Java, the JVM & Spring Boot

A Complete Integration Guide for Python Developers

Targeting Spring Boot Banking Codebases

Destinataire	Alexandro Disla
Phase	Études — Fondations Java & JVM
Objectif	Lecture et compréhension complète d'un codebase Spring Boot bancaire
Prérequis	Python (FastAPI, SQLAlchemy, Pydantic), notions ORM/JPA
Date	Février 2026

Table des Matières

Partie I — La JVM : Comprendre la Machine

Avant d'écrire la moindre ligne de Java, tu dois comprendre la machine qui exécute ton code. La différence fondamentale entre Python et Java commence ici : Python est un langage interprété (CPython lit et exécute ligne par ligne), Java est un langage compilé vers un bytecode intermédiaire qui tourne sur une machine virtuelle. Cette architecture a des conséquences profondes sur tout ce que tu écriras.

1.1 Le Modèle de Compilation Java

En Python, quand tu exécutes `python app.py`, l'interpréteur CPython parse ton code source, le compile en bytecode (.pyc) en mémoire, et l'exécute immédiatement. Tout se passe en une étape transparente. En Java, le processus est explicitement découpé en deux phases distinctes.

Phase 1 — Compilation (javac). Le compilateur Java (`javac`) prend tes fichiers source `.java` et les transforme en fichiers `.class` contenant du bytecode. Ce bytecode n'est pas du code machine natif — c'est un jeu d'instructions pour la JVM. Le compilateur vérifie à ce stade : les types sont-ils cohérents ? Les méthodes appelées existent-elles ? Les imports référencent-ils des classes réelles ? Si une erreur est détectée, la compilation échoue AVANT toute exécution. C'est la différence fondamentale avec Python où les erreurs de type ne sont découvertes qu'au runtime.

Phase 2 — Exécution (java). La JVM charge les fichiers `.class`, vérifie leur intégrité (bytecode verification), et les exécute. Le composant clé est le JIT (Just-In-Time) compiler : la JVM observe quelles parties du code sont exécutées fréquemment («hot spots») et les compile dynamiquement en code machine natif pour la performance. C'est pour ça que les applications Java sont lentes au démarrage (warm-up) mais rapides ensuite.

Aspect	Python (CPython)	Java (JVM)
Fichier source	.py	.java
Compilation	Implicite en mémoire (.pyc)	Explicite avec javac → .class
Vérification types	Runtime uniquement	Compile-time + Runtime
Exécution	Interpréteur CPython	JVM + JIT compiler
Portabilité	Partout où CPython est installé	Partout où une JVM existe (WORA)
Performance	Lent (interprété)	Rapide après warm-up (JIT)
Démarrage	Instantané	Lent (chargement classes + JIT)
Typage	Dynamique (duck typing)	Statique (déclaré à la compilation)

Analogie Python
Si tu utilises mypy pour le type checking en Python, tu as déjà un aperçu de ce que javac fait obligatoirement. La différence : mypy est optionnel et peut être ignoré. javac est un passage obligé — ton code ne tourne pas s'il ne compile pas.

1.2 WORA — Write Once, Run Anywhere

Le slogan historique de Java est «Write Once, Run Anywhere» (WORA). Le bytecode `.class` est identique quel que soit l'OS. C'est la JVM spécifique à chaque plateforme (Windows, Linux, macOS) qui traduit le bytecode en instructions machine natives. C'est exactement la même idée que Docker, mais au niveau du langage : la JVM est le «container» qui abstrait le système d'exploitation.

1.3 JDK, JRE, JVM — Les Trois Niveaux

Tu rencontreras constamment ces trois acronymes. Ils forment des couches emboîtées :

JVM (Java Virtual Machine) — Le moteur d'exécution pur. Charge le bytecode, le vérifie, l'exécute. Contient le garbage collector (GC) qui gère la mémoire automatiquement — pas de `malloc/free` comme en C, pas de compteur de références comme CPython. Le GC de Java utilise des algorithmes générationnels sophistiqués (G1, ZGC, Shenandoah) qui pausent l'application pour nettoyer la mémoire.

JRE (Java Runtime Environment) — JVM + les bibliothèques standard (java.lang, java.util, java.io, etc.). C'est le minimum pour EXÉCUTER du Java. Historiquement distribué séparément, aujourd'hui intégré au JDK.

JDK (Java Development Kit) — JRE + les outils de développement : `javac` (compilateur), `javadoc` (documentation), `jshell` (REPL que tu connais déjà d'IJava), `jar` (packaging). C'est ce que tu installes en tant que développeur.

Equivalent Python
JDK = Python installé (interpréteur + pip + venv + standard library). JRE = Un runtime Python minimaliste. JVM = CPython lui-même.

1.4 Le Classpath — L'Équivalent du PYTHONPATH

En Python, quand tu fais `import pandas`, l'interpréteur cherche le module dans une liste de chemins définie par `sys.path` (qui inclut le `PYTHONPATH`, le site-packages du virtualenv, etc.). En Java, le mécanisme équivalent s'appelle le **classpath**.

Le classpath est la liste des emplacements où la JVM cherche les fichiers `.class` et les `.jar` (Java ARchive — un ZIP contenant des .class). Quand ton code fait `import jakarta.persistence.Entity`, la JVM cherche dans le classpath un fichier `jakarta/persistence/Entity.class`.

En pratique, tu ne gères JAMAIS le classpath manuellement. **Maven** (ou Gradle) s'en charge. Quand tu déclares une dépendance dans `pom.xml`, Maven télécharge le JAR, résout les transitives, et configure automatiquement le classpath. C'est exactement ce que `uv add pandas` fait pour ton virtualenv Python, mais avec une résolution plus stricte.

1.5 Le Système de Packages — Structure Obligatoire

En Python, la structure des fichiers est flexible. Tu peux mettre `models.py` n'importe où et l'importer avec un chemin relatif ou absolu. En Java, la structure des fichiers est **dictée par les packages**.

Un package Java est un namespace hiérarchique, déclaré en première ligne de chaque fichier : `package com.creditscoring.model;`. Cette déclaration **doit correspondre exactement** à l'arborescence des dossiers. Le fichier `Client.java` avec le package `com.creditscoring.model` doit être physiquement dans `src/main/java/com/creditscoring/model/Client.java`.

La convention Java pour les noms de packages est le **reverse domain name** : `com.tekkod.banking`, `org.apache.commons`. Cela garantit l'unicité mondiale des noms. En Python, l'équivalent serait d'avoir `import com.tekkod.banking.model.Client` — mais Python n'impose pas cette structure.

Concept	Python	Java
Namespace	Module (fichier .py)	Package (dossier avec package declaration)
Convention nommage	snake_case (my_module)	reverse domain (com.company.project)
Déclaration	Implicite (nom du fichier)	Explicite : package com.x.y;
Correspondance fichier	Flexible	1 classe publique = 1 fichier du même nom
Import	import module / from module import X	import com.x.y.Class
Import wildcard	from module import *	import com.x.y.*
Visibilité par défaut	Tout est public (_ convention)	Package-private (visible dans le package)

Partie II — La Syntaxe Java : Cartographie Complète pour Développeurs Python

Cette partie couvre systématiquement chaque élément de syntaxe Java que tu rencontreras dans un codebase Spring Boot bancaire. Pour chaque concept, je donne l'équivalent Python que tu connais déjà.

