Présentation : Backend Best Practices & Good API

1 : Titre et Introduction

- **Titre de la présentation :**
  Backend Best Practices & Good API

- **Sous-titre :**
  Les principes et techniques pour construire un backend solide et des API intuitives

- **Intervenant :**
  (Votre nom, poste ou expérience)

- **Objectif :**
  Introduire les principes clés du développement backend et montrer comment concevoir des API efficaces.

2 : Agenda

- Introduction au backend et aux API
- Les meilleures pratiques en backend
- Conception d'une API « good API »
- Arborescence et architecture d'un projet
- Exemples concrets et analogies quotidiennes
- Sécurité, tests et documentation
- Q&R

---

3 : Pourquoi le backend et l'API sont cruciaux ?

- **Contexte :**

    - Le backend est le moteur de votre application : traitement, logique métier, stockage, sécurité.

    - L'API est le point d'entrée qui permet à différentes parties (front-end, applications mobiles, partenaires) de communiquer avec ce moteur.

- **Importance :**

    - Une bonne architecture facilite la maintenance, l'évolutivité et la collaboration.

    - Des API bien conçues améliorent l'expérience développeur et garantissent la cohérence des échanges de données.

---

4 : Définitions et Concepts de Base

- **Backend :**
  La partie "cachée" d'une application qui gère la logique métier, la persistance des données et les traitements côté serveur.

- **API (Application Programming Interface) :**
  Un ensemble de points de terminaison (endpoints) exposant les fonctionnalités du backend pour être utilisées par d'autres systèmes.

- **Exemple en vie quotidienne :**
  Imaginez un restaurant (backend) avec un menu (API) qui indique aux clients les plats disponibles et comment passer commande.

---

5 : Principes Fondamentaux du Backend

- **Modularité et Separation of Concerns :**

    - Diviser le code en modules (logique métier, accès aux données, gestion des erreurs…).

- **Scalabilité et performance :**
    - Concevoir pour supporter la montée en charge (exemple : mise en cache, répartition de charge).
- **Maintenance et testabilité :**
    - Écrire du code clair et testable (tests unitaires, d'intégration).
- **Sécurité :**
    - Gérer correctement l'authentification et l'autorisation (ex : JWT, OAuth2).

---

6 : Conception d'une API « Good API »

- **Principes RESTful (ou GraphQL en fonction du contexte) :**
    - Utiliser les méthodes HTTP (GET, POST, PUT, DELETE) de manière cohérente.
- **Clarté et cohérence :**
    - Nommage des endpoints, versioning de l'API pour faciliter l'évolution.
- **Documentation :**
    - Utiliser Swagger ou Postman pour documenter et tester l'API.
- **Exemple concret en vie quotidienne :**
    - Similaire à la carte d'un restaurant : chaque item (endpoint) est clairement identifié avec une description (documentation) et un mode de commande (méthode HTTP).

---

7 : Exemple d'Analogie — Le Restaurant

- **Backend = Cuisine :**
    - La cuisine prépare la commande en suivant une recette (logique métier).
- **API = Menu :**
    - Le menu décrit ce qui est proposé et permet au client de passer commande (le point d'entrée de l'API).
- **Communication :**
    - Le serveur (middleware) transmet la commande de la salle à la cuisine et inversement, assurant une communication fluide entre client et backend.

---

8 : Arborescence de Projet Type (Backend)

Voici une arborescence typique pour un projet Node.js (mais les principes sont généraux) :

```
/mon-projet-backend
├── src
│   ├── controllers       # Gestion des requêtes et réponses
│   ├── models            # Définition des schémas de données
│   ├── routes            # Définition des endpoints API
│   ├── services          # Logique métier isolée
│   ├── middlewares       # Gestion des authentifications, erreurs…
│   └── config            # Configuration de l'application (env variables, DB)
├── tests                 # Tests unitaires et d'intégration
├── docs                  # Documentation (Swagger, etc.)
└── package.json          # Dépendances et scripts
```

- **Explication :**
    - **Controllers :** Gèrent les requêtes entrantes et délèguent la logique.
    - **Models :** Définissent les structures de la base de données (ORM/Mongoose…).
    - **Routes :** Organisent les endpoints selon les ressources.

- **Services :** Regroupent la logique métier complexe.
- **Middlewares :** Intermédiaires pour la sécurité, le logging, la gestion des erreurs.

---

9 : Définir une API Simple — Exemple de Gestion de Produits

- **Endpoints :**
  - `GET /api/products` : Récupérer la liste des produits.
  - `GET /api/products/:id` : Récupérer un produit par son ID.
  - `POST /api/products` : Créer un nouveau produit.
  - `PUT /api/products/:id` : Mettre à jour un produit.
  - `DELETE /api/products/:id` : Supprimer un produit.
- **Diagramme des flux :**
  Imaginez un circuit où les requêtes du client passent par le routeur, arrivent aux controllers qui déclenchent des services, lesquels communiquent avec la base de données et renvoient la réponse via le serveur.

