جامعة مولاي إسماعيل
UNIVERSITÉ MOULAY ISMAÏL

ENSAM
المدرسة الوطنية العليا للفنون و المهن
ÉCOLE NATIONALE SUPÉRIEURE D'ARTS ET MÉTIERS

JOBINTECH

Architecture logicielle

Ahmed Laatabi
a•laatabi{at}umi•ac•ma
ENSAM - Meknès
2025-2026

Logiciel

- Un **logiciel** (*software*) est un ensemble de **programme**s, de **donnée**s, et de **règle**s qui permettent à un appareil informatique (ordinateur, smartphone, …) de **fonctionner** et d'**exécuter** des **tâches** spécifiques. Il se compose d'une suite d'**instructions** (code), écrites dans un **langage de programmation**, qui implémentent un ou plusieurs **algorithme**s.

- Le **matériel** (*hardware*) est l'ensemble des composants physiques et électroniques d'un système informatique (carte mère, disque dur, …) qui sert de support et permet l'**exécution des logiciels**.

Système informatique = *Software* (immatériel) + *Hardware* (matériel).

- Le système informatique est l'une des composantes principales du **système d'information** (**SI**).
- Le **SI** est l'ensemble organisé de ressources matérielles, logicielles, humaines et organisationnelles permettant la collecte, le stockage, le traitement et la diffusion de l'**information** (les données) nécessaire au fonctionnement et à la prise de décision au sein d'une organisation.

Architecture logicielle

- L'**architecture logicielle** décrit, schématise et documente l'ensemble des éléments (ou composantes) d'un système informatique ainsi que leurs interactions (ou relations) en termes d'échanges et d'entrées/sorties.

- Son objectif est de définir la structure, les modèles, et les solutions (technologies) nécessaires pour répondre aux besoins du client et assurer la cohérence, la fiabilité, et l'évolutivité du système.

- Une bonne architecture logicielle doit garantir les **qualités non fonctionnelles** du système :
  - **Maintenabilité** : facilité de corriger, modifier, ou faire évoluer le logiciel.
  - **Performance** : rapidité et efficacité d'exécution.
  - **Scalabilité** : capacité à s'adapter à une charge croissante de données et de trafic.
  - **Sécurité** : protection des données et des processus.

- Les architectures logicielles modernes tendent à adopter une **séparation** en modules (ou couches) : 1) **interface utilisateur** (couche de présentation); 2) **processus métiers** (couche logique); 3) **persistance des données** (couche d'accès aux données).

Architecte logiciel

- L'**architecte logiciel** est le concepteur de haut niveau du système. Son rôle consiste à :
  - Définir la structure globale et les principes de conception du logiciel :
  - Choisir les technologies, les styles d'architecture (monolithique, microservices, ...), les patterns (modèles de conception) et les normes à suivre.
  - Coordonner entre les équipes non techniques (produit, direction) et les équipes techniques (développement, test, déploiement).
  - Garantir l'intégration et la cohérence entre les différents modules du logiciel, assurant ainsi qu'il réponde aux besoins clients et au cahier des charges.

- **Architecte logiciel** : conçoit le plan stratégique du logiciel avant sa construction. Il produit des documents et des diagrammes qui répondent au "*quoi ?*" (structure et règles).
- **Ingénieur logiciel / Développeur** : construit et implémente le logiciel en suivant ce plan. Il se concentre sur le "*comment ?*" (algorithmes, codes, et implémentation).

Principes fondamentaux

- **Principes structurels** : qualités non fonctionnelles à maximiser.
  - Séparation des préoccupations (*separation of concerns*) : le système est divisé en **modules**, chacun ayant une responsabilité unique et bien définie. Les composantes du même module doivent être fortement liées et cohérentes (*high cohesion*) → **modularité** et **maintenabilité**.
  - Modularité : les modules qui composent le système sont indépendants et peuvent être développés, testés et déployés de manière autonome et parallèle. Chaque module doit encapsuler sa complexité, qui doit être abstraite aux interactions → **évolutivité**, **interopérabilité** et **réduction des coûts**.
  - Couplage Faible (*low coupling*) : les dépendances entre les modules doivent être minimales pour éviter les erreurs inter-modules. chaque module doit pouvoir évoluer indépendamment afin de faciliter les corrections ou l'ajout de nouvelles fonctionnalités → **interopérabilité** et **scalabilité**.
- **Principes de conception** (*design principles*) : règles à suivre pour répondre aux besoins fonctionnels.
  - **KISS** (*Keep It Simple & Stupid*) : toujours privilégier la **solution la plus simple** qui **fonctionne**.
  - **DRY** (*Don't Repeat Yourself*) : les **données** et les **logiques** ne doivent **pas être dupliquées**.
  - **YAGNI** (*You Aren't Gonna Need It*) : éviter d'implémenter des **données** ou **logiques inutiles**.

SOLID

SOLID regroupe cinq principes de conception à suivre pour produire de bonnes architectures logicielles (code compréhensible, flexible et maintenable), notamment en **programmation orienté objet** :

- ***S**ingle responsibility* (responsabilité unique) : une **fonction** ou une **classe** ne doit avoir qu'**une seule responsabilité** (un seul rôle, objectif).
- ***O**pen/closed* (ouvert/fermé) : une fonction ou une classe doit être **fermée à la modification** mais **ouverte à l'extension** : ajout de nouvelles fonctionnalités sans modifier le code existant.
- ***L**iskov substitution* (substitution de Liskov) : une **instance de type de base** doit pouvoir **être remplacée** par **une instance de l'un de ses sous-types** sans altérer le **bon fonctionnement du programme**. Les sous-classes peuvent donc être utilisées de manière interchangeable avec leurs classes parentes.
- ***I**nterface segregation* (ségrégation des interfaces) : préférer la définition de plusieurs **interfaces spécifiques** plutôt qu'une **seule interface générale**. Ainsi, les classes ne dépendent que des méthodes dont elles ont besoin, ce qui réduit les couplages inutiles.
- ***D**ependency inversion* (inversion des dépendances) : il faut dépendre des **abstractions**, pas des **implémentations** (les modules de haut niveau ne doivent pas dépendre des modules de bas niveau, mais tous deux doivent dépendre d'abstractions).

Responsabilité unique

"*A class should have only one reason to change*"

- Une fonction permet de récupérer (depuis le clavier) les informations d'un étudiant (CNE, nom complet, date de naissance) puis de les enregistrer dans une BD.
- Cette fonction a deux raisons de changer (car elle a deux responsabilités distinctes) :
  - Si l'on souhaite récupérer une donnée supplémentaire (lieu de naissance).
  - Si l'on souhaite modifier le mécanisme d'enregistrement des données :
    - MySQL → MongoDB.

```
def saisir_et_enregistrer_etudiant():
    cne = input("CNE : ")
    nom_complet = input("Nom complet : ")
    date_naissance = input("Date de naissance : ")

    print("Enregistrement dans MySQL...")
    print(f"INSERT INTO etudiant VALUES ('{cne}',",
          f"'{nom_complet}', '{date_naissance}');")

saisir_et_enregistrer_etudiant()
```

```
def saisir_etudiant():
    cne = input("CNE : ")
    nom_complet = input("Nom complet : ")
    date_naissance = input("Date de naissance : ")
    return cne, nom_complet, date_naissance

def enregistrer_etudiant(cne, nom_complet, date_naissance):
    print("Enregistrement dans MySQL...")
    print(f"INSERT INTO etudiant VALUES ('{cne}',"
          f"'{nom_complet}', '{date_naissance}');")

cne, nom_complet, date_naissance = saisir_etudiant()
enregistrer_etudiant(cne, nom_complet, date_naissance)
```

?

Responsabilité unique

- Le principe de responsabilité unique (**SRP**) exige qu'un changement de nature n'affecte qu'une seule composante.
- Ici, les deux fonctions dépendent directement de la même structure de données (cne, nom_complet, date_naissance, ...).
- Si la structure des données change → il faut modifier les deux fonctions (violation du SRP).
- L'utilisation des **fonctions** simples, **couplées fortement** aux détails de la **structure de données**, limite l'implémentation correcte du principe de responsabilité unique.
- Il faut que les deux fonctions dépendent d'une interface ou d'abstraction commune (un objet), et non directement de ses champs (structure interne).

```python
class Etudiant:
    def __init__(self, cne, nom_complet, date_naissance):
        self.cne = cne
        self.nom_complet = nom_complet
        self.date_naissance = date_naissance

    # objet --> dictionnaire
    def to_dict(self):
        return {
            "cne": self.cne,
            "nom_complet": self.nom_complet,
            "date_naissance": self.date_naissance
        }


def saisir_etudiant():
    cne = input("CNE : ")
    nom_complet = input("Nom complet : ")
    date_naissance = input("Date de naissance : ")
    return Etudiant(cne, nom_complet, date_naissance)


def enregistrer_etudiant(etudiant):
    data = etudiant.to_dict()
    print("Enregistrement dans MySQL...")
    print(f"INSERT INTO etudiant VALUES ("
          f"{', '.join([repr(v) for v in data.values()])});")


etu = saisir_etudiant()
enregistrer_etudiant(etu)
```

?

