),
    public=True,
)
```

UQAR

Interface graphique avec Swagger (*Suite*)

Pour la configuration, il faut effectuer les étapes suivantes :

1. Rajouter le ligne suivantes au niveau du fichier *app_name*/**urls.py** :

```
path("swagger", schema_view.with_ui("swagger", cache_timeout=0), name="schema-swagger-ui")
```

Dans la liste des chemins défini dans **urlpatterns**

2. Il faut modifier le fichier *app_name*/**views.py** :

```
from drf_yasg.utils import swagger_auto_schema
```

Importer les paquets nécessaires

```
@swagger_auto_schema(tags=['Entity Groupe'], responses={200: EntitySerializer(many=True)})
```

Décorateur pour une requête dans un **APIView**

Regrouper les requêtes dans l'interface ***Swagger***

Définir le ***sérialiseur*** *à utiliser*

UQAR

Interface graphique avec Swagger (*Suite*)

Afin que **Swagger** puisse accepter une *authentification par jeton*, il faut :

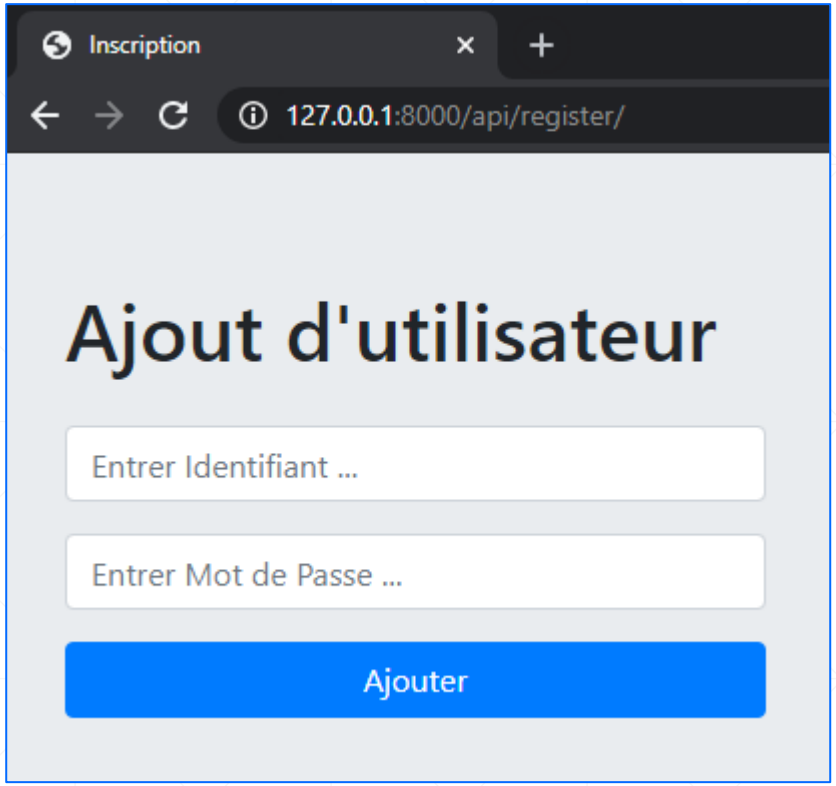
1. Ajouter les lignes suivantes au niveau du fichier *project_name*/**settings.py** :

