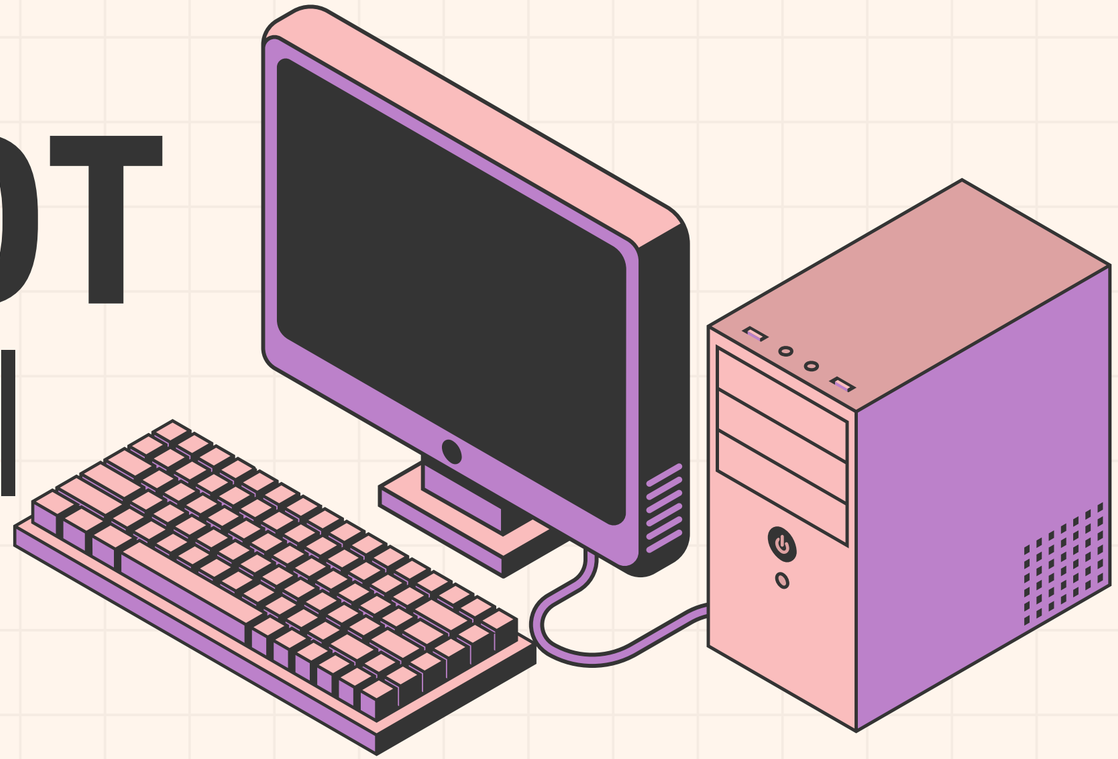
RUSESCU
ALEXANDRU

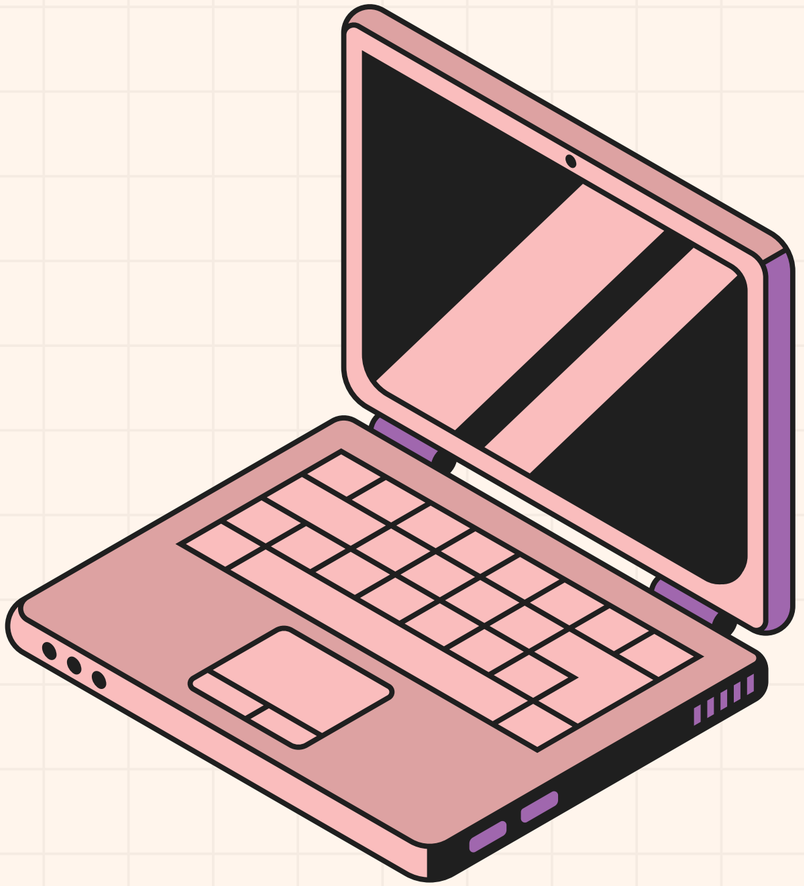
SPRING BOOT
PRODUCTS API

UNE API AVEC SPRING BOOT



CONTEXTE ET OBJECTIF

- Pourquoi Spring Boot
- Gestion centralisée d'un catalogue produits
- CRUD complet, validation poussée, pagination et authentification
- Spring Boot 3.2, JPA, PostgreSQL et Docker