Ouvert/fermé

- Une fois qu'une classe ou une fonction a été testée et validée, elle ne doit plus être modifiée, mais seulement étendue pour ajouter de nouvelles fonctionnalités.

```python
class Etudiant:
    def __init__(self, cne, nom_complet, date_naissance):
        self.cne = cne
        self.nom_complet = nom_complet
        self.date_naissance = date_naissance

    # objet --> dictionnaire
    def to_dict(self):
        return {
            "cne": self.cne,
            "nom_complet": self.nom_complet,
            "date_naissance": self.date_naissance
        }

def saisir_etudiant():
    cne = input("CNE : ")
    nom_complet = input("Nom complet : ")
    date_naissance = input("Date de naissance : ")
    return Etudiant(cne, nom_complet, date_naissance)


def enregistrer_etudiant(etudiant):
    data = etudiant.to_dict()
    print("Enregistrement dans MySQL...")
    print(f"INSERT INTO etudiant VALUES ("
          f"{', '.join([repr(v) for v in data.values()])});")
```

```python
class Etudiant2(Etudiant):
    def __init__(self, cne, nom_complet, date_naissance,
                 lieu_naissance):
        super().__init__(cne, nom_complet, date_naissance)
        self.lieu_naissance = lieu_naissance

    def to_dict(self):
        data = super().to_dict()
        data["lieu_naissance"] = self.lieu_naissance
        return data

def saisir_etudiant2():
    etud = saisir_etudiant()
    lieu_naissance = input("Lieu de naissance : ")
    return Etudiant2(etud.cne, etud.nom_complet, etud.date_naissance,
                     lieu_naissance)

etu = saisir_etudiant2()
enregistrer_etudiant(etu)
```

Substitution de Liskov

- Bonne utilisation de l'héritage : si ***G*** est un sous-type de ***T***, alors tout objet de type ***T*** peut être remplacé par un objet de type ***G*** sans altérer les propriétés désirables du programme.

  → les classes dérivées ***G*** ne doivent pas casser le code qui utilise les classes de base ***T***.

```python
class Etudiant:
    def __init__(self, cne, nom_complet, date_naissance):
        self.cne = cne
        self.nom_complet = nom_complet
        self.date_naissance = date_naissance

    # objet --> dictionnaire
    def to_dict(self):
        return {
            "cne": self.cne,
            "nom_complet": self.nom_complet,
            "date_naissance": self.date_naissance
        }

def saisir_etudiant():
    cne = input("CNE : ")
    nom_complet = input("Nom complet : ")
    date_naissance = input("Date de naissance : ")
    return Etudiant(cne, nom_complet, date_naissance)


def enregistrer_etudiant(etudiant):
    data = etudiant.to_dict()
    print("Enregistrement dans MySQL...")
    print(f"INSERT INTO etudiant VALUES ("
          f"{', '.join([repr(v) for v in data.values()])});")
```

```python
class Etudiant2(Etudiant):
    def __init__(self, cne, nom_complet, date_naissance,
                 lieu_naissance):
        super().__init__(cne, nom_complet, date_naissance)
        self.lieu_naissance = lieu_naissance

    def to_dict(self):
        data = super().to_dict()
        data["lieu_naissance"] = self.lieu_naissance
        return data

def saisir_etudiant2():
    etud = saisir_etudiant()
    lieu_naissance = input("Lieu de naissance : ")
    return Etudiant2(etud.cne, etud.nom_complet, etud.date_naissance,
                     lieu_naissance)

etu = saisir_etudiant2()
enregistrer_etudiant(etu)
```

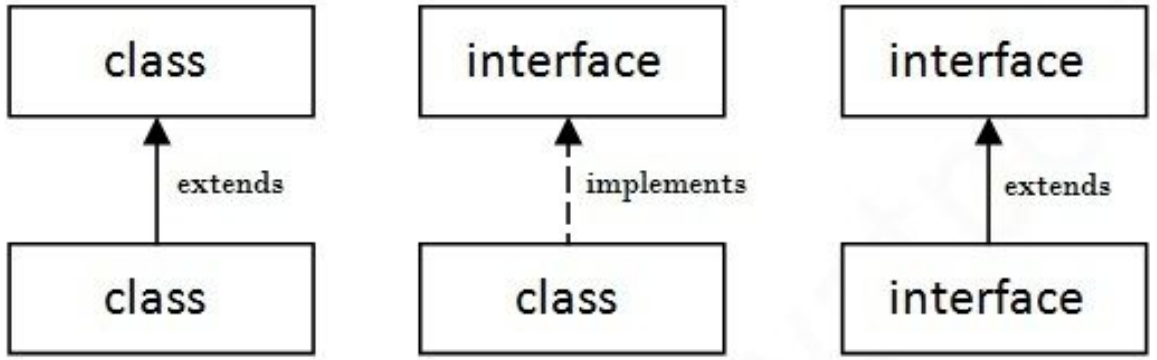
POO

- L'**héritage** permet à une classe (classe fille, classe dérivée, sous-classe) d'acquérir et réutiliser les propriétés (attributs) et les comportements (méthodes) d'une autre classe (classe mère, classe de base, super-classe), tout en ajoutant ou modifiant des fonctionnalités : class Etudiant2 (Etudiant).
- Le **polymorphisme** (multiples formes) permet de traiter des objets de types différents via une interface unique : une même méthode peut se comporter différemment selon l'objet sur lequel elle est appelée.

```
class Etudiant:
    def __init__(self, cne, nom_complet, date_naissance):
        self.cne = cne
        self.nom_complet = nom_complet
        self.date_naissance = date_naissance

    def afficher(self):
        print(f"CNE : {self.cne}, Nom : {self.nom_complet},"
              f"Date de naissance : {self.date_naissance}")

class Etudiant2(Etudiant):
    def __init__(self, cne, nom_complet, date_naissance,
                 lieu_naissance):
        super().__init__(cne, nom_complet, date_naissance)
        self.lieu_naissance = lieu_naissance

    def afficher(self):
        print(f"CNE : {self.cne}, Nom : {self.nom_complet}, "
              f"Date de naissance : {self.date_naissance}, "
              f"Lieu : {self.lieu_naissance}")

etudiants = [
    Etudiant("123", "Moha Sage", "01/01/79"),
    Etudiant2("456", "Moha Fou", "12/05/88", "Maroc")
]

for etu in etudiants:
    etu.afficher()
```

Interface

- L'**interface** est un concept fondamental de la programmation orientée objet (POO).
- Elle définit un ensemble de **méthodes publiques** (et parfois de constantes) qu'une **classe doit implémenter**.
- Toute classe qui implémente cette interface doit **fournir une définition** pour chacune de **ces méthodes**.
- C'est un moyen d'abstraction : on se concentre sur ce qu'une classe **doit faire** (le comportement), plutôt que sur **la manière dont elle le fait** (l'implémentation).




```python
class EnregistreurEtudiant:
    #Enregistrer un étudiant, quelle que soit la BD
    def enregistrer(self, etudiant):
        raise NotImplementedError(
        "La méthode enregistrer()doit être implémentée!")

class EnregistreurMySQL(EnregistreurEtudiant):
    def enregistrer(self, etudiant):
        data = etudiant.to_dict()
        print(f"Enregistrement dans MySQL...")
        print(f"INSERT INTO etudiant VALUES ("
                f"{', '.join([repr(v) for v in data.values()])});")
```

Ségrégation des interfaces

- Un objet ne doit **pas dépendre** de méthodes qu'il **n'utilise pas**.
- Il est préférable de diviser une **interface générale** (monolithique) en **plusieurs interfaces spécifiques** et ciblées.
- Chaque objet n'implémente et n'accède qu'aux méthodes qui le concernent, évitant les dépendances inutiles.
- La classe **EnregistreurMySQL** doit implémenter la méthode **afficher()** dont il n'aura jamais besoin !

```
class EnregistreurEtudiant(ABC):
    @abstractmethod
    def enregistrer(self, etudiant):
        pass

    @abstractmethod
    def afficher(self, etudiant):
        pass

class EnregistreurMySQL(EnregistreurEtudiant):
    def enregistrer(self, etudiant):
        data = etudiant.to_dict()
        print(f"Enregistrement dans MySQL...")
        print(f"INSERT INTO etudiant VALUES ("
              f"{', '.join([repr(v) for v in data.values()])});")

    def afficher(self, etudiant):
        print(f"{etudiant.nom_complet} ({etudiant.cne}) "
              f"- Né le {etudiant.date_naissance}")

def enregistrer_etud(etudiant: Etudiant,
                     enregistreur: EnregistreurEtudiant):
    enregistreur.enregistrer(etudiant)

etud = Etudiant("123", "Franz Kafka", "01/01/1900")
enregistrer_etud(etud, EnregistreurMySQL())
```