```
SWAGGER_SETTINGS = {
    "SECURITY_DEFINITIONS": {
        "Basic": {"type": "basic"},
        "Bearer": {"type": "apiKey", "name": "Authorization", "in": "header"},
    }
}
```

Démonstration

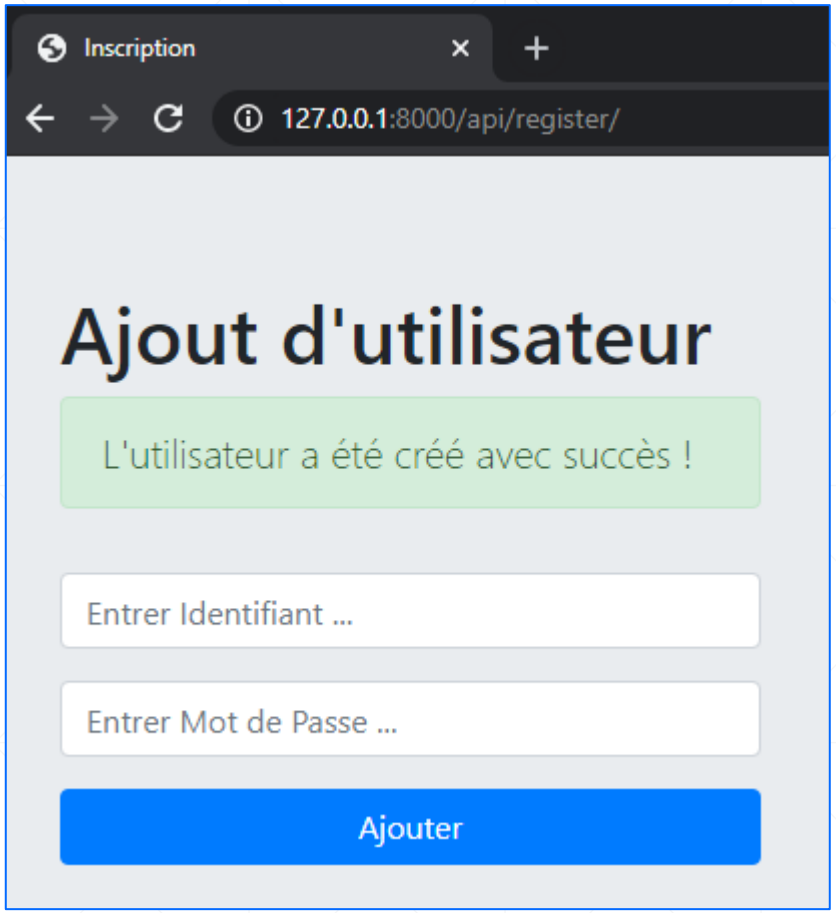
- Sur l'application de conversion entre :
  - **Degré Celsius °C** et **Degré Fahrenheit °F**
- Implémentation de l'authentification par *jeton* :
  - Enregistrement d'un nouvel utilisateur,
  - Connexion et déconnexion.
- Implémentation d'une API REST :
  - Demande de conversion,
  - Demande de l'historique.
- Appliquer les différentes permissions,
- Le résultat est comme suit :

1. Ajouter utilisateur

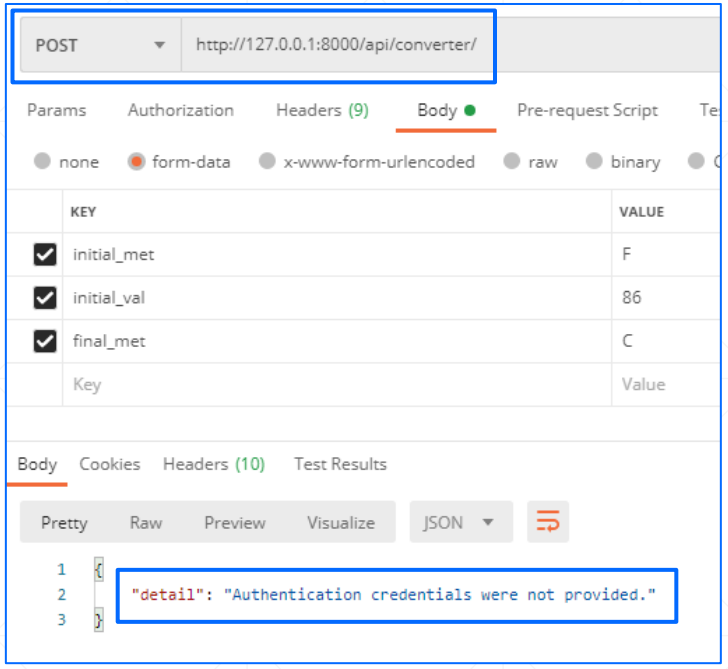


Page d'ajout d'utilisateur

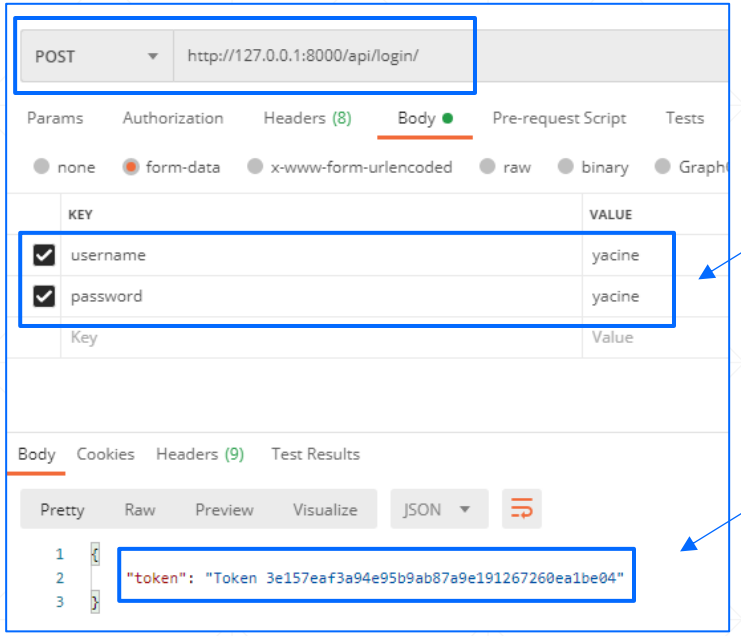
Réponse après ajout



2. Authentification par jeton



Demander une conversion sans authentification (connexion)



Information de Connexion

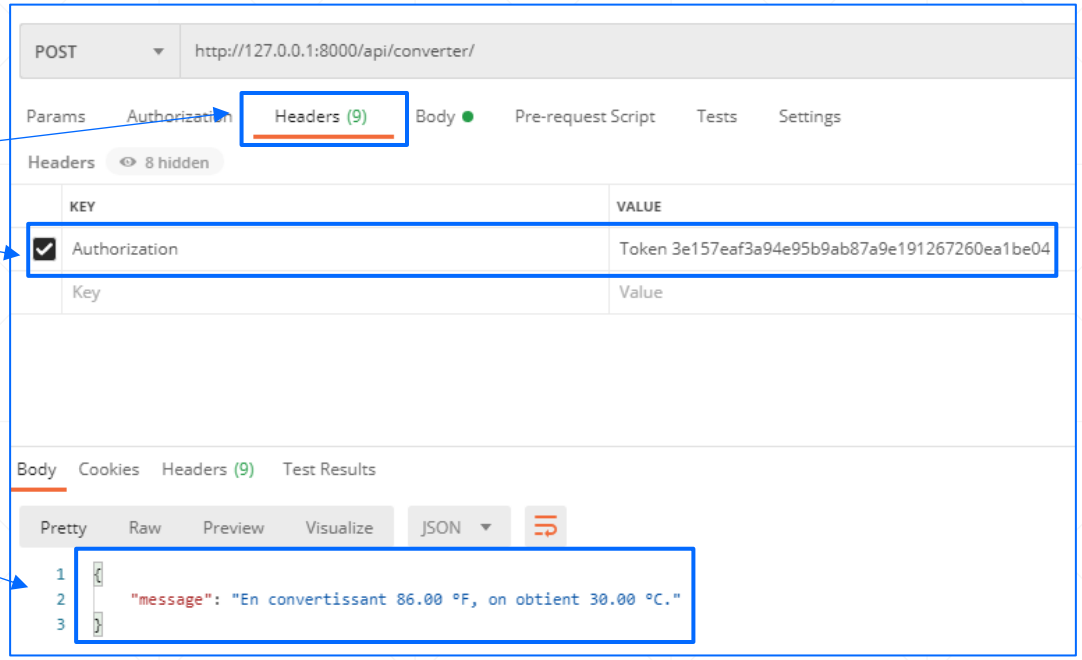
Jeton renvoyé

Demander une authentification

3. Accès aux services (permissions)

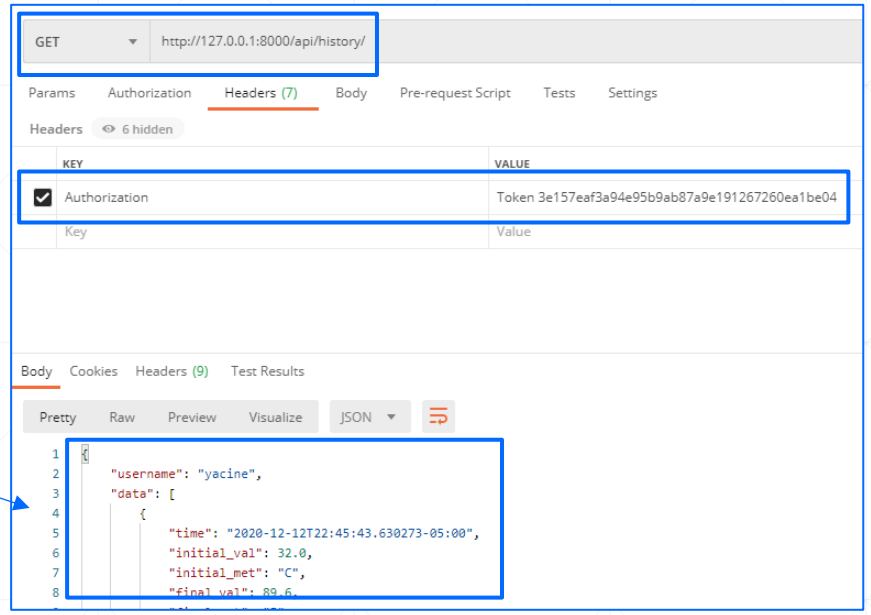
Jeton dans les entêtes (headers)

Réponse



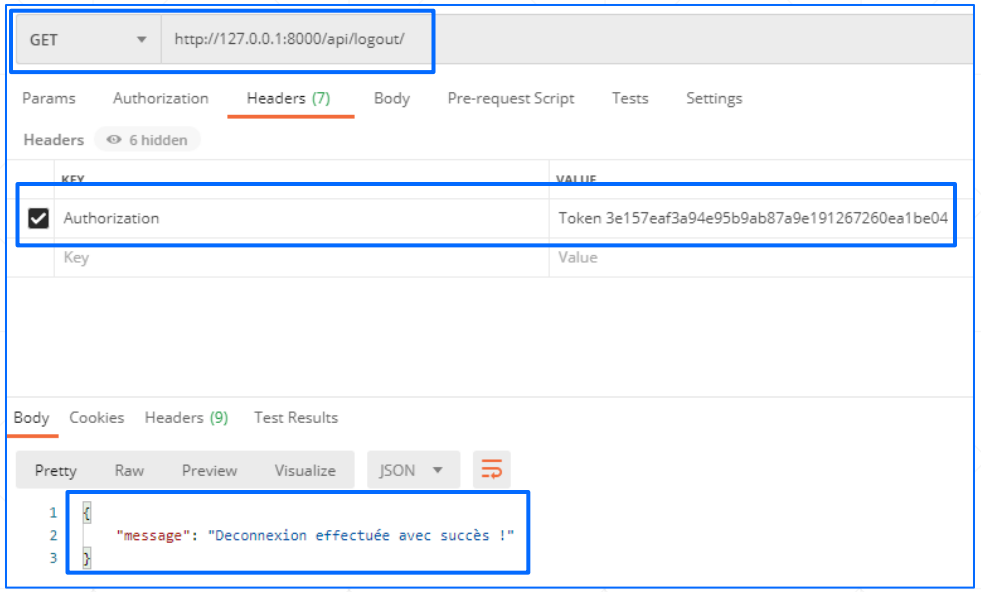
Demander une conversion avec authentification jeton (connexion)

4. Accès aux données et déconnexion



Données

Demander l'historique de conversion avec authentification jeton (connexion)



Se Déconnecter (Supprimer le jeton)

UQAR

Initiation à GraphQL

→ Introduction générale

- **Contexte**

  → Avant 2012, Facebook utilisait principalement des **APIs REST** pour ses applications Web et mobiles.

- **Problématique**

  ☒ Les *endpoints* des **APIs REST** renvoyaient trop ou pas assez de données.

  ☒ Chaque vue de l'application nécessitait plusieurs appels API

  → exemple : *profil d'un ami, ses publications, les likes, etc.*

  ☒ Les utilisateurs mobiles avaient une expérience lente et coûteuse en bande passante.

  ☒ Difficulté d'exploiter facilement les relations complexes entre données

  → exemple : *les amis de mes amis*, *les likes sur une publication*, *qui a commenté quel post, etc.*

UQAR

Initiation à GraphQL

→ Introduction générale

- **Besoins identifiés**
  - ☑ Réduire le nombre de requêtes.
  - ☑ Permettre au client de **choisir précisément quelles données récupérer**.
  - ☑ Accéder facilement aux **relations entre objets** (amis → amis de mes amis → posts → likes).
- **Solution**
  - ☑ Un **seul Endpoint** pour toutes les données.
  - ☑ Le client définit les **champs nécessaires** (ex. juste nom, photo, amis, likes).
  - ☑ Possibilité de naviguer dans les **relations entre données** dans une seule requête.

UQAR

Initiation à GraphQL

→ Présentation générale

- **Création**
  - → Développé par **Facebook** en **2012**.
  - → Rendu **open source** en **2015**.
- **Caractéristiques**
  - ☑ L’objectif étant d’optimiser les performances de ses applications mobiles.
  - ☑ Permet au client de choisir la structure des données et les attributs à renvoyer.
  - ☑ Fournie aux clients uniquement les données qu'ils ont demandées, et rien de plus.
- **Utilisation**
  - → Aujourd’hui largement utilisé par **GitHub, Netflix, Shopify, Twitter, Airbnb**, etc.

UQAR

Initiation à GraphQL

→ **Les schémas** et les types de données

→ Le **schéma GraphQL** agit en une **passerelle** entre le client et la source de données.