ARCHITECTURE EN COUCHES

Couche Présentation

@RestController — ils reçoivent les requêtes HTTP, valident les entrées avec **@Valid**, et délèguent immédiatement au service. Aucune logique métier ici.

Couche Application

@Service annotés **@Transactional**. C'est là que vit toute la logique métier : vérification des règles, orchestration des repositories.

Couche Domaine

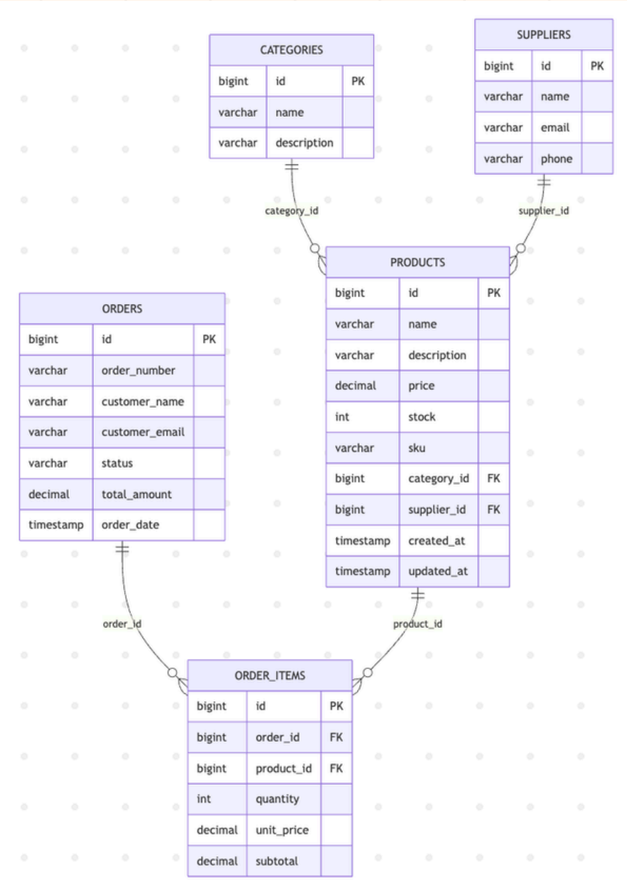
contient les entités JPA, les interfaces repository, et les contraintes de validation. J'ai utilisé l'injection par constructeur de Spring — c'est la bonne pratique car ça rend les dépendances explicites et facilite les tests.

Couche Infrastructure

PostgreSQL derrière Spring Data JPA. J'ai appliqué le pattern Repository avec des interfaces, le service dépend de l'interface JpaRepository, pas d'une implémentation concrète. C'est le Dependency Inversion Principle de SOLID."

RUSESCU
ALEXANDRU

MODÈLE DE DONNÉES



Product.java

```java
/**
 * Relation ManyToOne vers Category.
 *
 * LAZY : la catégorie n'est pas chargée automatiquement avec le produit.
 * Pour la charger, utiliser JOIN FETCH dans une requête JPQL.
 * Cela évite les requêtes inutiles quand on n'a pas besoin de la catégorie.
 */
@NotNull(message = "La catégorie est obligatoire")
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "category_id")
private Category category;

/**
 * Relation ManyToOne vers Supplier.
 *
 * LAZY : même raison que pour category.
 * LEFT JOIN FETCH dans les requêtes car le supplier peut être null.
 */
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "supplier_id")
private Supplier supplier;
```

Bean Validation

```java
    @NotBlank(message = "Le nom du produit est obligatoire")
    @Size(min = 2, max = 200, message = "Le nom doit contenir entre {min} et {max} caractères")
    @Column(nullable = false, length = 200)
    private String name;

    @NotNull(message = "Le prix est obligatoire")
    @DecimalMin(value = "0.01", message = "Le prix doit être supérieur à 0.01")
    @Digits(integer = 8, fraction = 2, message = "Le prix doit avoir au maximum 8 chiffres entiers et 2 décimales")
    @ValidPrice
    @Column(nullable = false, precision = 10, scale = 2)
    private BigDecimal price;

    @Min(value = 0, message = "Le stock ne peut pas être négatif")
    @Column(nullable = false)
    private int stock;

    @ValidSKU
    @Column(unique = true, length = 10)
    private String sku;
```