Inversion des dépendances

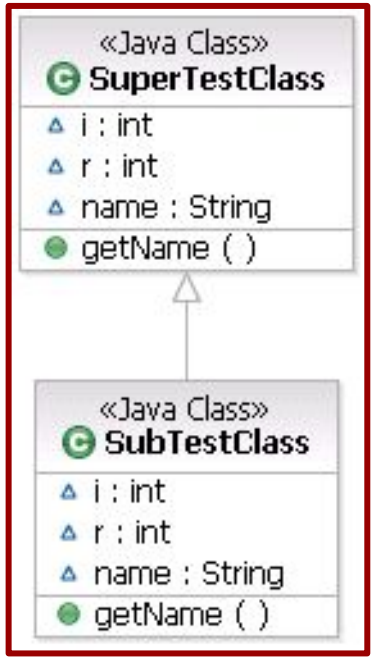
- Les **modules de haut niveau** ne doivent pas dépendre des **modules de bas niveau**. Les deux doivent dépendre d'**abstractions**.
- Les **abstractions** ne doivent pas dépendre des **détails**, mais l'inverse.

- La fonction d'enregistrement d'un étudiant (haut niveau) dépend directement de la logique spécifique à MySQL (bas niveau) → changer de BD obligerait à modifier cette fonction.
- **Solution** : inverser la dépendance pour que le code bas niveau (MySQL, MongoDB, …) dépende d'une abstraction définie au niveau supérieur (enregistrement d'un étudiant).

```
class EnregistreurEtudiant:
    #Enregistrer un étudiant, quelle que soit la BD
    def enregistrer(self, etudiant):
        raise NotImplementedError(
        "La méthode enregistrer()doit être implémentée!")

class EnregistreurMySQL(EnregistreurEtudiant):
    def enregistrer(self, etudiant):
        data = etudiant.to_dict()
        print(f"Enregistrement dans MySQL...")
        print(f"INSERT INTO etudiant VALUES ("
              f"{', '.join([repr(v) for v in data.values()])});")
```

```
class EnregistreurMongoDB(EnregistreurEtudiant):
    def enregistrer(self, etudiant):
        data = etudiant.to_dict()
        print(f"Enregistrement dans MongoDB...")
        print(f"db.etudiants.insert_one({data})")

def enregistrer_etud(etudiant: Etudiant,
                     enregistreur: EnregistreurEtudiant):
    enregistreur.enregistrer(etudiant)

etud = Etudiant("123", "Franz Kafka", "01/01/1900")
enregistrer_etud(etud, EnregistreurMySQL())
enregistrer_etud(etud, EnregistreurMongoDB())
```

UML/JAVA



SuperTestClass.java:

```
public class SuperTestClass {
	int i = 3;
	int r = 5;
	String name ="myName";

	public void getName(){
	};

}
```

SubTestClass.java:

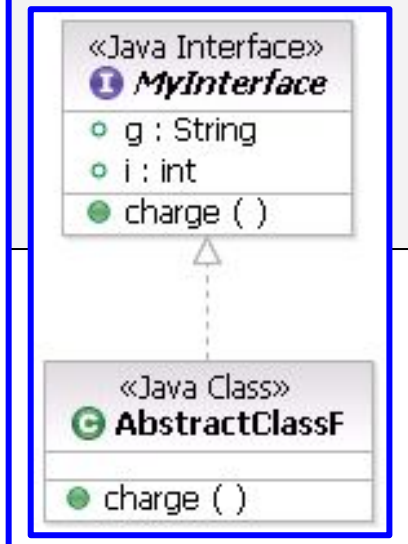
```
public class SubTestClasss
	 extends SuperTestClass {
	int i = 2;
	int r = 3;
	String name ="myName";

	public void getName(){
	};

}
```

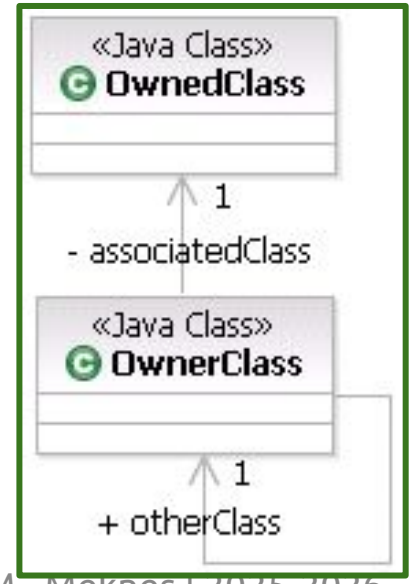
MyInterface.java:

```
public interface MyInterface {
	String g= "";
	int i= 0;

	public void charge (int x);

}
```

AbstractClassF.java:

```
public class AbstractClassF
		implements MyInterface {

	public void charge(int x){
	};

}
```





OwnedClass.Java:

```
public class OwnedClass {

	// <<class body>>
}
```

OwnerClass.Java:

```
public class OwnerClass {
private OwnedClass associatedClass;
public OwnerClass otherClass;

	// <<class body>>
}
```

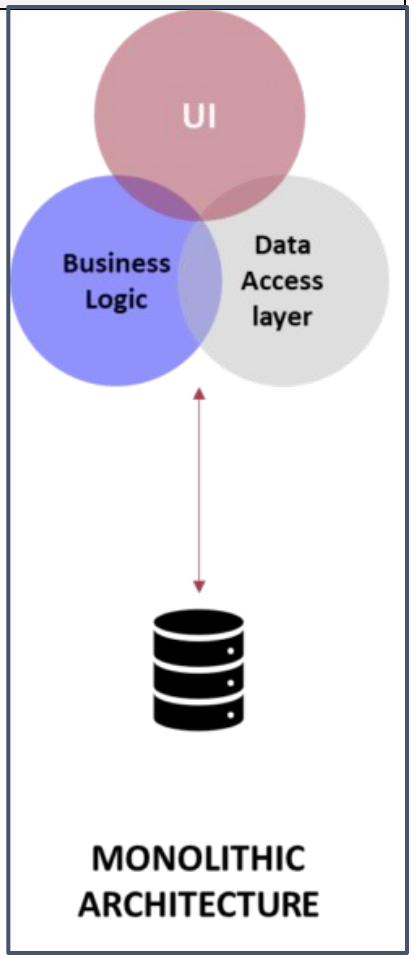
Application

Réaliser un petit projet Java qui respecte les principes **SOLID**

Nous souhaitons modéliser une entité **Etudiant** et son sous-type **EtudiantUMI**, qui introduit un attribut supplémentaire. Le programme doit permettre d'afficher et de sauvegarder (*simulé simplement par un affichage à l'écran, pour le moment!*) les informations des étudiants vers des différentes BD (MySQL, MongoDB).

Architecture monolithique

- L'**architecture monolithique** est le modèle traditionnel où toutes les fonctionnalités d'une application sont regroupées dans une seule unité:
  - Un seul exécutable, un seul répertoire de code source, une seule BD.
  - Les composantes sont étroitement couplées.
  - Application autonome et indépendante.
- Facile à prendre en main, rapide à développer (au début), simple à déployer.
- Un changement de code → reconstruire et redéployer toute l'application.
- Complexité de mises à jour et d'ajout de nouvelles fonctionnalités, en particulier avec des applications volumineuses.
- Difficile de faire évoluer une seule fonctionnalité indépendamment.
- Les architectures moderne → décomposition en services / fonctionnalités spécialisés et faiblement couplés → agilité, flexibilité, et évolutivité.