→ Il définit un **contrat clair** entre le client et le serveur sur les données accessibles.

→ Pour une requête reçue, le serveur la **valide selon le schéma**, l'exécute et renvoie la réponse.

- **Le schéma précise**
  - ☑ les **types de données disponibles**,
  - ☑ les **relations** entre ces données,
  - ☑ les **règles de validation**,
  - ☑ et la **structure des échanges** entre client et serveur.

UQAR

Initiation à GraphQL

→ Les schémas et **les types de données**

- **Primitives**
  - `Int` → nombre entier
  - `Float` → nombre décimal
  - `String` → chaîne de caractères
  - `Boolean` → vrai ou faux
  - `ID` → identifiant unique
- **Objet**
  - `User { id, name, email }` (utilisateur)
- **Listes**
  - `[Book]` (liste de livres)

UQAR

Initiation à GraphQL

→ Principales opérations

- **Query** ou *requête*
  - ☑ Opération de lecture qui permet aux clients de spécifier précisément les données dont ils ont besoin auprès du serveur.
  - ☑ Réduire la sur-extraction et la sous-extraction des données, optimisant ainsi les performances du serveur et du client.
- **Mutations**
  - ☑ Opération conçue spécifiquement pour l'écriture de données plutôt que pour leur lecture.
  - ☑ Les mutations permettent de modifier les données côté serveur.
  - ☑ Ajout de nouvelles entrées, mettre à jour celles existantes, voire supprimer les données inutiles.

UQAR

Initiation à GraphQL

→ GraphQL & Django

- **Graphene**

  → Bibliothèque **Python** qui permet d'implémenter facilement **GraphQL** dans une application.

  → Utiliser la commande **`pip`** pour l'installation et configuration au niveau de **`settings.py`**

  ☑ Elle fournit des outils pour créer des **schémas GraphQL** basés sur les modèles Django.

  ☑ Elle facilite la mise en place des **Queries** (*lecture*) et **Mutations** (*ajout*, *modification*, *suppression*).

- **GraphiQL**

  → **Interface web** interactive qui permet de tester et d'exécuter des requêtes **GraphQL** directement depuis le *navigateur Web*.

  → Interface Web : **`https://127.0.0.1/graphql/`**

  ☑ Fournie automatiquement avec **Graphene**.

  ☑ Permet d'écrire des **queries** et des **mutations**.

  ☑ Affiche les **résultats JSON** immédiatement.

Initiation à GraphQL

→ Utilisation de GraphQL dans Django

Il faut suivre les étapes suivantes :

1. Installer **Graphene** et configuration dans le fichier **`settings.py`**
2. Créer un modèle pour des données et effectuer la migration.
3. Configurer une route au niveau de **`urls.py`** afin d'accéder à **GraphiQL**.
4. Créer un fichier `schema.py` dans l'application et rajouter :
   a. Définir la *requête* ou **query**
   b. Définir les différentes *mutations* nécessaires
5. Lancer le serveur afin de tester avec **GraphiQL**.

UQAR

Initiation à GraphQL

→ Utilisation de GraphQL dans Django

☑ Installer **Graphene** et configuration dans le fichier **`settings.py`**

1. L'installation se fait avec la commande → `python -m pip install graphene-django`
2. Déclarer l'utilisation du nouveau paquet dans le *`project_name`*`/settings.py` :
   - Aller dans la section **INSTALLED_APPS**,
   - Ajouter la ligne : *"graphene_django"*.
3. Déclarer l'utilisation du **schéma** au niveau de *`project_name`*`/settings.py` :
   - Ajouter le code suivant :