2.1 Le Système de Types — La Différence Fondamentale

Java est un langage à **typage statique fort**. Chaque variable, chaque paramètre de méthode, chaque retour de fonction doit avoir un type déclaré à la compilation. Il n'y a aucun équivalent au duck typing de Python.

2.1.1 Types Primitifs vs Types Objets

Java distingue deux catégories de types. Les **types primitifs** sont des valeurs brutes stockées directement en mémoire stack — ils n'ont pas de méthodes, pas de null, pas d'héritage. Les **types objets** (ou types référence) sont des instances allouées sur le heap, référencées par un pointeur, et peuvent être null.

Primitif	Wrapper (Objet)	Taille	Python équivalent	Usage bancaire
byte	Byte	8 bits	int (petit)	Rarement utilisé
short	Short	16 bits	int (petit)	Rarement utilisé
int	Integer	32 bits	int	Compteurs, identifiants
long	Long	64 bits	int (grand)	IDs base de données (clés primaires)
float	Float	32 bits	float	JAMAIS pour montants financiers
double	Double	64 bits	float	JAMAIS pour montants financiers
boolean	Boolean	1 bit	bool	Flags (is_active, is_approved)
char	Character	16 bits	str (1 char)	Rarement utilisé directement

Règle bancaire critique : BigDecimal pour l'argent
En Python, tu utilises Decimal (from decimal import Decimal) pour les montants. En Java, c'est BigDecimal. JAMAIS float ou double pour les montants financiers. Le motif est

identique : les flottants IEEE 754 ont des erreurs d'arrondi (0.1 + 0.2 != 0.3). Dans un système bancaire, une erreur de 0.01€ sur des millions de transactions = pertes réelles. Tu as déjà vu ça dans tes modèles JPA : @Column(precision = 12, scale = 2) private BigDecimal annualIncome.

2.1.2 Autoboxing et Unboxing

Java convertit automatiquement entre primitifs et wrappers. `int x = 5;` est un primitif. `Integer x = 5;` déclenche l'**autoboxing** : le compilateur insère automatiquement `Integer.valueOf(5)`. L'inverse (Integer → int) est l'**unboxing**.

Pourquoi c'est important : les collections Java (List, Map, Set) ne peuvent **PAS** contenir de primitifs. Tu ne peux pas faire `List<int>` — tu dois faire `List<Integer>`. Les génériques (qu'on verra plus loin) ne fonctionnent qu'avec des types objets. Attention : un `Integer` peut être `null` alors qu'un `int` ne le peut pas. Déboxer un null provoque un `NullPointerException` — l'erreur la plus fréquente en Java.

2.1.3 String — Immuable et Omniprésent

En Python, `str` est immuable. En Java, `String` est aussi immuable. La différence : en Java, String est une **classe** (type objet), pas un type primitif. Les conséquences pratiques :

Comparaison : En Python, `==` compare la valeur des strings. En Java, `==` compare les **références mémoire** (comme `is` en Python). Pour comparer le contenu, tu dois utiliser `.equals()`. C'est un piège classique. `"hello" == "hello"` peut retourner true (grâce au string pool), mais `new String("hello") == "hello"` retourne false. Utilise **TOUJOURS** `.equals()` pour comparer des Strings.

Text Blocks (Java 15+) : L'équivalent des triple-quotes Python. Au lieu de concaténer des strings pour une requête SQL, tu peux écrire un text block avec `"""`. Tu as déjà vu ça dans tes requêtes JDBC du document de scoring.

2.2 Variables et Déclarations

En Python, tu écris `name = "Alexandro"` et le type est inféré. En Java, tu déclares le type explicitement : `String name = "Alexandro";`. Chaque déclaration suit le pattern : `Type nomVariable = valeur;`.

2.2.1 var — L'Inférence de Type Locale (Java 10+)

Depuis Java 10, le mot-clé `var` permet l'inférence de type pour les variables locales. `var name = "Alexandro";` est compilé exactement comme `String name = "Alexandro";` — le compilateur infère le type. **Attention** : `var` ne fonctionne que pour les variables locales (dans

une méthode), pas pour les champs de classe, les paramètres de méthode, ou les types de retour. Ce n'est PAS du typage dynamique — le type est fixé à la compilation.

2.2.2 final — L'Équivalent de la Constante

Le mot-clé `final` rend une variable non-réassignable après initialisation. `final int MAX_SCORE = 850;` est l'équivalent conceptuel d'une constante Python (`MAX_SCORE = 850` en convention UPPER_CASE). La différence : `final` est **enforcé par le compilateur**. Toute tentative de réassignation provoque une erreur de compilation. En Python, rien n'empêche de modifier une «constante».

Note subtile : `final` empêche la **réassignation**, pas la **mutation**. `final List<String> names = new ArrayList<>();` signifie que `names` pointera toujours vers la même liste, mais tu peux toujours ajouter/retirer des éléments dans cette liste. C'est identique au comportement de Python avec les listes « assignées à une variable ».

2.3 Modificateurs d'Accès — La Visibilité Structurée

En Python, tout est public par convention. Le underscore préfixe (`_private`) est une **convention** non enforcée. En Java, la visibilité est contrôlée par des modificateurs d'accès **enforcés par le compilateur**.

Modificateur	Classe	Package	Sous-classe	Monde	Python équivalent
public	Oui	Oui	Oui	Oui	Attribut normal
protected	Oui	Oui	Oui	Non	Pas d'équivalent direct
(aucun) — package-private	Oui	Oui	Non	Non	Pas d'équivalent
private	Oui	Non	Non	Non	_underscore (convention)

Dans un codebase Spring Boot bancaire, tu verras principalement : `private` pour les champs des entités et services (encapsulation), `public` pour les méthodes de service et les endpoints API, `protected` pour les méthodes destinées à être surchargées dans des sous-classes. Le **package-private** (aucun modificateur) est le défaut quand tu ne spécifies rien — visible uniquement dans le même package.

2.4 Classes, Méthodes et Constructeurs

La classe Java est l'unité fondamentale de code. Contrairement à Python où tu peux écrire des fonctions libres dans un module, en Java **tout code doit vivre dans une classe**. Il n'y a pas de fonctions indépendantes.

2.4.1 Anatomie d'une Classe

Une classe Java contient, dans cet ordre conventionnel : les champs (fields/attributs), les constructeurs, les méthodes. Chaque élément a un modificateur d'accès et un type. Les champs sont généralement `private` avec des `public` getters/setters — c'est le pattern **JavaBeans** que tu as vu dans tes entités JPA (`getFirstName()`, `setFirstName()`).

En Python, le `__init__` est le constructeur. En Java, le constructeur est une méthode spéciale qui porte **le même nom que la classe** et n'a **pas de type de retour** (même pas void). Java supporte la **surcharge de constructeurs** (overloading) : plusieurs constructeurs avec des paramètres différents. Tu as vu ça dans tes entités JPA : un constructeur vide (obligatoire pour JPA) et un constructeur avec paramètres.

2.4.2 this — La Référence à l'Instance

En Python, `self` est un paramètre explicite de chaque méthode d'instance. En Java, `this` est une référence implicite à l'instance courante — tu ne le déclares pas dans la signature de la méthode. Tu l'utilises pour lever l'ambiguïté quand un paramètre a le même nom qu'un champ : `this.firstName = firstName;` dans un constructeur. Si les noms sont différents, `this` est optionnel.