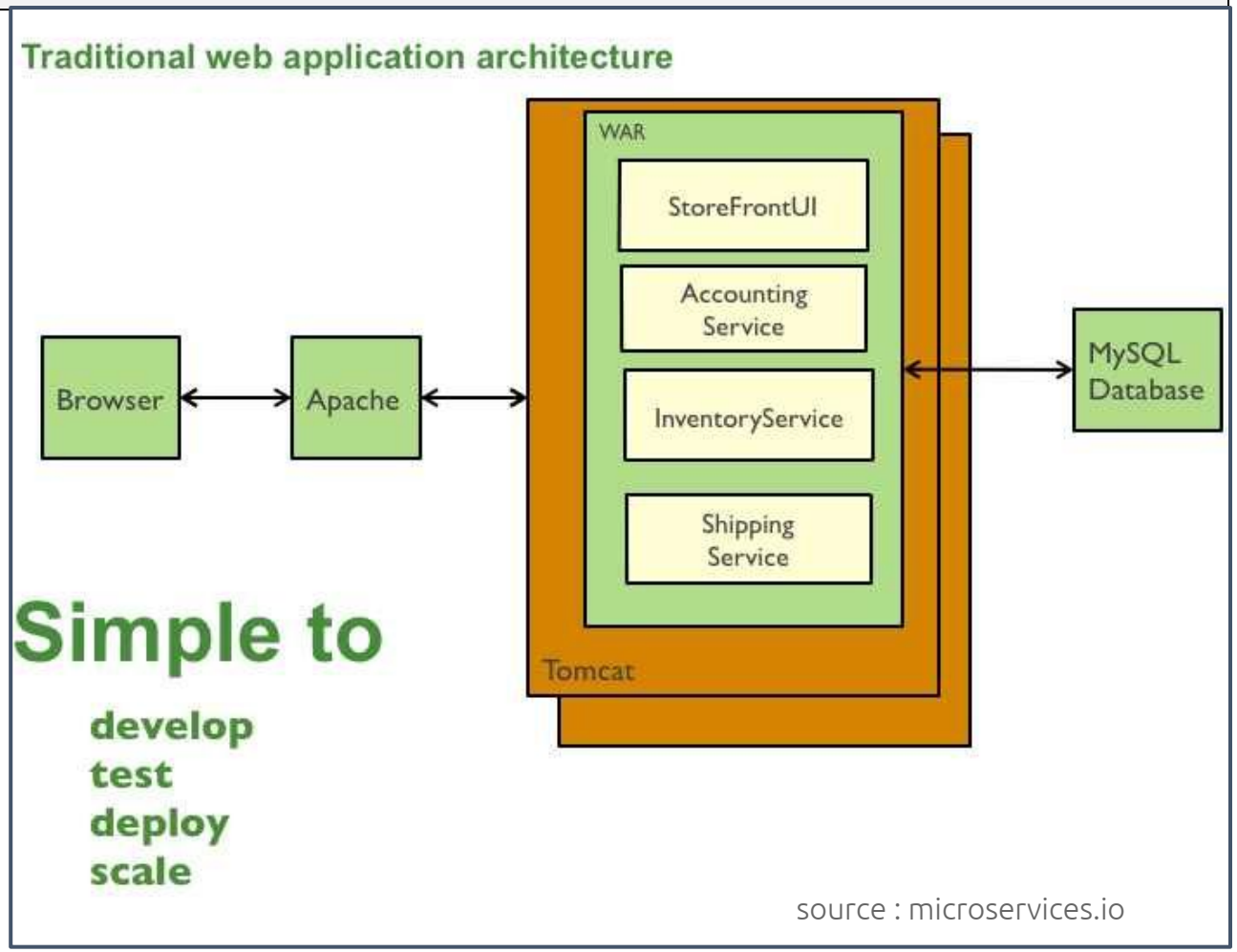
Architecture monolithique

Une application e-commerce est déployée comme une seule application monolithique. Les trois fonctionnalités métiers sont :

- Prise de commandes.
- Vérification de l'inventaire (stock) et du crédit disponible.
- Expédition des commandes clients.

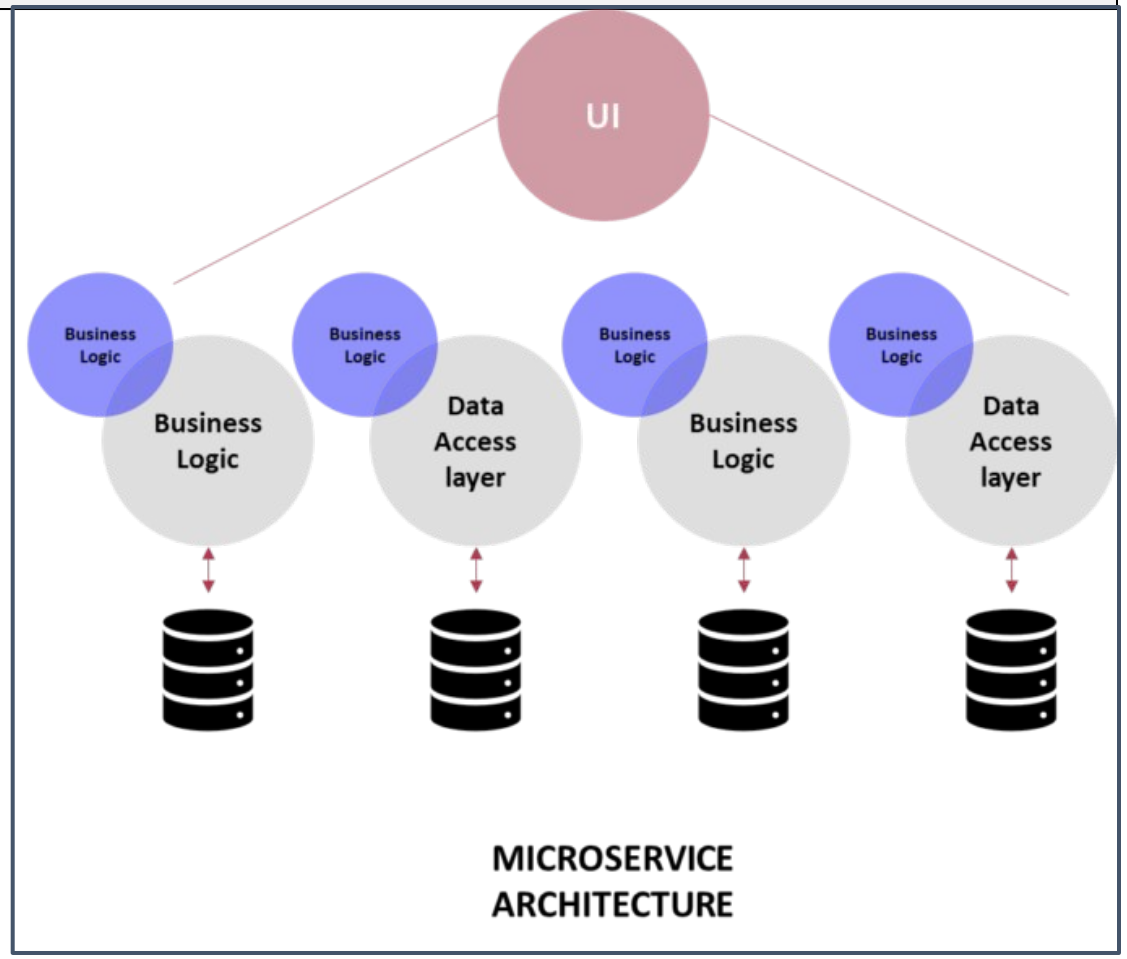
Toutes les composantes, y compris l'**interface utilisateur** (StoreFrontUI) et les services **backend** (gestion du crédit, inventaire, expédition) sont regroupées dans un même projet.

→ *Par exemple, une **application Java** peut être déployée dans **un seul fichier WAR** sur un serveur **Tomcat**.*



Architecture en micro services

- Une **architecture microservices** divise une application en fonctionnalités encapsulées au sein de petits services autonomes et faiblement couplés.
- Chaque services est géré (test, déploiement, …) et évolue (MAJ, extension, ..) indépendamment des autres.
  - Un microservice a son propre **objectif** : gestion des utilisateurs, paiement, ….
  - Un microservice a sa propre **logique métier** et sa propre BD.
  - Un microservice peut être implémenté dans un langage de programmation différent des autres.
- Les microservices communiquent entre eux à l'aide d'interfaces de programmation d'application (API) indépendantes de tout langage/technologie.
- Un microservice conçu pour une tâche peut être réutilisé par d'autres applications.