```
GRAPHENE = {
    'SCHEMA': 'application_name.schema.schema',
}
```

UQAR

Initiation à GraphQL

→ Utilisation de GraphQL dans Django

☑ Créer un modèle pour des données et effectuer la migration

1. À titre d'exemple, nous utiliserons un modèle pour la gestion de livres :

```python
class Book(models.Model):
    id = models.AutoField(primary_key=True)
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=100)
    published_date = models.DateField()
```

2. Utiliser les commandes **`makemigrations`** et **`migrate`**

UQAR

Initiation à GraphQL

→ Utilisation de GraphQL dans Django

☑ Configurer une route au niveau de **`urls.py`** afin d'accéder à **GraphiQL**.

1. Activation de l'interface Web de **GraphiQL**
2. Désactiver la protection contre les CSRF ou *Cross-Site Request Forgery*.

```python
from django.views.decorators.csrf import csrf_exempt
from graphene_django.views import GraphQLView


urlpatterns = [
    # Active l'interface GraphiQL et désactive la protection CSRF pour les requêtes GraphQL
    path("graphql/", csrf_exempt(GraphQLView.as_view(graphiql=True))),
]
```

UQAR

Initiation à GraphQL

→ Utilisation de GraphQL dans Django

☑ Créer un fichier **`schema.py`** dans l'application et rajouter :

→ Définir les différents objets ou entités (*avec les champs*) à retourner.

1. Définir l'objet **`BookType`** dans le fichier **`schema.py`** :

```python
from graphene_django import DjangoObjectType

from .models import Book


class BookType(DjangoObjectType):
    class Meta:
        model = Book # Modèle associé
        fields = ("id", "title", "author", "published_date")  # Champs exposés via GraphQL
```

UQAR

Initiation à GraphQL

→ Utilisation de GraphQL dans Django

☑ Modifier le fichier **`schema.py`** dans l'application et rajouter :

→ Définir deux *requêtes* : l'une pour **tous les livres**, l'autre pour **un livre précis**.

```python
import graphene

class Query(graphene.ObjectType):
    # Liste de tous les livres
    all_books = graphene.List(BookType)
    # Recherche d'un livre par son identifiant
    book_by_id = graphene.Field(BookType, id=graphene.Int(required=True))


    # Renvoie tous les livres
    def resolve_all_books(root, info):
        return Book.objects.all()

    # Renvoie un livre selon son ID
    def resolve_book_by_id(root, info, id):
        try:
            return Book.objects.get(pk=id)
        except Book.DoesNotExist:
            return None
```

UQAR

Initiation à GraphQL

→ Utilisation de GraphQL dans Django

☑ Modifier le fichier **`schema.py`** dans l'application et rajouter :

→ Définir deux *mutations* : **création** d'un livre et **suppression** d'un livre existant.

```python
class CreateBook(graphene.Mutation):
    # Les arguments que la mutation attend en entrée
    class Arguments:
        title = graphene.String(required=True)
        author = graphene.String(required=True)
        published_date = graphene.Date(required=True)


    # Le champ retourné par la mutation : ici, un objet BookType (le livre créé)
    book = graphene.Field(BookType)


    # Méthode principale exécutée lors de l'appel de la mutation
    def mutate(root, info, title, author, published_date):
        # Création d'un nouvel objet Book à partir des arguments fournis
        book = Book(title=title, author=author, published_date=published_date)
        book.save() # Sauvegarde du livre en base de données
        return CreateBook(book=book) # Retourne la mutation avec le livre créé
```

UQAR

Initiation à GraphQL

→ Utilisation de GraphQL dans Django

```python
class DeleteBook(graphene.Mutation):
    # Argument attendu : l'identifiant du livre à supprimer
    class Arguments:
        id = graphene.Int(required=True)