2.4.3 static — Appartient à la Classe, Pas à l'Instance

En Python, une méthode de classe se décore avec `@classmethod` ou `@staticmethod`. En Java, le mot-clé `static` indique qu'un champ ou une méthode appartient à la **classe elle-même**, pas à une instance. Exemple courant : `public static void main(String[] args)` — le point d'entrée du programme est static car il est appelé avant qu'aucune instance n'existe.

Dans Spring Boot, tu verras `static` rarement sauf pour : les constantes de classe (`private static final String API_VERSION = "v1";`), les méthodes utilitaires, et les factory methods.

2.5 Interfaces et Classes Abstraites

C'est ici que Java diverge le plus de Python, et c'est **critique** pour comprendre Spring Boot. Java n'a pas de duck typing. Le polymorphisme passe entièrement par les **interfaces** et l'**héritage de classes**.

2.5.1 Interfaces — Le Contrat

Une interface Java est un **contrat** : elle définit **quelles méthodes** une classe doit implémenter, sans fournir le **comment**. C'est l'équivalent le plus proche des `Protocol` de Python

(`typing.Protocol`), mais avec une différence fondamentale : en Java, la classe doit **explicitement déclarer** qu'elle implémente l'interface avec le mot-clé `implements`.

En Python, si un objet a une méthode `read()` et une méthode `write()`, il est traitable comme un fichier (duck typing). En Java, la classe doit déclarer `implements Readable, Writable` sinon le compilateur refuse. C'est plus rigide mais plus sûr : tu sais à la compilation que le contrat est respecté.

Pourquoi c'est crucial pour Spring Boot : Toute l'architecture Spring repose sur les interfaces. Un `CreditApplicationRepository` est défini comme une interface. Spring fournit l'implémentation au runtime via l'injection de dépendances. Tu programmes **contre l'interface**, pas contre l'implémentation — c'est le principe fondamental du découplage.

2.5.2 Classes Abstraites — Le Template

Une classe abstraite est un hybride : elle peut contenir des méthodes implémentées ET des méthodes abstraites (sans corps). Elle ne peut pas être instanciée directement — il faut en hériter. C'est l'équivalent de `abc.ABC` avec `@abstractmethod` en Python.

Règle Java : Une classe ne peut hériter que d'**une seule classe** (pas d'héritage multiple), mais peut implémenter **plusieurs interfaces**. C'est le choix de design de Java pour éviter le «diamond problem». En Python, l'héritage multiple est autorisé avec le MRO (Method Resolution Order).

Concept	Python	Java
Interface (contrat pur)	Protocol (typing.Protocol)	interface
Implémentation	Implicite (duck typing)	class X implements Y
Classe abstraite	abc.ABC + @abstractmethod	abstract class
Héritage simple	class Child(Parent):	class Child extends Parent
Héritage multiple	class C(A, B): (autorisé)	NON AUTORISÉ (interfaces multiples à la place)
Méthode par défaut	Implémentation dans Protocol	default method dans interface (Java 8+)

2.6 Génériques (Generics) — Le Système de Types Paramétrés

En Python, `list[int]` est un type hint optionnel. En Java, `List<Integer>` est un **contrat enforcé par le compilateur**. Les génériques sont omniprésents dans Spring Boot.

Le concept : un générique est une classe ou interface paramétrée par un type. `List<String>` est une liste qui ne peut contenir QUE des String. `List<CreditApplication>` ne contient que

des CreditApplication. Le compilateur vérifie à la compilation que tu ne mets pas un Integer dans une List<String>.

Dans Spring Boot bancaire, tu verras constamment :

`Optional<Client>` — un conteneur qui peut ou non contenir un Client (l'équivalent de `Optional` en Python / `Client | None`)

`List<CreditApplication>` — une liste d'applications de crédit

`Map<String, Object>` — l'équivalent de `dict[str, Any]` en Python

`ResponseEntity<ClientDTO>` — une réponse HTTP contenant un DTO client

`Page<CreditApplication>` — une page de résultats paginés

2.6.1 Wildcards — La Flexibilité des Génériques

Les wildcards permettent de travailler avec des types génériques de manière flexible :

`? extends Number` — **Upper Bounded**. Accepte Number ou toute sous-classe (Integer, Double, BigDecimal). Lecture seulement. L'équivalent de `T_co` (covariant) en Python.

`? super Integer` — **Lower Bounded**. Accepte Integer ou toute super-classe (Number, Object). Écriture possible.

`?` — **Unbounded**. Accepte n'importe quel type. Utile quand le type spécifique n'importe pas.

2.7 Enums — Tu les Connais Déjà

Les enums Java sont des classes à part entière. Tu les as déjà utilisées dans tes entités JPA : `RiskLevel`, `EmploymentStatus`, `ApplicationStatus`. La différence avec les enums Python : les enums Java peuvent avoir des champs, des constructeurs, et des méthodes. Elles ne sont pas simplement des constantes nommées — ce sont des objets complets.

Exemple bancaire : un enum `RiskLevel` pourrait avoir un champ `minScore` et un champ `maxScore`, avec un constructeur qui les initialise. Cela encapsule la logique métier directement dans l'enum : pas besoin de chercher dans le code quelle plage de scores correspond à quel niveau de risque.

2.8 Records (Java 16+) — L'Équivalent des Dataclasses

Si tu connais les `@dataclass` Python ou les modèles `Pydantic BaseModel`, tu comprendras immédiatement les Records Java. Un Record est une classe immuable dont le compilateur génère automatiquement : le constructeur, les accesseurs (getters), `equals()`, `hashCode()`, et `toString()`.

Dans Spring Boot bancaire, les Records sont utilisés principalement pour les **DTOs** (Data Transfer Objects) — les objets qui transportent des données entre les couches (API → Service → Réponse). `record ClientResponse(Long id, String name, BigDecimal income)` génère tout ce dont tu as besoin pour renvoyer des données au client HTTP.

Concept	Python	Java
Classe de données	@dataclass / BaseModel	record
Immutabilité	@dataclass(frozen=True)	record (toujours immuable)
Accesseurs	Direct (obj.name)	obj.name() (méthode)
equals/hash auto	__eq__ et __hash__ générés	equals() et hashCode() générés
Validation	Pydantic validators	Compact constructors

2.9 Exceptions — Checked vs Unchecked

C'est un concept clé que Python n'a pas. Java distingue deux types d'exceptions :

Checked Exceptions (héritent de `Exception`) — Le compilateur OBLIGE le code appelant à les gérer (avec try-catch ou throws). Exemples : `IOException`, `SQLException`. L'idée est de forcer le développeur à réfléchir aux cas d'erreur. En pratique, c'est souvent perçu comme verbeux.

Unchecked Exceptions (héritent de `RuntimeException`) — Pas d'obligation de les gérer. Exemples : `NullPointerException`, `IllegalArgumentException`. Ce sont les erreurs « de programmation » qui ne devraient pas arriver si le code est correct.

Dans Spring Boot, la plupart des exceptions sont **unchecked**. Spring convertit les checked exceptions (comme les exceptions JDBC) en unchecked exceptions pour simplifier le code. Tu verras rarement des blocs try-catch dans un codebase Spring Boot bien écrit — les exceptions sont gérées globalement par un `@ControllerAdvice`.