Architecture en micro services

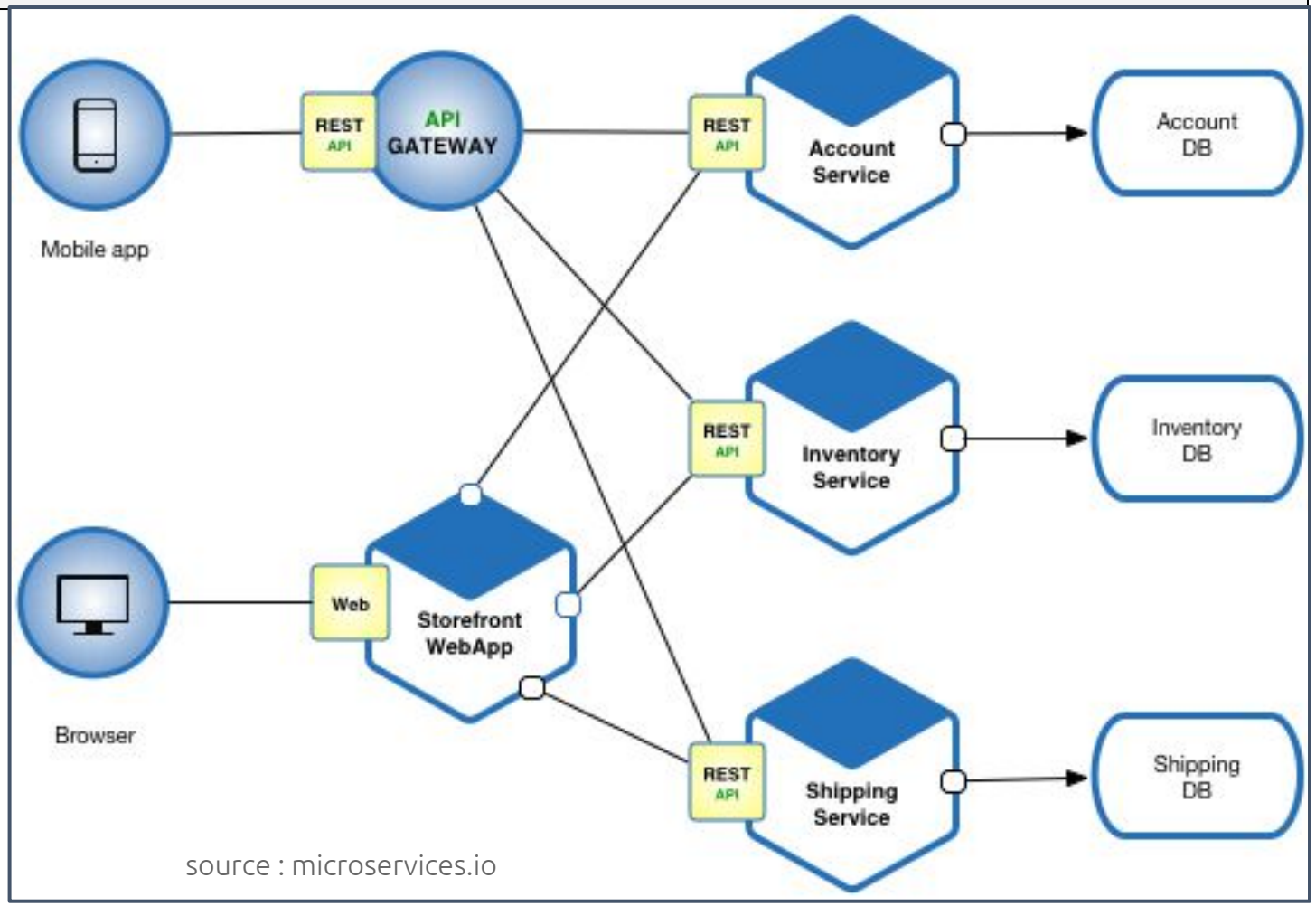
La même application e-commerce peut être conçue comme un ensemble de services indépendants.

Chaque service gère une fonctionnalité spécifique :

- Interface utilisateur.
- Vérification du crédit.
- Gestion de l'inventaire .
- Expédition des commandes.

Les services communiquent entre eux via des API :

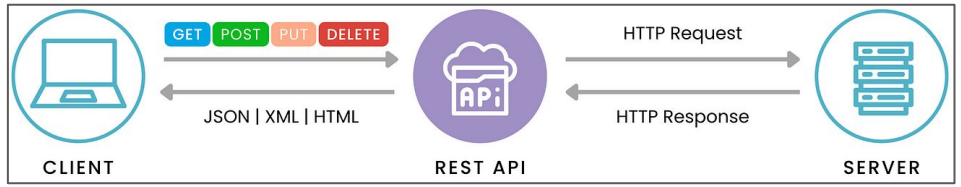
- Mettre à jour, déployer, et faire évoluer chaque service indépendamment.
- Agilité et flexibilité par rapport à la version monolithique.

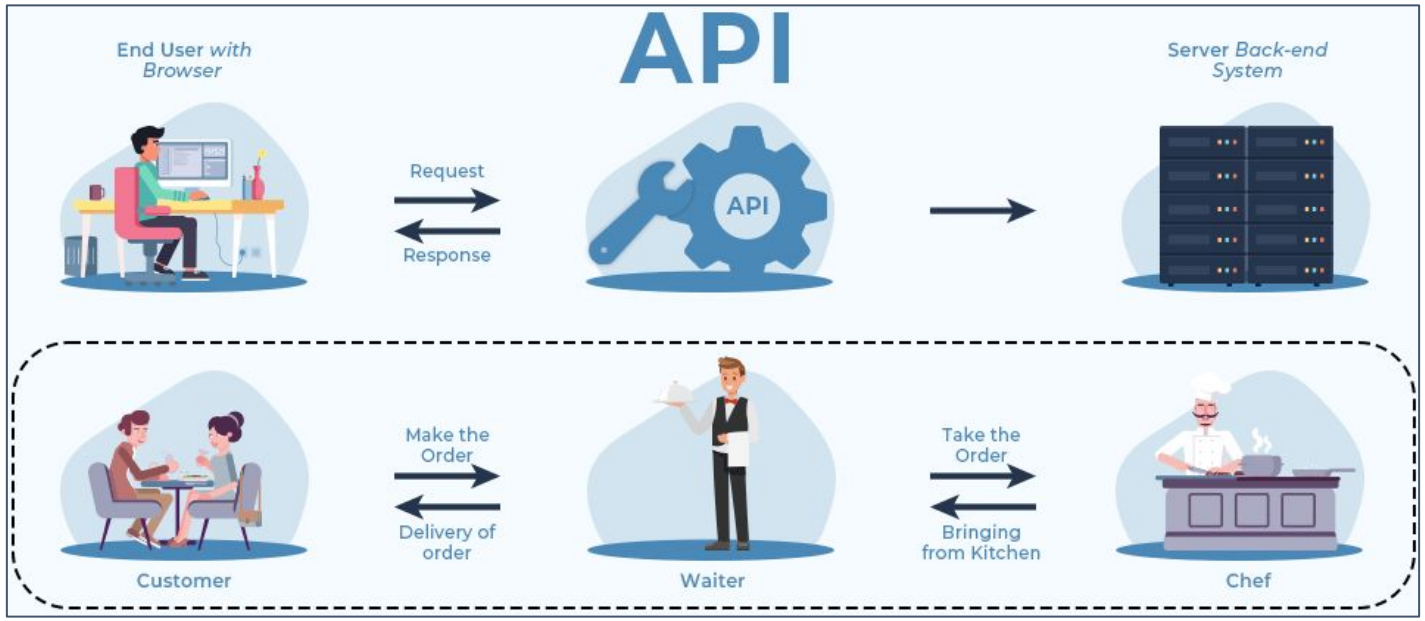


source : microservices.io

API

- Une API (Application Programming Interface) est un ensemble de règles et de protocoles qui permet à deux applications de communiquer et d'échanger des données (intermédiaire, moyen de communication).





- REST (Representational State Transfer) est une API qui définit la communication entre deux applications via le protocole HTTP, notamment à travers les requêtes GET et POST.

API REST

```
 * Serving Flask app 'jobintech'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production dep
loyment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
127.0.0.1 - - [11/Nov/2025 13:34:21] "GET /addjs?a=2&b=3 HTTP/1.1" 200 -
127.0.0.1 - - [11/Nov/2025 13:36:08] "GET /addjs?a=2&b=3 HTTP/1.1" 200 -
127.0.0.1 - - [11/Nov/2025 13:36:14] "GET /add?a=2&b=3 HTTP/1.1" 200 -
```

http://localhost:5000/addjs?a=2&b=3


```python
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/')
def accueil():
    return "API JOBINTECH OK !"

@app.route('/add')
def addition():
    a = float(request.args.get('a', 0))
    b = float(request.args.get('b', 0))
    return str(a + b + 0.5)

@app.route('/addjs')
def additionjs():
    a = float(request.args.get('a', 0))
    b = float(request.args.get('b', 0))
    return jsonify({"a": a, "b": b, "resultat": a + b + 0.5})

if __name__ == '__main__':
    app.run()
```

API REST

```
 * Serving Flask app 'jobintech'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production dep
loyment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
127.0.0.1 - - [11/Nov/2025 13:34:21] "GET /addjs?a=2&b=3 HTTP/1.1" 200 -
127.0.0.1 - - [11/Nov/2025 13:36:08] "GET /addjs?a=2&b=3 HTTP/1.1" 200 -
127.0.0.1 - - [11/Nov/2025 13:36:14] "GET /add?a=2&b=3 HTTP/1.1" 200 -
```

```
Saisir les deux variables à additionner :
21
3
Résultat : 24.5
```