    # Champ retourné : un booléen indiquant si la suppression a réussi
    success = graphene.Boolean()


    # Méthode exécutée lors de la mutation
    def mutate(root, info, id):
        try:
            # Recherche du livre par son identifiant (pk = primary key)
            book = Book.objects.get(pk=id)
            # Suppression du livre trouvé
            book.delete()
            # Retourne True si la suppression est réussie
            return DeleteBook(success=True)
        except Book.DoesNotExist:
            # Si le livre n'existe pas, retourne False
            return DeleteBook(success=False)
```

UQAR

Initiation à GraphQL

→ Utilisation de GraphQL dans Django

☑ Modifier le fichier **`schema.py`** dans l'application et rajouter :

1. Définition des deux *mutations* disponibles dans l'**API GraphQL**:

```
class Mutation(graphene.ObjectType):
    create_book = CreateBook.Field()   # Mutation pour créer un livre
    delete_book = DeleteBook.Field()   # Mutation pour supprimer un livre existant
```

2. Schéma **GraphQL** principal : inclut à la fois les **requêtes** et les **mutations**

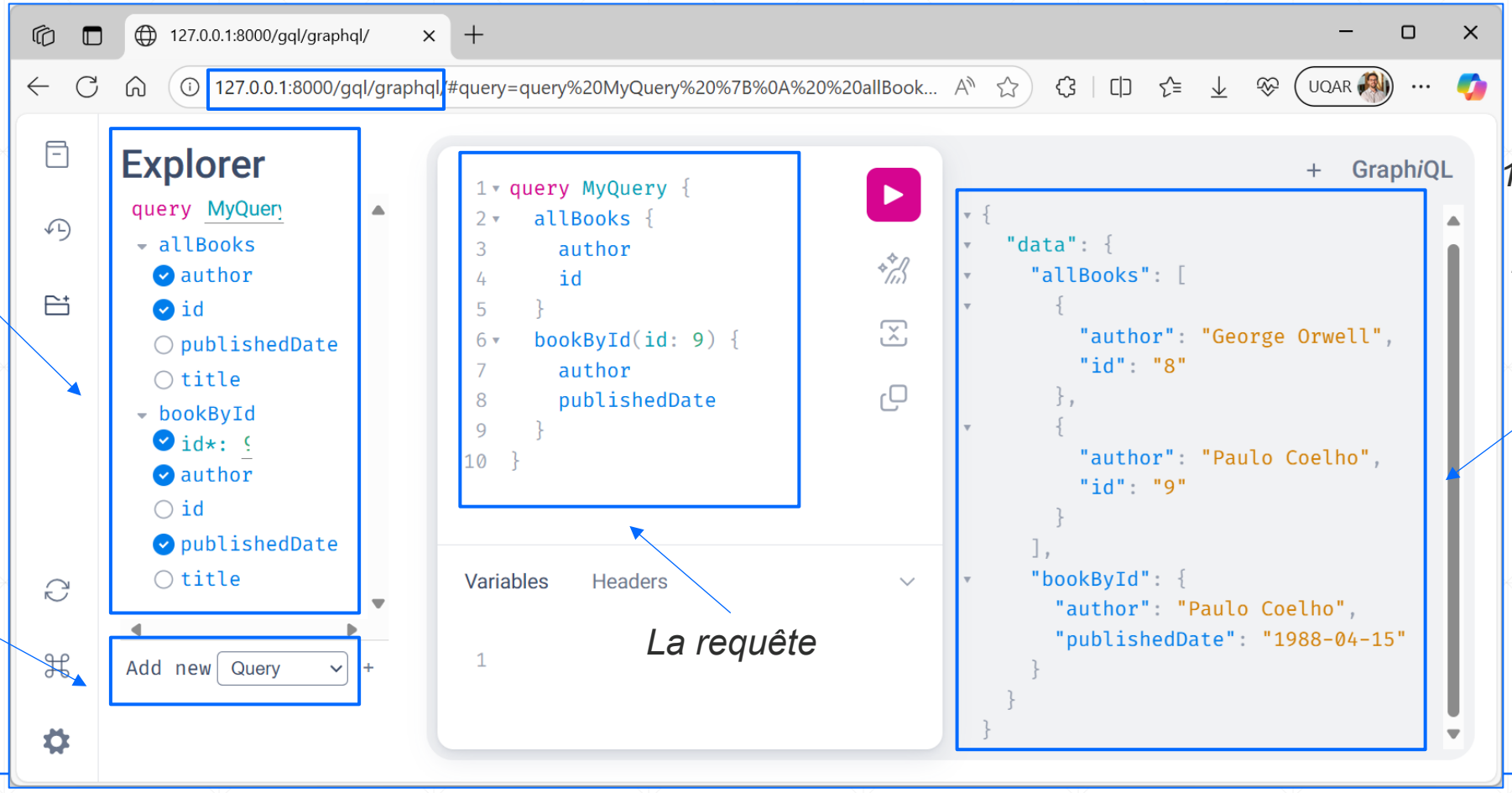
```
schema = graphene.Schema(query=Query, mutation=Mutation)
```

Initiation à GraphQL

→ Utilisation de GraphQL dans Django – Résultat (*requêtes*)

Aide à la création d'une requête

Choisir entre requête et mutation



La requête

Résultat

1. L'ensemble des livres

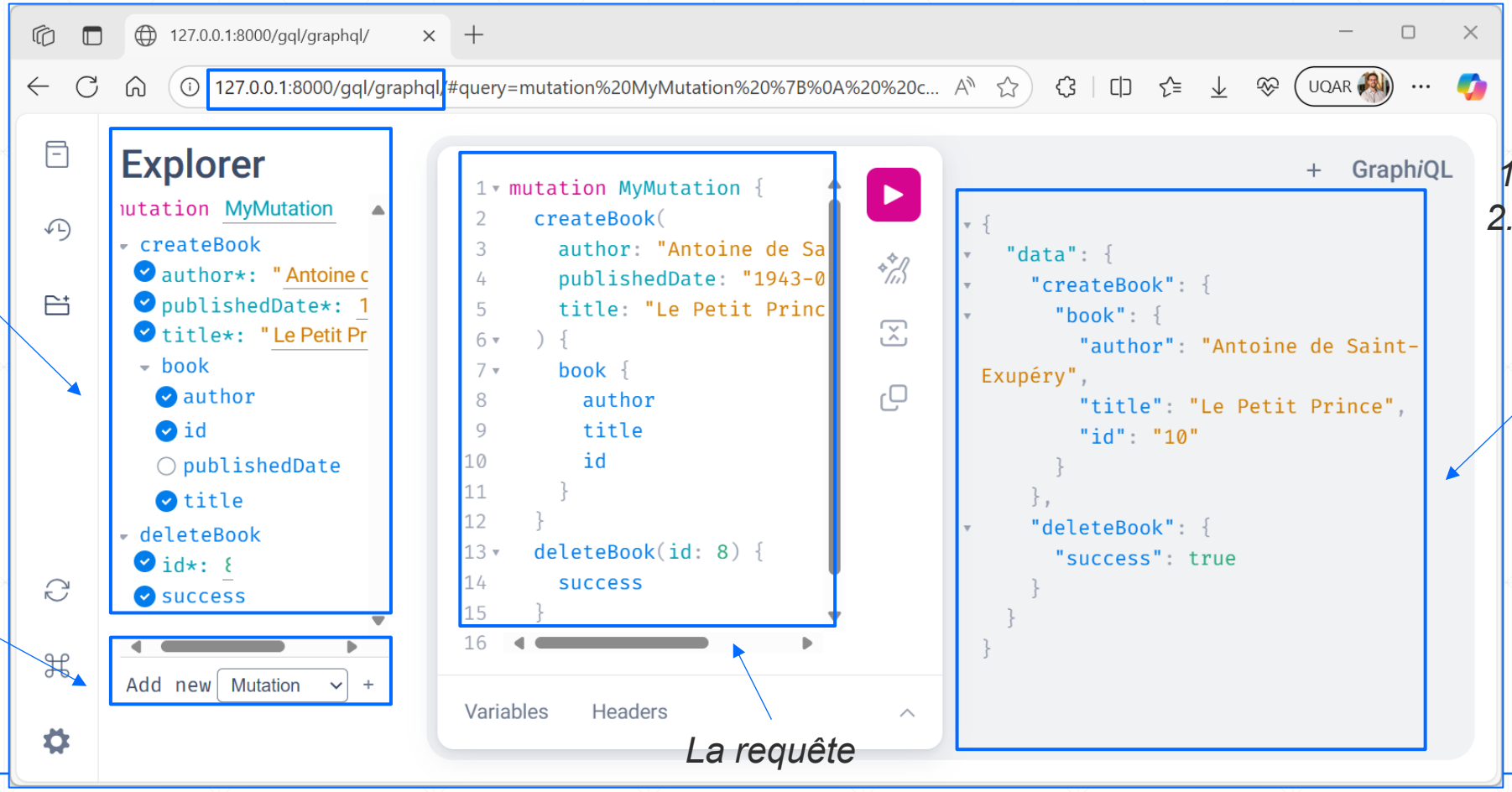
2. Un livre précis avec ID = 9

Initiation à GraphQL

→ Utilisation de GraphQL dans Django – Résultat (*mutations*)

Aide à la création d'une requête

Choisir entre requête et mutation



La requête

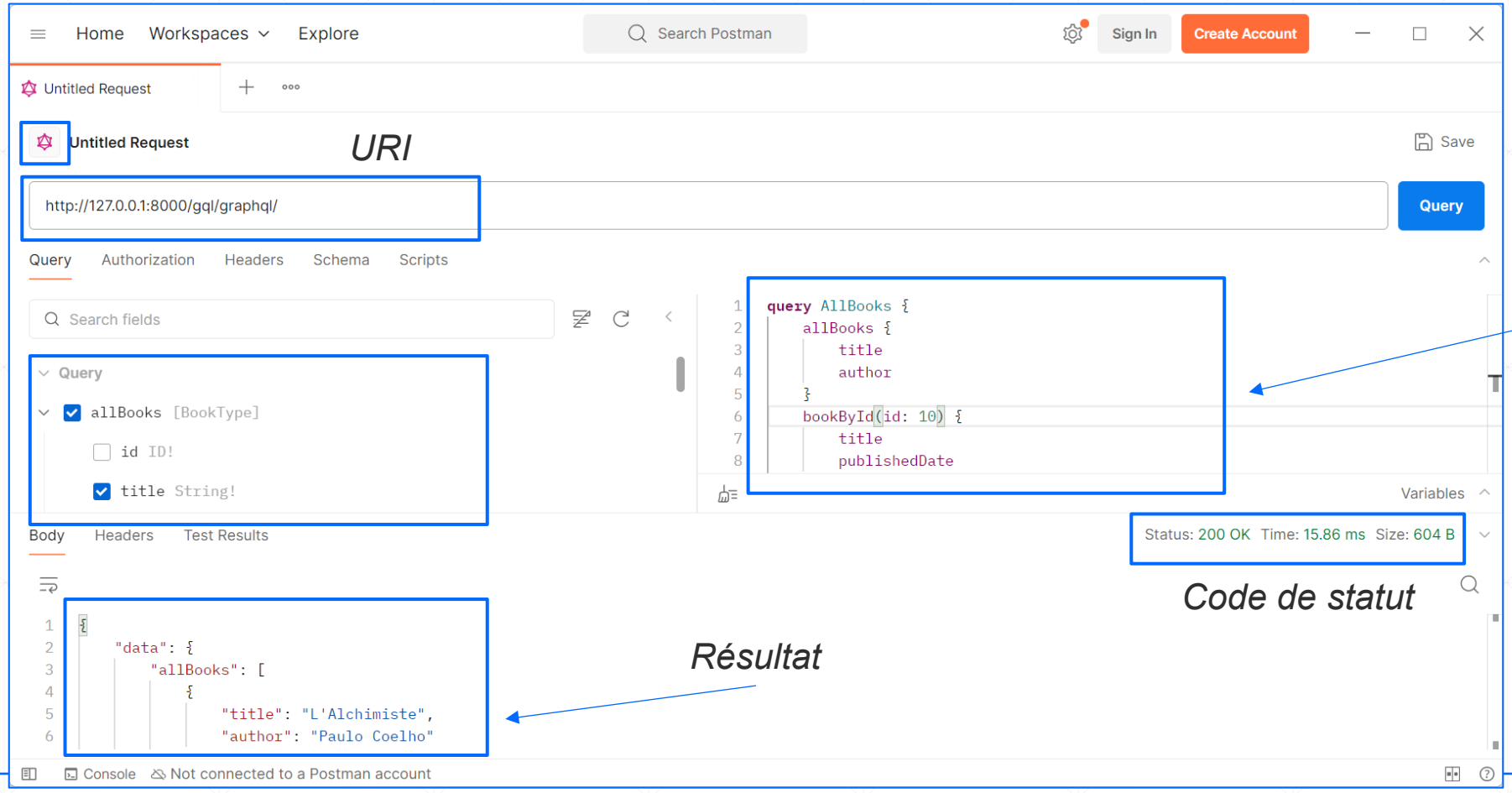
Résultat

1. Ajoute d'un livre

2. Suppression d'un livre avec ID = 8

Initiation à GraphQL

→ Utilisation de GraphQL dans Django – Postman



Choisir GraphQL

Aide à la création d'une requête

Requête

Code de statut

Résultat

Les microservices

Introduction

UQAR

Les microservices

1. Introduction et Contexte
2. Les deux différents types d'architectures
   a. Qu'est-ce que l'architecture *monolithique* ?
   b. Qu'est-ce qu'une architecture en *microservices* ?
   c. Bilan → avantages et inconvénients
3. Les *microservices* et les *conteneurs*
4. Les outils et configurations nécessaires
   a. Django