2.10 Annotations — Le Langage dans le Langage

Les annotations Java (préfixées par `@`) sont le mécanisme le plus important à comprendre pour Spring Boot. Elles sont l'équivalent enrichi des décorateurs Python, mais avec une différence fondamentale : les annotations sont des **métadonnées** attachées au code, lisibles à la compilation et/ou au runtime par des frameworks.

Tu les connais déjà de JPA : `@Entity`, `@Id`, `@Column`, `@ManyToOne`. Ces annotations ne modifient pas le comportement de la classe Java elle-même — elles sont lues par Hibernate au runtime pour configurer le mapping objet-relationnel.

Spring Boot pousse ce concept encore plus loin. Quasiment toute la configuration passe par les annotations. Comprendre une annotation Spring, c'est comprendre ce que le framework fait « en coulisses » quand il la rencontre.

Annotation	Contexte	Ce qu'elle fait	Équiv. Python approximatif
@Override	Méthode	Vérifie que la méthode surcharge bien une méthode parent	Pas d'équivalent (duck typing)
@Deprecated	Classe/Méthode	Marque comme obsolète, warning à la compilation	warnings.warn(DeprecationWarning)
@FunctionalInterface	Interface	Vérifie que l'interface a exactement une méthode abstraite	Callable / Protocol
@SuppressWarnings	Tout	Supprime des warnings spécifiques du compilateur	# noqa / # type: ignore

2.11 Lambdas et Streams (Java 8+) — Le Fonctionnel en Java

Depuis Java 8, le langage supporte les **expressions lambda** et l'**API Streams** pour le traitement fonctionnel des collections. Si tu connais les list comprehensions et les fonctions `map`/`filter`/`reduce` de Python, tu comprendras immédiatement.

Une lambda Java s'écrit : `(paramètres) -> expression` ou `(paramètres) -> { bloc }`. C'est l'équivalent de `lambda x: expression` en Python, mais sans la limitation d'une seule expression — tu peux mettre un bloc complet.

L'API Streams est l'équivalent de la programmation fonctionnelle sur les collections. Tu crées un stream à partir d'une collection, tu chaînes des opérations intermédiaires (`filter`, `map`, `sorted`), et tu termines par une opération terminale (`collect`, `forEach`, `count`). C'est très similaire au chaînage Pandas/DFlib que tu connais.

Opération	Python	Java Stream
Filtrer	[x for x in lst if x > 5] / filter()	.filter(x -> x > 5)
Transformer	[f(x) for x in lst] / map()	.map(x -> f(x))
Trier	sorted(lst, key=...)	.sorted(Comparator.comparing(...))
Réduire	functools.reduce()	.reduce(identity, accumulator)
Collecter en liste	list(result)	.collect(Collectors.toList())
Collecter en map	dict(result)	.collect(Collectors.toMap(...))

Opération	Python	Java Stream
Compter	len(list(...))	.count()
Premier élément	next(iter(...))	.findFirst()
Vérifier tout	all(cond for x)	.allMatch(cond)
Vérifier au moins un	any(cond for x)	.anyMatch(cond)

2.12 Optional (Java 8+) — Gestion Explicite du Null

`Optional<T>` est un conteneur qui peut contenir une valeur de type T ou être vide. C'est la réponse de Java au «million dollar mistake» de null. En Python, l'équivalent est `T | None` avec les type hints, mais ce n'est qu'une indication — `Optional` en Java force le code appelant à gérer le cas « absent » explicitement.

Dans Spring Boot bancaire, tu verras `Optional` partout dans les repositories : `Optional<Client> findByEmail(String email);`. Le résultat peut être présent (client trouvé) ou absent (pas de client avec cet email). Tu ne peux pas accéder à la valeur sans vérifier d'abord qu'elle est présente : `.isPresent()`, `.orElseThrow()`, `.orElse(default)`, `.map()`.

Partie III — Maven : Construire un Projet Java de Zéro

Maven est à Java ce que `uv` / `pip` + `pyproject.toml` est à Python, mais en beaucoup plus large. C'est à la fois un gestionnaire de dépendances, un système de build, un framework de cycle de vie du projet, et un système de plugins.

3.1 Le pom.xml — Le Coeur du Projet

Le `pom.xml` (Project Object Model) est l'équivalent de ton `pyproject.toml`. Il définit : les coordonnées du projet (groupId, artifactId, version), les dépendances, les plugins de build, et les propriétés (version Java, etc.).

Coordonnées Maven — Chaque artefact Java est identifié par un triplet unique :

`groupId` — L'organisation (`com.tekkod.banking`). Équivalent du namespace PyPI.

`artifactId` — Le nom du projet (`credit-scoring-service`). Équivalent du nom du package PyPI.

`version` — La version (`1.0.0-SNAPSHOT`). SNAPSHOT indique une version en développement.

3.1.1 Les Scopes de Dépendances

Maven a un concept que Python n'a pas : les **scopes** de dépendances. Ils contrôlent quand une dépendance est disponible dans le classpath.

Scope	Compilation	Runtime	Test	Python équivalent
compile (défaut)	Oui	Oui	Oui	dependencies dans pyproject.toml
runtime	Non	Oui	Oui	Driver DB (présent au runtime, pas compilé contre)
test	Non	Non	Oui	[project.optional-dependencies] test = [...]
provided	Oui	Non	Oui	Fourni par le serveur d'application

3.2 Le Cycle de Vie Maven

Maven définit un cycle de vie standardisé en phases. Chaque phase exécute toutes les phases précédentes. Si tu lances `mvn package`, Maven exécute : validate → compile → test → package.

Phase Maven	Action	Équivalent Python
validate	Vérifie que le pom.xml est valide	uv lock --check
compile	Compile les sources Java en .class	python -m py_compile (jamais utilisé manuellement)
test	Exécute les tests unitaires	pytest
package	Crée le JAR/WAR	python -m build / uv build
install	Installe le JAR dans le cache local ~/.m2	pip install -e .
deploy	Publie sur un repo distant (Nexus, Artifactory)	twine upload / uv publish

3.3 Structure Standard d'un Projet Maven

Maven impose une structure de dossiers conventionnelle. Tout projet Maven suit cette arborescence. Ne pas la respecter est possible (via configuration) mais fortement déconseillé — c'est «convention over configuration».

Répertoire	Contenu	Équivalent Python
src/main/java/	Code source Java (production)	src/ ou package_name/
src/main/resources/	Fichiers de config (application.yml, etc.)	configs, .env files
src/test/java/	Code source des tests	tests/
src/test/resources/	Fichiers de config pour les tests	tests/fixtures/
target/	Artefacts générés (compilés, JAR)	dist/ ou build/
pom.xml	Configuration du projet	pyproject.toml

3.4 Le JAR — L'Unité de Déploiement

Un **JAR** (Java ARchive) est un fichier ZIP contenant les fichiers .class compilés, les ressources, et un fichier MANIFEST.MF. C'est l'équivalent d'un `.whl` Python (wheel). Quand tu fais `mvn package` sur un projet Spring Boot, Maven crée un **Fat JAR** (ou Uber JAR) : un JAR qui contient TOUTES les dépendances, le serveur Tomcat embarqué, et ton code. Tu peux l'exécuter directement avec `java -jar app.jar`.

C'est l'équivalent de mettre ton application FastAPI + Uvicorn + toutes les dépendances dans un seul fichier exécutable. Pas besoin d'installer un serveur web séparément — tout est embarqué.