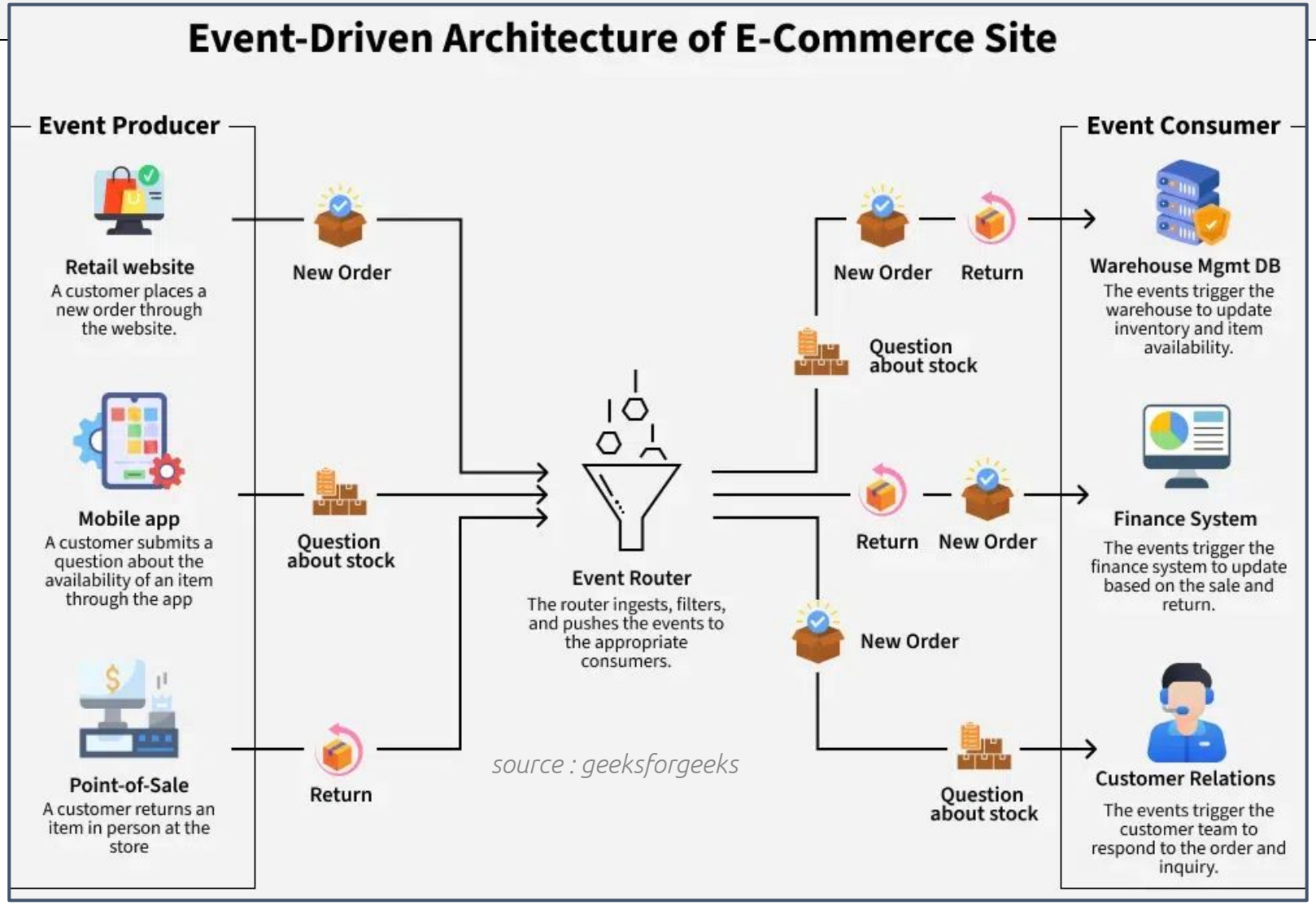
```java
import java.io.*;
import java.net.*;
import java.util.Scanner;

public class TestFlaskAPI {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Saisir les deux variables à additionner :");
        float x =  sc.nextFloat();
        float y =  sc.nextFloat();
        sc.close();

        try {
            URL url = new URL("http://127.0.0.1:5000/add?a=" + x + "&b=" + y);
            HttpURLConnection con = (HttpURLConnection) url.openConnection();
            con.setRequestMethod("GET");

            BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream()));
            String line = in.readLine();  // lire la première ligne seulement
            in.close();

            System.out.println("Résultat : " + line);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Architectures orientées événements

- Une architecture orientée événements (*EDA : Event-Driven Architecture*) est un modèle d'architecture logicielle basé sur la production et la consommation d'événements.
  - Événement : action ou changement d'état d'une composante du système.
    Exemple : {type: "EtudiantInscrit", EtudCIN: "JK6764"}
  - Producteur : composante du système qui **émet l'événement**.
    Exemple : le modèle ou le service d'insertion dans la BD.
  - Consommateur : composante qui **souscrit à un événement pour effectuer une action**.
    Exemple : envoyer un SMS de bienvenue au nouvel étudiant inscrit.
  - Canal de communication : moyen ou technologie qui permet d'échanger les messages et de gérer les souscriptions.

- L'EDA est **faiblement couplée**, et la communication peut être **synchrone** ou **asynchrone**.
  - Adaptée aux applications modernes, distribuées, et en temps réel.
  - Facilement scalable : rajouter de nouveaux producteurs ou consommateurs.
- Une architecture microservices peut être orientée événements.

Architectures orientées événements

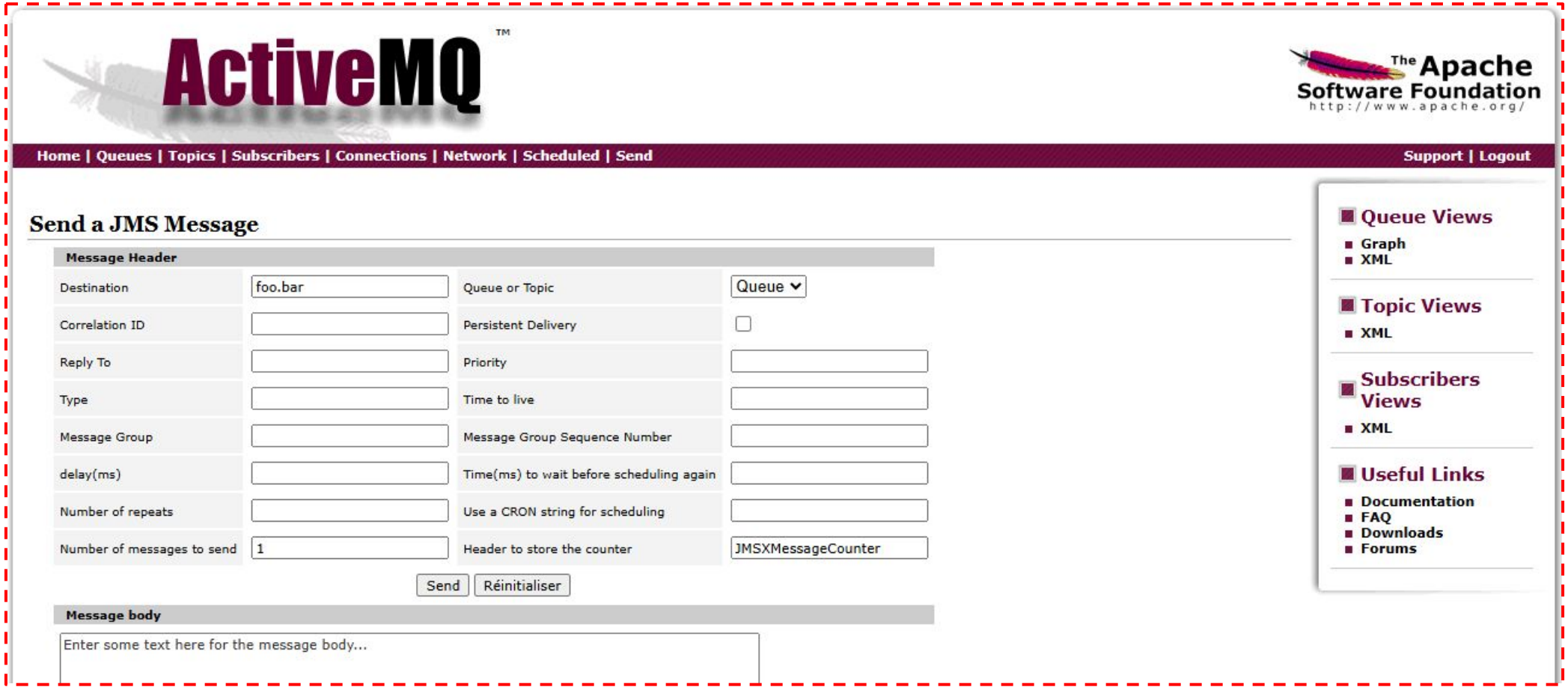


ActiveMQ

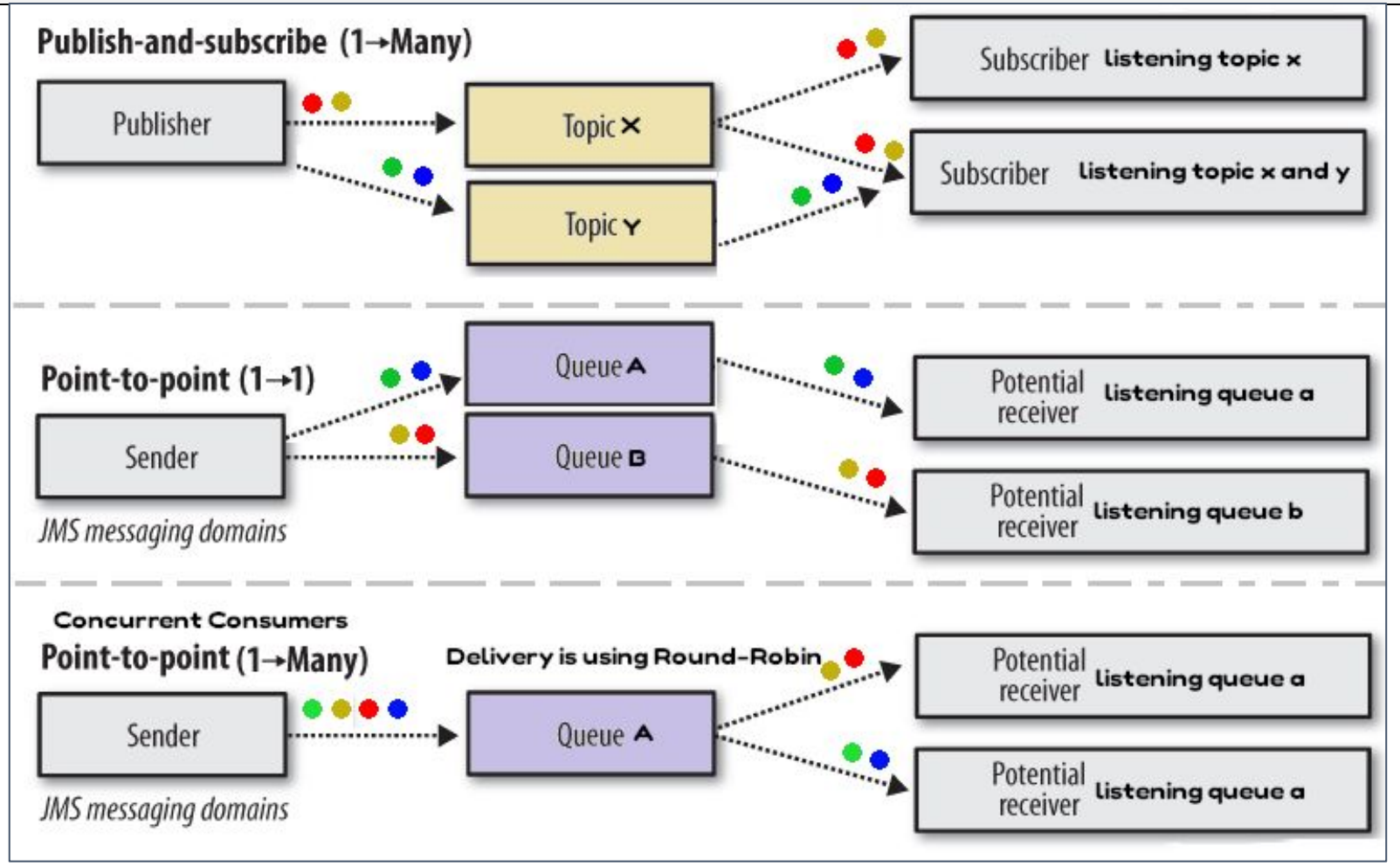
- Apache ActiveMQ est un broker de messages open-source implémenté en **Java**.
  - Reçoit des messages des producteurs et les achemine aux consommateurs.
- Permet la **communication asynchrone** entre différentes applications ou services.
- Supporte de nombreux protocoles standards (OpenWire, MQTT, STOMP, REST, ...), permettant l'interaction avec des applications hétérogènes (C++, Python, ….).
- Permet de **découpler les composantes** (microservices) dans une architecture logicielle :
  - Les producteurs et les consommateurs échangent sans avoir besoin de se connaître.
  - Si une composante tombe en panne, les autres peuvent continuer à fonctionner.
  - Les messages et les données échangées sont conservés (persistance) et ne sont pas perdus.
- **Queue** (**Point-à-Point**) : un message envoyé à la file n'est reçu que par un seul consommateur. Si plusieurs consommateurs sont disponibles, ActiveMQ distribue les messages selon une stratégie de répartition de charge (Round Robin). *Persistance par défaut*.
- **Topic** (**Publisher/Subscriber**) : tous les consommateurs qui se sont abonnés au topic reçoivent une copie du message qui y est publié. *Non persistant par défaut (abonnement **durable** possible)*.

ActiveMQ

> activemq start



ActiveMQ


Publish-and-subscribe (1→Many)
Publisher
Topic X
Topic Y
Subscriber listening topic x
Subscriber listening topic x and y
Point-to-point (1→1)
Sender
Queue A
Queue B
Potential receiver listening queue a
Potential receiver listening queue b
JMS messaging domains
Concurrent Consumers
Point-to-point (1→Many)
Delivery is using Round-Robin
Sender
Queue A
Potential receiver listening queue a
Potential receiver listening queue a
JMS messaging domains

ActiveMQ

Producteur (Python)