   b. Requests
5. Questions et discussion

UQAR

Introduction et Contexte

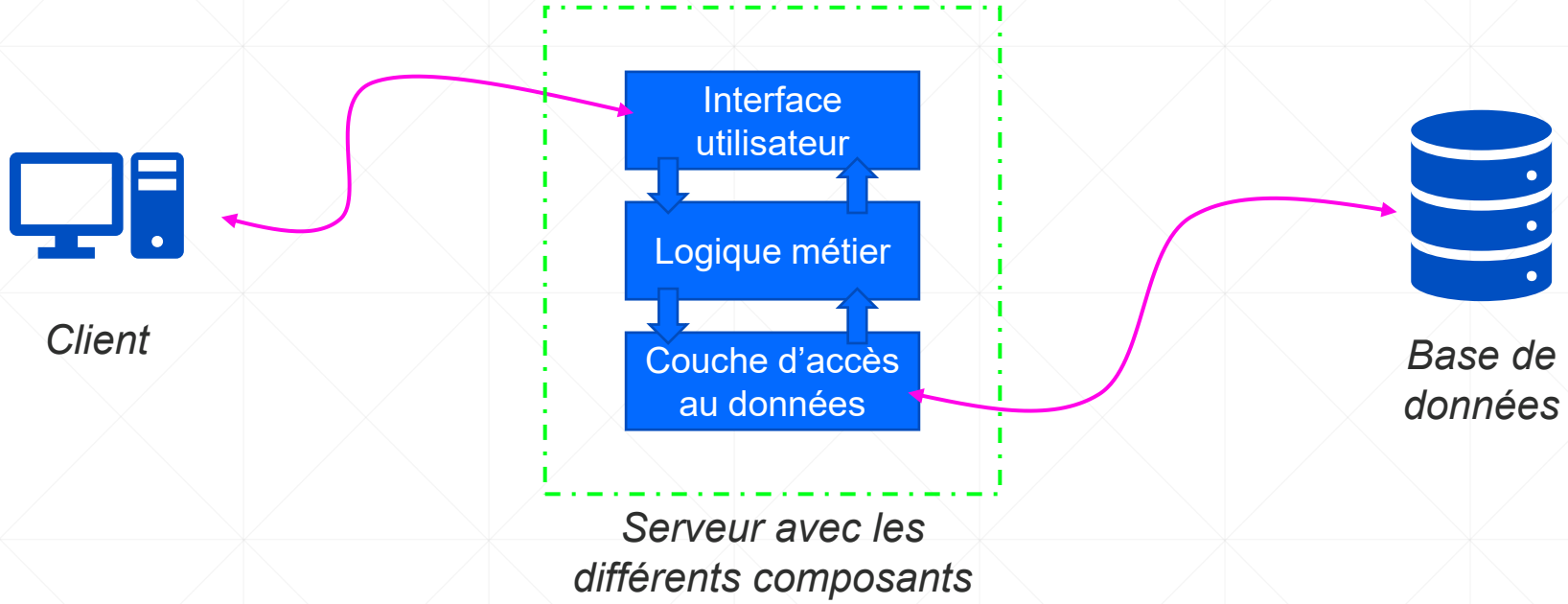
- **Contexte :**
  - Les applications développées deviennent de plus en plus complexes,
  - Les équipes qui interviennent sont de plus en plus nombreuses,
  - Les besoins des entreprises deviennent de plus en plus complexes.
- **Problématique :**
  - Difficulté à maintenir l'application sachant qu'il y a plusieurs technologies,
  - Difficulté à faire évoluer l'architecture courante en y ajoutant de nouvelles fonctionnalités.
- **Solution :**
  - Utilisation d'une architecture en *microservices,*
  - C'est une alternative à l'architecture *monolithique* classique remédiant à ses défauts.

UQAR

Architecture *monolithique*

- **Qu'est-ce qu'une architecture monolithique ?**
  - *C'est un style d'architecture logiciel où l'ensemble des composants de l'application sont construits autour d'une seule et unique entité. Le résultat est une seule et même application.*
- Généralement, on fera appel à *motif d'architecture logicielle* afin de faciliter le développement, comme :
  - MVP (*Model-View-Presentation*),
  - MVC *(Model-View-Controller),*
  - **MVT** (*Model-View-Template*),
  - MVVM (*Model-View-ViewModel*), …
- Elle aura, généralement, qu'une seule base de données pour la *persistance*,
- C'est l'architecture la plus commune et celle qui a été la plus utilisée.

Aperçu → Architecture *monolithique*

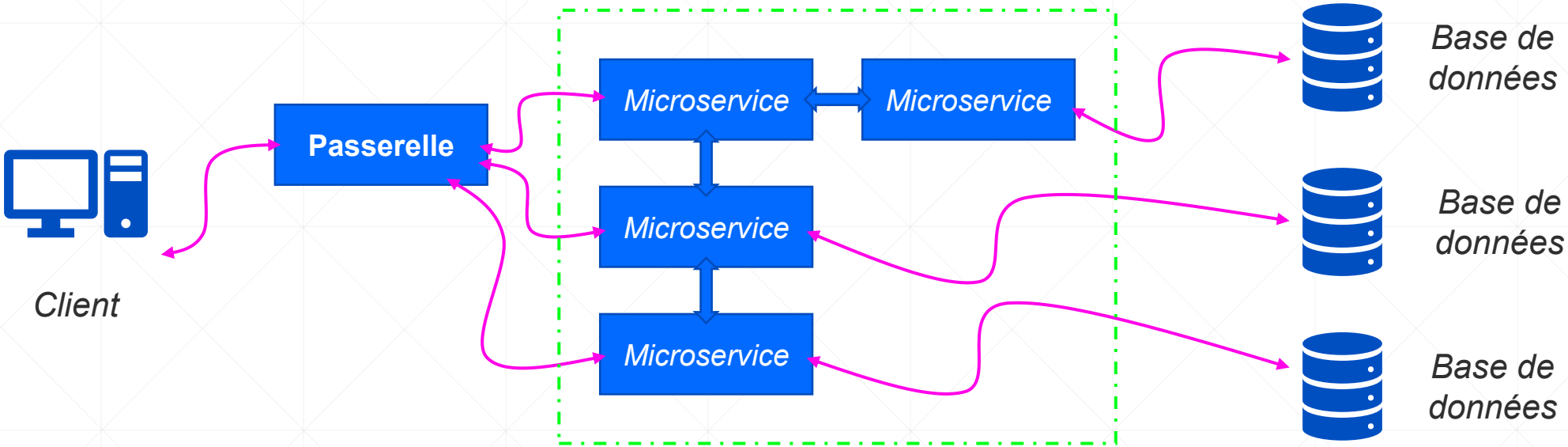


UQAR

Architecture en *microservices*

- **Qu'est-ce qu'une architecture en microservices ?**
  - *C'est un style d'architecture logiciel où l'application développée est construite autour de plusieurs microservices indépendants interagissant et communicants entre eux. Chaque microservice s'occupe de l'exécution d'une tâche particulière.*
- Généralement, cette architecture est déployée sous forme d'**API REST**.
- Cette architecture n'est pas forcément déployée sur un seul et même serveur, mais distribuée sur plusieurs serveurs,
- La *communication* entre les microservices s'effectuer à travers de **requêtes HTTP**,
- Pour la *persistance* des données, plusieurs bases de données sont utilisées,
- Elle a été discutée pour la première fois par **Adrian Cockcroft** *(ingénieur en chef chez Netflix)* en 2011.

Aperçu → Architecture en *microservices*