Partie IV — Spring Framework : L'Écosystème

Spring est le framework Java le plus utilisé en entreprise. Il est à Java ce que Django est à Python — mais en beaucoup plus vaste et modulaire. Pour comprendre un codebase bancaire Spring Boot, tu dois d'abord comprendre les concepts fondamentaux du Spring Framework, sur lesquels Spring Boot est construit.

4.1 Inversion of Control (IoC) et Injection de Dépendances (DI)

C'est **LE concept** à comprendre. Tout le reste découle de là.

En Python/FastAPI, quand ton endpoint a besoin d'un service, tu fais généralement l'une de ces choses : tu importes et instancies directement (`service = CreditService()`), ou tu utilises le système `Depends()` de FastAPI pour l'injection.

En Spring, tu ne crées **JAMAIS** tes objets toi-même avec `new` pour les composants du framework. C'est le **conteneur Spring (IoC Container)** qui crée, configure, et gère le cycle de vie de tes objets. Ces objets gérés par Spring s'appellent des **Beans**.

L'**Injection de Dépendances** (DI) est le mécanisme par lequel Spring fournit automatiquement les dépendances dont un Bean a besoin. Si ton `CreditService` a besoin d'un `CreditApplicationRepository`, tu déclares cette dépendance dans le constructeur, et Spring l'injecte automatiquement au démarrage.

> **Analogie FastAPI**
> Le système Depends() de FastAPI EST de l'injection de dépendances. Quand tu écris def endpoint(db: Session = Depends(get_db)), FastAPI injecte la session. Spring fait la même chose mais de manière plus généralisée : TOUS les composants de ton application sont injectés, pas seulement les dépendances des endpoints.

4.1.1 Les Annotations de Définition des Beans

Spring utilise des annotations pour identifier quelles classes doivent devenir des Beans gérés :

Annotation	Usage	FastAPI/Python équivalent
@Component	Bean générique	Classe instanciée par le framework
@Service	Couche logique métier	Classe de service (convention)
@Repository	Couche accès données	Repository pattern (comme tu l'as vu)
@Controller / @RestController	Couche présentation (endpoints HTTP)	Routeur FastAPI (@app.get, @app.post)

Annotation	Usage	FastAPI/Python équivalent
@Configuration	Classe de configuration	settings.py / config.py
@Bean	Méthode dans @Configuration qui crée un Bean	Fonction factory dans Depends()

Hiérarchie des stéréotypes : `@Service`, `@Repository` et `@Controller` sont des spécialisations de `@Component`. Techniquement, Spring les traite de la même manière (il les enregistre comme Beans). La différence est **sémantique** : elle communique le rôle de la classe dans l'architecture. Un développeur lisant le code sait immédiatement qu'un `@Repository` accède aux données et qu'un `@Service` contient la logique métier.

4.1.2 Les Mécanismes d'Injection

Il existe trois façons d'injecter des dépendances en Spring. La **recommandée** est l'injection par constructeur :

Injection par constructeur (recommandée) — Les dépendances sont des paramètres du constructeur. Le Bean est immuable une fois créé. Si la classe n'a qu'un seul constructeur, l'annotation `@Autowired` est optionnelle (Spring l'infère). C'est le style que tu verras dans les codebases modernes.

Injection par champ (déconseillée mais courante) — `@Autowired` directement sur un champ `private`. Plus concis mais rend les tests plus difficiles car on ne peut pas fournir facilement des mocks.

Injection par setter (rare) — `@Autowired` sur une méthode setter. Utile uniquement pour les dépendances optionnelles.

4.2 Spring Boot — Spring avec Batteries Incluses

Spring Framework seul demande beaucoup de configuration. Spring Boot est une surcouche qui applique le principe d'**auto-configuration** : il détecte les dépendances présentes dans le classpath et configure automatiquement les Beans nécessaires.

Exemple concret : si `spring-boot-starter-data-jpa` est dans ton `pom.xml` et qu'un driver MySQL est présent, Spring Boot configure **automatiquement** : un DataSource (pool de connexions HikariCP), un EntityManagerFactory (Hibernate), un TransactionManager, et le classpath scanning pour trouver tes entités `@Entity`.

C'est l'équivalent de ce que tu fais manuellement dans ton `config.py` avec `create_engine()` et `sessionmaker()` — sauf que Spring Boot le fait tout seul en lisant tes propriétés dans `application.yml`.

4.2.1 Les Starters — Packs de Dépendances

Un **starter** est une dépendance Maven qui regroupe tout ce qu'il faut pour un cas d'usage. C'est l'équivalent d'un méta-package.

Starter	Inclut	Python équivalent
spring-boot-starter-web	Spring MVC + Tomcat embarqué + Jackson (JSON)	FastAPI + Uvicorn + Pydantic
spring-boot-starter-data-jpa	Spring Data JPA + Hibernate + HikariCP	SQLAlchemy + pymysql + sessionmaker
spring-boot-starter-validation	Jakarta Validation (Bean Validation)	Pydantic validators
spring-boot-starter-security	Spring Security (auth, JWT, etc.)	FastAPI security + python-jose
spring-boot-starter-test	JUnit 5 + Mockito + Spring Test	pytest + unittest.mock

4.2.2 application.yml — La Configuration Centralisée

Toute la configuration de l'application vit dans `src/main/resources/application.yml` (ou `application.properties`). C'est l'équivalent de ton fichier `.env` + la configuration SQLAlchemy + les settings de l'application, le tout centralisé.

Les propriétés clés pour le bancaire : `spring.datasource.url` (URL de connexion DB), `spring.jpa.hibernate.ddl-auto` (gestion du schéma — toujours `validate` ou `none` en production, jamais `create` ou `update`), `spring.jpa.show-sql` (pour le debug — équivalent de `echo=True` dans SQLAlchemy).

4.2.3 Profiles — Configuration par Environnement

Spring Boot supporte les **profiles** pour varier la configuration selon l'environnement : `application-dev.yml` pour le développement, `application-prod.yml` pour la production. On active un profil avec `spring.profiles.active=dev`. C'est l'équivalent de tes fichiers `.env.dev` / `.env.prod` mais intégré au framework.

Partie V — Architecture en Couches d'un Projet Spring Boot Bancaire

C'est la partie la plus importante pour ton objectif : lire et comprendre un codebase Spring Boot bancaire. L'architecture suit un pattern en couches strict où chaque couche a une responsabilité unique et ne communique qu'avec ses voisines directes.

5.1 La Vue d'Ensemble — Flux d'une Requête HTTP

Quand un client HTTP envoie une requête (par exemple POST /api/credit-applications), voici le chemin qu'elle suit à travers les couches :

1. Controller — Reçoit la requête HTTP, valide l'entrée (via DTOs), et délègue au Service. Ne contient JAMAIS de logique métier. Équivalent : tes fonctions de route FastAPI.

2. Service — Contient la logique métier : calcul du credit score, détermination du risk level, application des règles de pricing. Orchestre les appels aux repositories. Équivalent : ton CreditScoringAnalyzer, mais pour les opérations transactionnelles.

3. Repository — Interface d'accès aux données. Spring Data génère l'implémentation automatiquement. Équivalent : tes repositories SQLAlchemy.

4. Entity — Les entités JPA que tu connais déjà (Client, CreditApplication, LoanProduct).

5. DTO — Les objets de transfert qui séparent la représentation interne (entité) de la représentation externe (API). Équivalent : tes schémas Pydantic.