```python
import stomp

conn = stomp.Connection([('localhost', 61613)])
conn.connect(login='admin', passcode='admin', wait=True)

#conn.send(body='Message depuis Python !', destination='/queue/TEST.QUEUE')
conn.send(body='Message depuis Python !', destination='/topic/TEST.TOPIC')
print("Message envoyé à ActiveMQ !")

conn.disconnect()
```

Les messages envoyés dans un topic/queue peuvent être lus via **REST** :
http://localhost:8161/api/message/TEST.TOPIC?type=**topic**

ActiveMQ

Consommateur (Python)

```python
import stomp
import time

class MyListener(stomp.ConnectionListener):
    def on_message(self, message):
        print('Message reçu :', message.body)

conn = stomp.Connection([('localhost', 61613)])
conn.set_listener('', MyListener())
conn.connect('admin', 'admin', wait=True)
#conn.subscribe(destination='/queue/TEST.QUEUE', id=1, ack='auto')
conn.subscribe(destination='/topic/TEST.TOPIC', id=1, ack='auto')

print("En attente de nouveaux messages ...")

try:
    while True:
        time.sleep(1)
except KeyboardInterrupt:
    conn.disconnect()
```

ActiveMQ

Producteur (JAVA)

```java
public class Producteur {

    public static void main(String[] args) {
        Connection connection = null;
        Session session = null;
        MessageProducer producer = null;

        try {
            ActiveMQConnectionFactory connectionFactory =
                        new ActiveMQConnectionFactory("tcp://localhost:61616");
            connection = connectionFactory.createConnection();
            connection.start();

            session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            //Destination destination = session.createQueue("TEST.QUEUE");
            Destination destination = session.createTopic("TEST.TOPIC");
            producer = session.createProducer(destination);

            TextMessage message = session.createTextMessage("Message depuis JAVA !");
            producer.send(message);
            System.out.println("Message envoyé à ActiveMQ !");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (producer != null) producer.close();
                if (session != null) session.close();
                if (connection != null) connection.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