Architecture ***monolithique*** vs. *microservices*

Avantages

- ☑ Le développement se fait rapidement,
- ☑ Gestion des données est plus simplifiée,
- ☑ Gestion et maintenance plus aisée :

  → Cas d'une petite application.

Inconvénients

- ☒ Difficulté de mise à l'échelle,
- ☒ L'application doit être redéployée :

  → Même en cas de modification mineure.

- ☒ Difficulté du développement, car tous les développeurs de l'équipe travaillent sur une même application.

UQAR

Architecture *monolithique* vs. ***microservices***

Avantages

☑ Évolutivité et flexibilité,

☑ Facilité de déploiement :

→ Chaque *microservice* est indépendant.

☑ Réduction de la complexité :

→ Chaque *microservice* effectue une tâche.

Inconvénients

☒ Gestion de la complexité :

→ Car les *microservices* sont interconnectés

☒ Coût élevé du développement :

→ Plusieurs *développeurs* (équipes).

☒ Gestion des données plus complexe :

→ Plusieurs bases de données sont utilisées.

IMPORTANT : *le choix de l'architecture adéquate dépend de plusieurs paramètres.*

Architecture en *microservices*
→ *Utilisation des conteneurs – Docker & Kubernetes*

Afin de faciliter le déploiement des différents *microservices*, on fera appel à :

- Des *conteneurs*
  - ☑ **Définition** : *« c'est environnement d'exécution léger et portable permettant d'isoler une application et ses dépendances logicielle du reste du système d'exploitation. »*.
  - ☑ La technologie utilisée est **Docker** permettant une virtualisation au niveau du système d'exploitation. Ils sont *légers et faciles* à *déployer*.
- Un *gestionnaire* de conteneurs
  - ☑ **Définition** : *« c'est un système Open Source permettant la gestion et l'orchestration des conteneurs. Il permet de déployer, gérer et mettre à l'échelle des applications conteneurisées. »*.
  - ☑ La technologie utilisée est **Kubernetes**, elle a été développée par **Google**.

UQAR

Les outils et configurations nécessaires
→ *Django*

Dans le cas de **Django**, la façon la plus simple d'implémenter une architecture en *microservices* est de suivre les étapes suivantes :

- Chaque *microservice* est représenté par un projet **Django** indépendant,
- Chaque projet aura sa propre base de données,
- Il est également possible d'utiliser d'autre technologies comme :