Couche Spring Boot	Annotation	Responsabilité	FastAPI équivalent
Controller	@RestController	Reçoit/renvoie HTTP, validation	@app.get/@app.post
Service	@Service	Logique métier, transactions	Fonctions de service
Repository	@Repository / interface	CRUD et requêtes DB	Repository SQLAlchemy
Entity	@Entity	Mapping ORM	Modèle SQLAlchemy (Base)
DTO	record / class	Transfert de données API	Pydantic BaseModel
Mapper	MapStruct / manuel	Conversion Entity ↔ DTO	model_validate / model_dump
Exception Handler	@ControllerAdvice	Gestion globale des erreurs	Exception handlers FastAPI
Config	@Configuration	Beans custom, Security	settings.py, dependencies

5.2 La Couche Controller — Anatomie Détaillée

Le Controller est la porte d'entrée HTTP de ton application. En Spring Boot, il est annoté `@RestController` (qui combine `@Controller` et `@ResponseBody` — toutes les réponses sont automatiquement sérialisées en JSON).

5.2.1 Les Annotations de Mapping HTTP

Annotation Spring	Méthode HTTP	FastAPI équivalent	Usage bancaire
@GetMapping("/clients")	GET	@app.get("/clients")	Lister les clients
@PostMapping("/applications")	POST	@app.post("/applications")	Soumettre une demande
@PutMapping("/clients/{id}")	PUT	@app.put("/clients/{id}")	Mettre à jour un client
@DeleteMapping("/clients/{id}")	DELETE	@app.delete("/clients/{id}")	Supprimer un client
@PatchMapping("/applications/{id}")	PATCH	@app.patch("/applications/{id}")	Mise à jour partielle
@RequestMapping("/api/v1")	Préfixe de classe	@app.include_router(prefix=...)	Préfixe d'URL

5.2.2 Les Annotations de Paramètres

Annotation Spring	Source	FastAPI équivalent
@PathVariable Long id	URL path (/clients/{id})	id: int (path parameter)
@RequestParam String status	Query string (?status=APPROVED)	status: str = Query(...)
@RequestBody CreateAppDTO dto	Corps JSON de la requête	dto: CreateAppDTO (Pydantic model)
@RequestHeader String token	En-tête HTTP	token: str = Header(...)
@Valid	Déclenche la validation Bean	Validation Pydantic (automatique)

5.3 La Couche Service — Le Coeur Métier

La couche Service est où vit la logique métier bancaire : calculer un credit score, déterminer le risk level, appliquer les règles de pricing, vérifier les limites d'exposition, déclencher les workflows d'approbation.

L'annotation `@Transactional` est cruciale ici. Elle définit les frontières transactionnelles. Quand une méthode annotée `@Transactional` est appelée, Spring ouvre une transaction. Si la méthode se termine normalement, la transaction est commitée. Si une exception est levée, la transaction est rollback. C'est l'équivalent du context manager `with session.begin():` dans SQLAlchemy.

Règle bancaire importante : En contexte bancaire, `@Transactional` est non-négociable. Une opération qui crée une demande de crédit ET met à jour l'exposition du client doit être atomique : soit les deux réussissent, soit aucune ne persiste. Il n'y a pas de demi-mesure.

5.4 La Couche Repository — Spring Data JPA

C'est là que Spring Boot brêle. **Spring Data JPA** génère automatiquement l'implémentation des repositories à partir d'interfaces. Tu définis une interface qui étend `JpaRepository<Entity, IdType>`, et Spring génère au runtime toutes les méthodes CRUD standard.

5.4.1 Méthodes Héritées Automatiquement

Méthode Spring Data	SQL généré	SQLAlchemy équivalent
save(entity)	INSERT ou UPDATE	session.add(entity) + commit
findById(id)	SELECT ... WHERE id = ?	session.get(Entity, id)
findAll()	SELECT * FROM table	session.execute(select(Entity)).scalars().all()
deleteById(id)	DELETE FROM table WHERE id = ?	session.delete(entity) + commit
count()	SELECT COUNT(*) FROM table	session.execute(select(func.count(Entity.id)))
existsById(id)	SELECT EXISTS(...)	session.get(Entity, id) is not None

5.4.2 Derived Query Methods — Le Nommage Magique

Spring Data JPA peut générer des requêtes à partir du **nom de la méthode**. C'est le concept le plus « magique » de Spring Data et celui qui déroute le plus les développeurs Python. Le nom de la méthode EST la requête.

Méthode déclarée dans l'interface	SQL généré automatiquement
findByStatus(ApplicationStatus s)	SELECT ... WHERE status = ?
findByRiskLevelAndStatus(RiskLevel r, ApplicationStatus s)	SELECT ... WHERE risk_level = ? AND status = ?
findByClientAnnualIncomeGreaterThan(BigDecimal min)	SELECT ... WHERE client.annual_income > ?
findByCreditScoreBetween(int min, int max)	SELECT ... WHERE credit_score BETWEEN ? AND ?
countByRiskLevel(RiskLevel r)	SELECT COUNT(*) ... WHERE risk_level = ?
findByStatusOrderByCreditScoreDesc(ApplicationStatus s)	SELECT ... WHERE status = ? ORDER BY credit_score DESC
findTop10ByOrderByCreditScoreDesc()	SELECT ... ORDER BY credit_score DESC LIMIT 10

Les mots-clés reconnus par Spring Data dans les noms de méthodes : `findBy`, `countBy`, `existsBy`, `deleteBy` pour le préfixe. `And`, `Or`, `Between`, `LessThan`, `GreaterThan`, `Like`, `In`, `OrderBy`, `Top`/`First` pour les opérateurs. Le framework parse le nom, identifie les champs de l'entité (y compris à travers les relations comme `ClientAnnualIncome` = `client.annualIncome`), et génère le JPQL.

5.4.3 @Query — Requêtes Personnalisées

Pour les requêtes complexes que le nommage ne peut pas exprimer (comme tes requêtes analytiques de credit scoring), tu utilises l'annotation `@Query` avec du JPQL ou du SQL natif. C'est l'équivalent de tes statements SQLAlchemy Core dans `AnalyticalQueries`.

5.5 Les DTOs et la Séparation des Préoccupations

Dans un codebase bancaire, tu ne renvoies **JAMAIS** une entité JPA directement dans une réponse HTTP. Les raisons : l'entité peut contenir des données sensibles (mot de passe hashé, score interne), le lazy loading peut déclencher des requêtes inattendues lors de la sérialisation, et tu veux contrôler précisément la forme de la réponse.

La solution : les **DTOs** (Data Transfer Objects). En Python/FastAPI, tu as les schémas Pydantic (`ClientResponse`, `CreateApplicationRequest`). En Spring Boot, tu utilises des **Records** (Java 16+) ou des classes dédiées.

Le flux complet : `JSON Request → DTO (validation) → Service (logique) → Entity (persistence) → DTO (réponse) → JSON Response`. La conversion Entity ↔ DTO se fait via un **Mapper** (manuellement ou avec MapStruct).

5.6 Gestion des Erreurs — @ControllerAdvice

En FastAPI, tu définis des exception handlers avec `@app.exception_handler(MyException)`. En Spring Boot, le mécanisme équivalent est `@ControllerAdvice` combiné avec `@ExceptionHandler`.