Génération%20des%20ids

```java
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

FONCTIONNALITÉS CLÉS

- CRUD complet sur les 4 entités avec Spring MVC et les bons codes HTTP
- Validation et gestion des erreurs avec un @ControllerAdvice qui centralise toutes les exceptions.
- Pagination sur GET /api/products
- Spring Security avec HTTP Basic Auth
- Swagger UI sur /swagger-ui.html

DÉMO LIVE

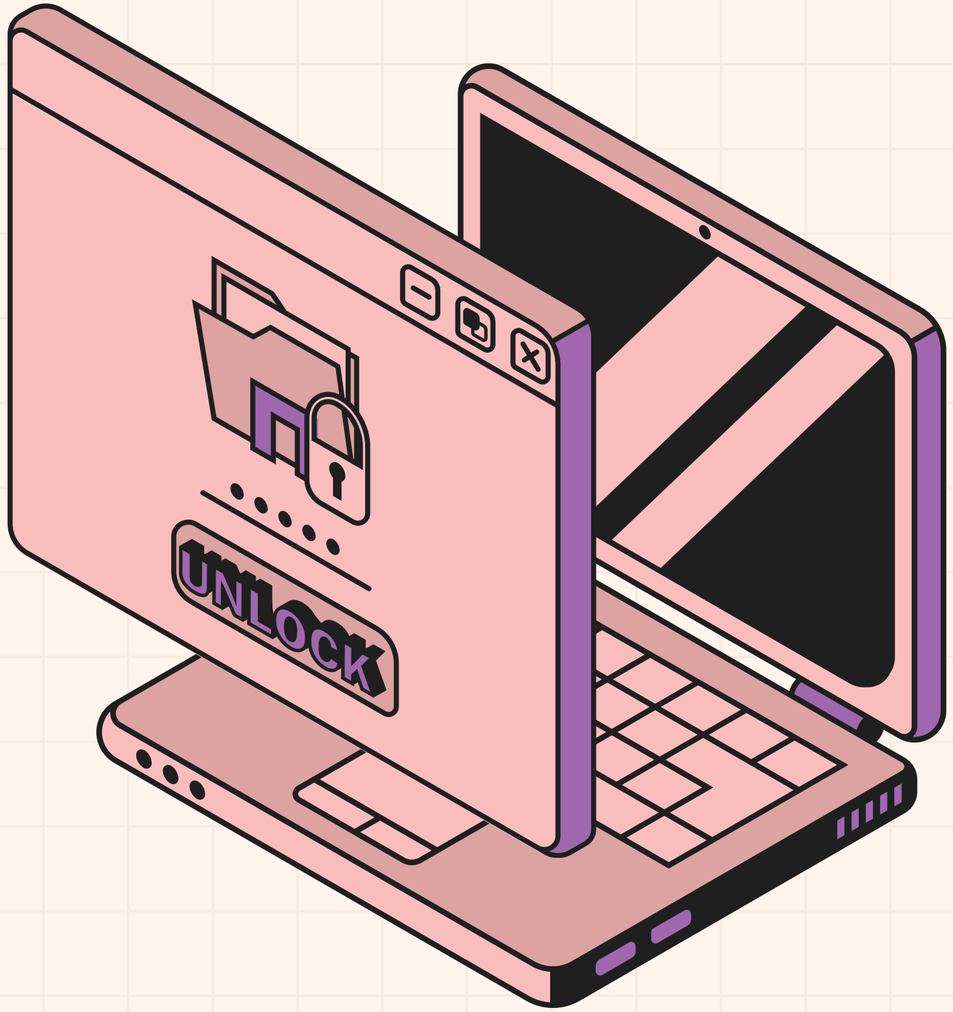
RUSESCU
ALEXANDRU

CONCLUSION

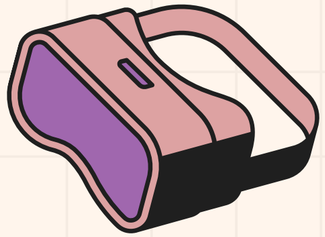
Pour conclure, j'ai livré une API REST complète avec une architecture professionnelle en 4 couches, du CRUD validé, une gestion des erreurs centralisée, de la pagination, de la sécurité et de la documentation.

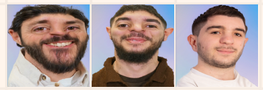
Ce que j'ai retenu de plus important : JPA demande de comprendre ce qu'il génère. Le problème N+1 est totalement invisible jusqu'à ce qu'on active les logs SQL, spring.jpa.open-in-view=false force à y penser dès la conception des requêtes.

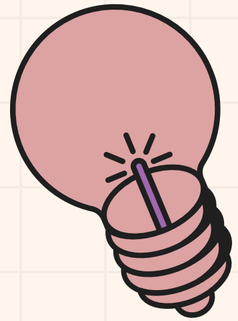
Si j'avais plus de temps, je remplacerais le Basic Auth par du JWT pour des tokens à durée limitée, et j'ajouterais des tests automatisés avec @SpringBootTest pour valider chaque endpoint au build.



RUSESCU
ALEXANDRU







MERCI

HTTP://LOCALHOST:8081/SWAGGER-UI