  → **RabbitMQ** : « c'*est un logiciel de messagerie open-source qui facilite la communication entre différentes applications, systèmes ou composants distribués. Il agit comme un intermédiaire, permettant aux applications de partager des données de manière fiable et asynchrone, en utilisant le modèle de messagerie basé sur le protocole AMQP (Advanced Message Queuing Protocol). RabbitMQ aide à orchestrer la transmission efficace des messages entre les différents éléments d'un système distribué.* »

  → **gRPC** : « *est un framework open-source développé par Google qui facilite la communication entre des applications réparties. Il utilise le protocole RPC (Remote Procedure Call) pour permettre à des services de s'appeler mutuellement de manière transparente, quel que soit le langage de programmation dans lequel ils sont écrits. gRPC utilise le protocole de sérialisation de données protobuf (Protocol Buffers) pour échanger des messages entre les applications de manière efficace. En résumé, gRPC simplifie le développement de services distribués en fournissant un mécanisme standard pour la communication entre les différentes parties d'une application distribuée.* »

UQAR

Les outils et configurations nécessaires
→ *Requests*

Afin d'assurer la communication entre les différent *microservices* :

- Il faut installer le module **Requests** :
  → `python -m pip install` **`requests`**
  → Il permettra d'envoyer des *requêtes* **HTTP** d'un *microservice* à un autre.
- Il faut également utiliser le module **JSON** de Python :
  → Il permettra de *désérialiser* les données : **JSON** vers **Entité** (*classe*),
  → Il y a *différentes façons* de retrouver l'entité à partir du texte formaté en **JSON**.

Questions & Discussion

UQAR

Bibliographie

1. Bersini, H, Alexis, P. & Degols, G. (2018). Apprendre la programmation web avec Python et Django. Eyrolles.
2. Samson, P. (2020). DJANGO Developpez vos applications web en Python. Édition ENI.
3. Haverbeke, M. (2014). Eloquent javascript: A modern introduction to programming. No Starch Press.
4. Melé, A. (2020). Django 3 By Example: Build powerful and reliable Python web applications from scratch. P
5. Hillar, G. C. (2018). Django RESTful Web Services: The Easiest Way to Build Python RESTful APIs and Web Services with Django. Packt Publishing Ltd.ackt Publishing Ltd.