Un `@ControllerAdvice` est un Bean Spring qui intercepte les exceptions levées par n'importe quel Controller. Chaque méthode annotée `@ExceptionHandler(SpecificException.class)` gère un type d'exception spécifique. Dans un contexte bancaire, tu trouveras typiquement : `ClientNotFoundException`, `CreditLimitExceededException`, `InsufficientScoreException`, `DuplicateApplicationException`.

Partie VI — Spring Data JPA en Profondeur pour le Bancaire

Tu connais déjà JPA/Hibernate depuis tes études ORM. Spring Data JPA est une **surcouche** qui simplifie l'utilisation de JPA dans le contexte Spring. Voyons les concepts supplémentaires que Spring Data ajoute.

6.1 Les Interfaces de Repository — La Hiérarchie

Spring Data fournit une hiérarchie d'interfaces de repository, chacune ajoutant des fonctionnalités :

Interface	Hérite de	Ajoute
Repository<T, ID>	(racine)	Marqueur vide — rien
CrudRepository<T, ID>	Repository	save, findById, findAll, delete, count, existsById
ListCrudRepository<T, ID>	CrudRepository	Retourne des List au lieu d'Iterable
PagingAndSortingRepository<T, ID>	CrudRepository	findAll(Pageable), findAll(Sort)
JpaRepository<T, ID>	ListCrud + PagingAndSorting	flush, saveAndFlush, deleteInBatch, getById

En pratique, tu étends presque toujours `JpaRepository` car il donne accès à tout. La déclaration ressemble à : `public interface ClientRepository extends JpaRepository<Client, Long>`. Le premier type générique est l'entité, le second est le type de la clé primaire.

6.2 Specifications — Requêtes Dynamiques

Pour les requêtes dynamiques (filtres optionnels, recherche multi-critères), Spring Data fournit les **Specifications**. C'est l'équivalent de construire des requêtes SQLAlchemy Core en chaînant des conditions.

Une Specification est une interface fonctionnelle qui encapsule un critère de filtrage. Tu peux les combiner avec `and()` et `or()`. L'avantage : chaque Specification est réutilisable et testable indépendamment. C'est le Criteria API de JPA rendu composable.

6.3 Projections — Sélectionner des Colonnes Spécifiques

Par défaut, Spring Data retourne l'entité complète. Pour les requêtes analytiques où tu n'as besoin que de quelques colonnes, tu peux utiliser des **projections** : des interfaces ou records

qui ne contiennent que les champs nécessaires. C'est l'équivalent de `select(Client.first_name, Client.annual_income)` dans SQLAlchemy Core au lieu de `select(Client)`.

6.4 Auditing — Traçabilité Bancaire

Dans le bancaire, chaque modification doit être traçable : qui a créé/modifié quoi et quand. Spring Data JPA fournit l'**auditing** automatique via des annotations : `@CreatedDate`, `@LastModifiedDate`, `@CreatedBy`, `@LastModifiedBy`. Ces champs sont remplis automatiquement par Spring lors des opérations de persistance.

Tu mets ces annotations sur des champs de ton entité, et tu actives le mécanisme avec `@EnableJpaAuditing` sur ta classe de configuration. Spring peuple les dates automatiquement, et pour les `@CreatedBy`/`@LastModifiedBy`, il utilise le `SecurityContext` de Spring Security pour identifier l'utilisateur courant.

Partie VII — Lire un Codebase Bancaire Spring Boot : Guide Pratique

Cette partie est ton guide de référence pour naviguer dans un codebase Spring Boot bancaire. Quand tu ouvres le projet pour la première fois, voici comment le lire systématiquement.

7.1 Première Lecture : La Structure du Projet

Commence par le `pom.xml`. Les dépendances te disent immédiatement ce que l'application fait : si tu vois `spring-boot-starter-data-jpa`, c'est une application qui accède à une base de données. `spring-boot-starter-security` signifie de l'authentification. `spring-boot-starter-actuator` signifie du monitoring. Les starters sont la table des matières de l'application.

Ensuite, regarde `application.yml` (ou `application.properties`). Tu y trouveras : l'URL de la base de données, le port du serveur, les propriétés JPA, les secrets (ou références à des secrets), et les configurations métier custom.

7.2 Deuxième Lecture : Les Entités

Les entités (`@Entity`) définissent le modèle de données — c'est le coeur du domaine métier. Regarde les champs, les types, les relations (`@ManyToOne`, `@OneToMany`), et les enums. Tu sais déjà lire ça grâce à tes études JPA.

Dans un contexte bancaire, fais attention à : les `BigDecimal` pour les montants, les `@Enumerated(EnumType.STRING)` (jamais ORDINAL), les `@Temporal` ou `LocalDate`/`LocalDateTime` pour les dates, et les annotations d'auditing.

7.3 Troisième Lecture : Les Repositories

Les interfaces Repository te montrent quelles requêtes l'application effectue. Les derived query methods dans les noms de méthodes te disent les critères de recherche. Les `@Query` te montrent les requêtes complexes. Cela te donne une carte des interactions entre l'application et la base de données.

7.4 Quatrième Lecture : Les Services

Les classes `@Service` contiennent la logique métier. C'est là que tu comprends le **quoi** et le **comment** de l'application. Regarde les méthodes publiques — elles représentent les **cas d'usage** de l'application. `submitCreditApplication()`, `evaluateCreditScore()`, `approveApplication()` te disent exactement ce que l'application permet de faire.

Fais attention aux `@Transactional` : les méthodes annotées sont des unités de travail atomiques. Si la méthode a `@Transactional(readOnly = true)`, c'est une lecture seule (performance optimisée). Si c'est `@Transactional` simple, c'est une opération d'écriture.

7.5 Cinquième Lecture : Les Controllers

Les Controllers (`@RestController`) définissent l'API REST. Chaque méthode = un endpoint. Les annotations de mapping (`@GetMapping`, `@PostMapping`) donnent l'URL et la méthode HTTP. Les paramètres (`@PathVariable`, `@RequestBody`) montrent l'entrée. Le type de retour (`ResponseEntity<ClientDTO>`) montre la sortie.

7.6 Le Pattern Complet — D'un Bout à l'Autre

Voici le flux complet d'une demande de crédit dans un système bancaire Spring Boot, couche par couche :

1. Le client HTTP envoie un POST /api/v1/credit-applications avec un JSON body.

2. Spring désérialise le JSON en un DTO (`CreateCreditApplicationRequest`) grâce à Jackson (l'équivalent de Pydantic).

3. `@Valid` déclenche la validation Bean (`@NotNull`, `@Min`, `@Max`) — si échec, 400 Bad Request.

4. Le Controller appelle `creditService.submitApplication(dto)`.

5. Le Service ouvre une transaction (`@Transactional`).

6. Le Service récupère le Client via `clientRepository.findById(dto.clientId())` → `Optional<Client>`.

7. Le Service récupère le LoanProduct via `productRepository.findByCode(dto.productCode())`.

8. Le Service calcule le credit score, détermine le risk level, calcule le taux d'intérêt.

9. Le Service crée l'entité CreditApplication et la sauvegarde via `applicationRepository.save(entity)`.

10. La transaction est commitée. Le Service retourne un DTO de réponse.

11. Le Controller retourne `ResponseEntity.created(uri).body(responseDTO)` → 201 Created + JSON.

Partie VIII — Glossaire des Annotations Spring Boot Bancaire

Référence rapide de toutes les annotations que tu croiseras dans un codebase Spring Boot bancaire, organisées par couche.