---

10 : Bonnes Pratiques en Conception d'API

- **Versioning :**
  - Permettre d'évoluer sans casser la compatibilité (ex : /api/v1/… puis /api/v2/…).
- **Gestion des erreurs :**

    - Fournir des messages d’erreur clairs et des codes HTTP pertinents (400, 401, 404, 500…).

- **Pagination et filtrage :**
    - Pour les endpoints listant des ressources importantes (exemple avec une liste de produits).

- **Utilisation de normes :**
    - REST, JSON:API ou GraphQL selon le besoin, et assurer la cohérence entre les endpoints.

---

11 : Sécurité et Gestion des Identités

- **Authentification et Autorisation :**
    - Implémenter des mécanismes robustes (ex : JSON Web Tokens, OAuth2).

- **Validation des données :**
    - Vérifier systématiquement les entrées utilisateurs pour prévenir les injections et autres attaques.

- **Exemple quotidien :**
    - Comme un contrôle d’accès dans un bâtiment : seule une personne autorisée peut entrer dans certaines zones, et un ticket (token) permet d’identifier l’utilisateur.

---

12 : Tests et Fiabilité du Backend

- **Pourquoi tester ?**

    - Assurer que chaque composant fonctionne de manière prévisible et que les modifications n’introduisent pas de régressions.

- **Types de tests :**
    - Tests unitaires (vérifient chaque fonction isolément).
    - Tests d’intégration (évaluent les interactions entre différents modules).
- **Exemples d’outils :**
    - Jest, Mocha pour Node.js ; pytest pour Python.
- **Bénéfice :**
    - Facilite la maintenance et permet d’avoir confiance en son code lors des déploiements continus.

---

13 : Documentation de l’API

- **Pourquoi documenter ?**
    - Permet aux développeurs internes et externes de comprendre et d’utiliser l’API sans ambiguïté.
- **Outils :**
    - Swagger / OpenAPI, Postman pour créer des collections de tests, ou Redoc pour générer des sites statiques.
- **Bonnes pratiques :**
    - Mettre à jour la documentation en parallèle de l’évolution du code.
- **Exemple concret :**
    - La documentation est comparable à un manuel d’utilisateur dans un restaurant expliquant le fonctionnement des commandes en ligne.

-

---

14 : Exemple de Code Simplifié (Pseudo-code)

Voici un exemple équivalent en Python utilisant Django Rest Framework (DRF) pour illustrer la création d'une API de gestion de produits.

1. Modèle (models.py)

Définissez votre modèle « Product » dans votre application (par exemple dans une app appelée *products*) :

```
# products/models.py
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=10, decimal_places=2)

    def __str__(self):
        return self.name
```

Ici, le modèle « Product » contient deux champs : un nom et un prix. Ce modèle servira à stocker les informations relatives aux produits.

2. Sérialiseur (serializers.py)

Créez un sérialiseur pour transformer vos instances de modèle en formats JSON exploitables par l'API :

```
# products/serializers.py
from rest_framework import serializers
from .models import Product

class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = '__all__'
```

Le sérialiseur « ProductSerializer » utilise le ModelSerializer de DRF pour simplifier la conversion entre le modèle et le JSON.

3. ViewSet (views.py)

Utilisez un viewset pour gérer automatiquement les opérations CRUD de l'API :

```
# products/views.py
from rest_framework import viewsets
from .models import Product
from .serializers import ProductSerializer

class ProductViewSet(viewsets.ModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

Le viewset « ProductViewSet » offre par défaut la gestion des opérations :

- **GET /products/** pour récupérer la liste des produits
- **GET /products/<id>/** pour récupérer un produit spécifique
- **POST /products/** pour créer un nouveau produit
- **PUT /products/<id>/** pour mettre à jour un produit
- **DELETE /products/<id>/** pour supprimer un produit

4. Configuration des URLs (urls.py)

Configurez les routes pour votre API en utilisant un routeur DRF :

```
# project/urls.py (ou products/urls.py)
from django.urls import include, path
from rest_framework.routers import DefaultRouter
from products.views import ProductViewSet
```

```
router = DefaultRouter()
router.register(r'products', ProductViewSet)

urlpatterns = [
    path('api/', include(router.urls)),
]
```

Cette configuration expose l'API sur l'URL de base « /api/ ». Le routeur se charge de générer les différentes routes pour les opérations CRUD.

---

15 : Récapitulatif & Conclusion

- **Points clés :**
    - Le backend est le cœur de l'application, et une API bien conçue favorise la communication et l'évolutivité.
    - Suivre les bonnes pratiques (modularité, tests, documentation, sécurité) améliore la qualité du code et la collaboration.
    - Une arborescence structurée permet une meilleure maintenance et compréhension du projet.
- **Conseils pour aller plus loin :**
    - Expérimenter avec des projets réels ou open source
    - Participer à des revues de code pour améliorer constamment ses pratiques