8.1 Annotations de Configuration

Annotation	Cible	Effet
@SpringBootApplication	Classe main	Combine @Configuration + @EnableAutoConfiguration + @ComponentScan
@Configuration	Classe	Définit des Beans via des méthodes @Bean
@Bean	Méthode	Le retour de la méthode est enregistré comme Bean Spring
@Value("${prop}")	Champ	Injecte une propriété depuis application.yml
@ConfigurationProperties	Classe	Mappe un bloc de propriétés vers un objet Java
@Profile("dev")	Classe/Méthode	Actif uniquement dans le profil spécifié
@EnableJpaAuditing	Config	Active le remplissage auto des @CreatedDate etc.
@EnableScheduling	Config	Active les tâches planifiées (@Scheduled)

8.2 Annotations Web / Controller

Annotation	Effet
@RestController	Combine @Controller + @ResponseBody — toutes les réponses en JSON
@RequestMapping("/api/v1/clients")	Préfixe d'URL pour toutes les méthodes du controller
@GetMapping / @PostMapping / etc.	Mappe une méthode HTTP vers une méthode Java
@PathVariable	Extrait un segment de l'URL
@RequestParam	Extrait un query parameter
@RequestBody	Désérialise le corps JSON en objet Java
@ResponseStatus(HttpStatus.CREATED)	Définit le code de réponse HTTP (201 ici)
@CrossOrigin	Active CORS pour ce controller

8.3 Annotations Service / Business

Annotation	Effet
@Service	Marque la classe comme Bean de couche métier
@Transactional	Ouvre une transaction autour de la méthode
@Transactional(readOnly = true)	Transaction read-only (optimisations Hibernate)
@Transactional(propagation = REQUIRES_NEW)	Nouvelle transaction indépendante
@Async	Exécute la méthode dans un thread séparé
@Scheduled(cron = "0 0 2 * * ?")	Tâche planifiée (ici : chaque jour à 2h)
@Cacheable("clients")	Cache le résultat (utile pour les données de référence)

8.4 Annotations JPA / Persistence

Annotation	Effet
@Entity	Marque la classe comme entité JPA (table)
@Table(name = "credit_applications")	Nom explicite de la table
@Id	Marque le champ comme clé primaire
@GeneratedValue(strategy = IDENTITY)	Auto-incrément
@Column(precision = 12, scale = 2)	Configuration de la colonne
@Enumerated(EnumType.STRING)	Stocke l'enum comme String (JAMAIS ORDINAL)
@ManyToOne(fetch = LAZY)	Relation N-1 avec lazy loading
@OneToMany(mappedBy = "client")	Relation 1-N inverse
@JoinColumn(name = "client_id")	Colonne de clé étrangère
@CreatedDate / @LastModifiedDate	Rempli automatiquement par Spring Data
@EntityListeners(AuditingEntityListener.class)	Active l'auditing sur l'entité

8.5 Annotations de Validation

Annotation	Validation	Pydantic équivalent
@NotNull	Le champ ne peut pas être null	Champ requis (pas Optional)
@NotBlank	String non null et non vide	constr(min_length=1)

Annotation	Validation	Pydantic équivalent
@Min(300) / @Max(850)	Valeur numérique min/max	Field(ge=300, le=850)
@Size(min=1, max=100)	Taille de String ou Collection	constr(min_length, max_length)
@Email	Format email valide	EmailStr
@Pattern(regexp="...")	Regex	constr(pattern="...")
@DecimalMin("0.00")	BigDecimal minimum	Field(ge=Decimal("0.00"))
@Valid	Déclenche la validation récursive	Validation automatique Pydantic
@Positive / @PositiveOrZero	Valeur strictement positive ou ≥ 0	Field(gt=0) / Field(ge=0)

8.6 Annotations de Sécurité (Spring Security)

Annotation	Effet
@EnableWebSecurity	Active Spring Security
@PreAuthorize("hasRole('ADMIN')")	Vérifie le rôle AVANT l'exécution
@PostAuthorize("returnObject.owner == principal.username")	Vérifie APRÈS l'exécution
@Secured("ROLE_ANALYST")	Version simplifiée de @PreAuthorize
@AuthenticationPrincipal UserDetails user	Injecte l'utilisateur authentifié

Partie IX — Synthèse et Prochaines Étapes

9.1 Ce que Tu Sais Déjà vs Ce que Ce Guide Ajoute

Déjà maîtrisé	Nouveau dans ce guide	Prochain niveau (pair programming)
Entités JPA (@Entity, @Id, @Column)	Architecture en couches Spring Boot	Implémenter un CRUD complet
Relations (@ManyToOne, @OneToMany)	Injection de dépendances (IoC/DI)	Configurer Spring Security JWT
JPQL et SQL natif	Derived Query Methods	Écrire des Specifications dynamiques
Enums Java + @Enumerated	DTOs (Records) + Mappers	Implémenter un ControllerAdvice
pom.xml et dépendances Maven	Starters Spring Boot	Tests d'intégration (@SpringBootTest)
Docker + MySQL	application.yml + Profiles	Déployer un JAR conteneurisé

9.2 Checklist d'Auto-Évaluation

Après lecture de ce guide, tu devrais pouvoir répondre à ces questions avec assurance :

Sur Java/JVM : Quelle est la différence entre javac et java ? Pourquoi les applications Java sont lentes au démarrage ? Qu'est-ce que le classpath ? Pourquoi utiliser BigDecimal et jamais double pour l'argent ? Quelle est la différence entre == et .equals() pour les String ?

Sur le système de types : Qu'est-ce que l'autoboxing ? Pourquoi List<int> n'existe pas ? Qu'est-ce qu'un Optional et pourquoi l'utiliser ? Comment les génériques garantissent-ils la type safety ?

Sur l'OOP Java : Quelle différence entre interface et classe abstraite ? Pourquoi Java interdit l'héritage multiple ? Qu'est-ce qu'un Record et quand l'utiliser ? Que fait l'annotation @Override ?

Sur Spring Boot : Qu'est-ce que l'injection de dépendances ? Que fait @SpringBootApplication ? Comment Spring Data génère-t-il les requêtes à partir des noms de méthodes ? Pourquoi ne pas renvoyer une entité JPA dans une réponse HTTP ? Que fait @Transactional ?

Sur l'architecture : Quel est le flux d'une requête HTTP du Controller à la base de données ? Comment lire un pom.xml pour comprendre ce que fait l'application ? Quelle est la différence entre un DTO et une Entity ?

9.3 Progression Suggérée

Phase 1 (immédiat) — Lecture passive. Lire ce document. Identifier les concepts que tu reconnais (JPA, Maven, etc.) et ceux qui sont nouveaux (DI, Spring Data, Controllers).

Phase 2 (après lecture) — Exploration. Ouvrir un codebase Spring Boot open-source bancaire ou de credit scoring. Naviguer avec la structure apprise : pom.xml → entities → repositories → services → controllers. Vérifier que tu comprends chaque annotation.

Phase 3 (quand prêt) — Pair programming. Implémenter le CreditGuard AI en Spring Boot, en utilisant les patterns décrits ici. Commencer par le CRUD simple, puis ajouter la logique métier de scoring, puis les DTOs et la validation, puis la sécurité.

Fin du Guide — Java, the JVM & Spring Boot pour Développeurs Python Bancaires