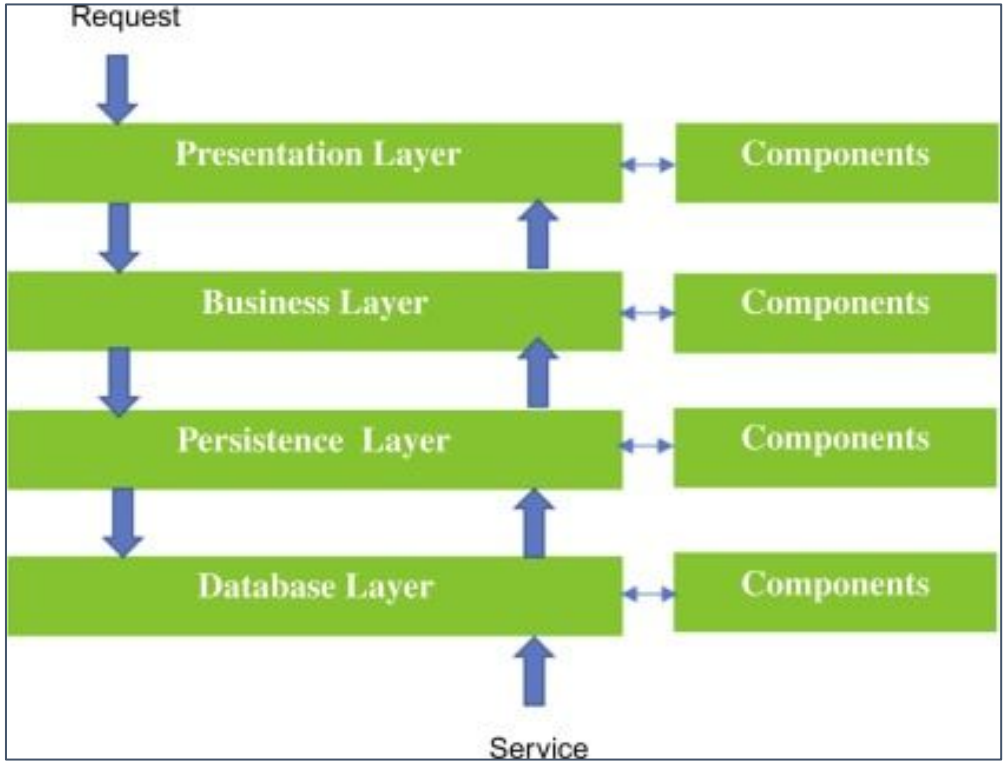
ActiveMQ

Consommateur (JAVA)

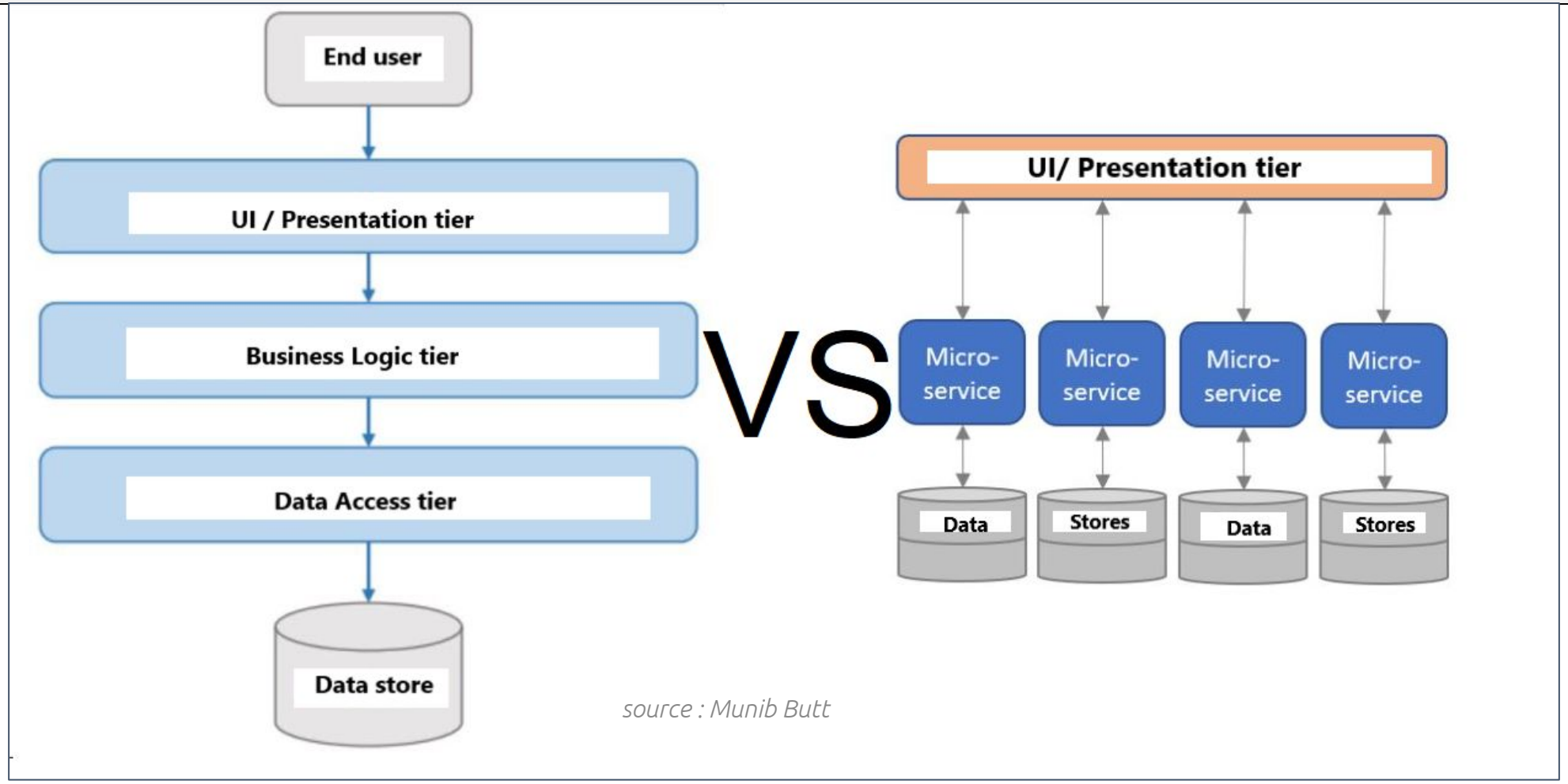
```java
import javax.jms.Connection;

public class Consommateur {

    public static void main(String[] args) {
        Connection connection = null;
        Session session = null;
        MessageConsumer consumer = null;

        try {
            ActiveMQConnectionFactory factory =
                    new ActiveMQConnectionFactory("tcp://localhost:61616");
            connection = factory.createConnection();
            connection.start();

            session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            //Destination destination = session.createQueue("TEST.QUEUE");
            Destination destination = session.createTopic("TEST.TOPIC");
            consumer = session.createConsumer(destination);

            System.out.println("En attente de nouveaux messages ...");
            while (true) {
                Message msg = consumer.receive();
                if (msg instanceof TextMessage) {
                    System.out.println("Message reçu : " + ((TextMessage) msg).getText());
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Architecture en couches

- L'architecture en couches (*Layered Architecture*) organise les composants horizontalement, en couches à responsabilité unique.
- Chaque couche utilise les services de la couche inférieure pour offrir des services à la couche supérieure (flux généralement unidirectionnel).
- La plupart des architectures logicielles sont structurées en 4 couches principales :
  - Présentation (UI) : gère l'interface utilisateur et les interactions (saisie, affichage, …).
  - Métier (Business) : contient la logique métier de l'application : règles, traitements, calculs, …
  - Persistence (Data Access) : assure l'accès aux données et l'interaction avec le système du stockage.
  - BD (Database) : représente le SGBD qui stocke physiquement les données.



Couches vs microservices

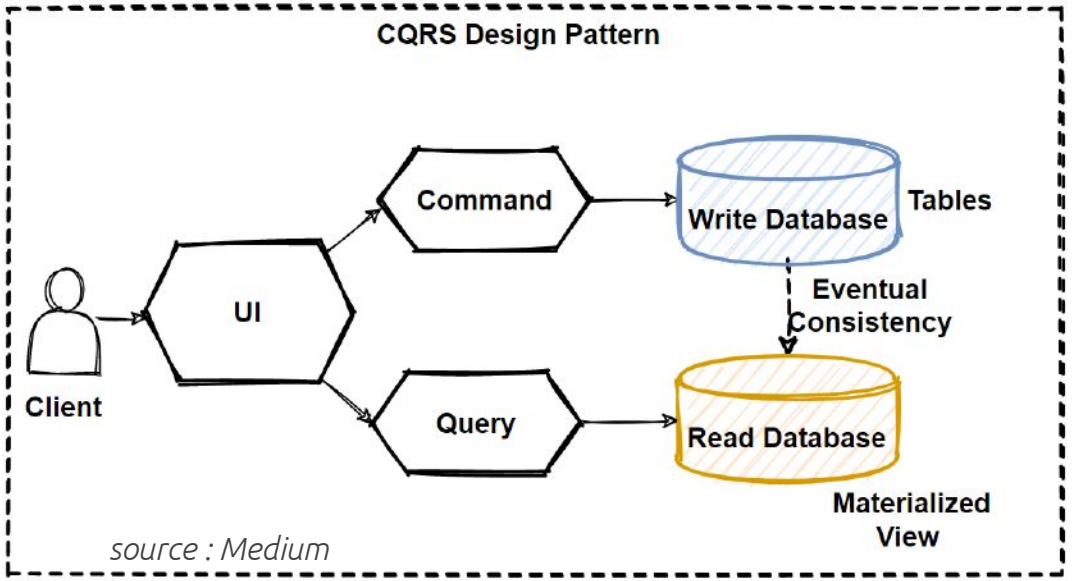


source : Munib Butt

CQRS

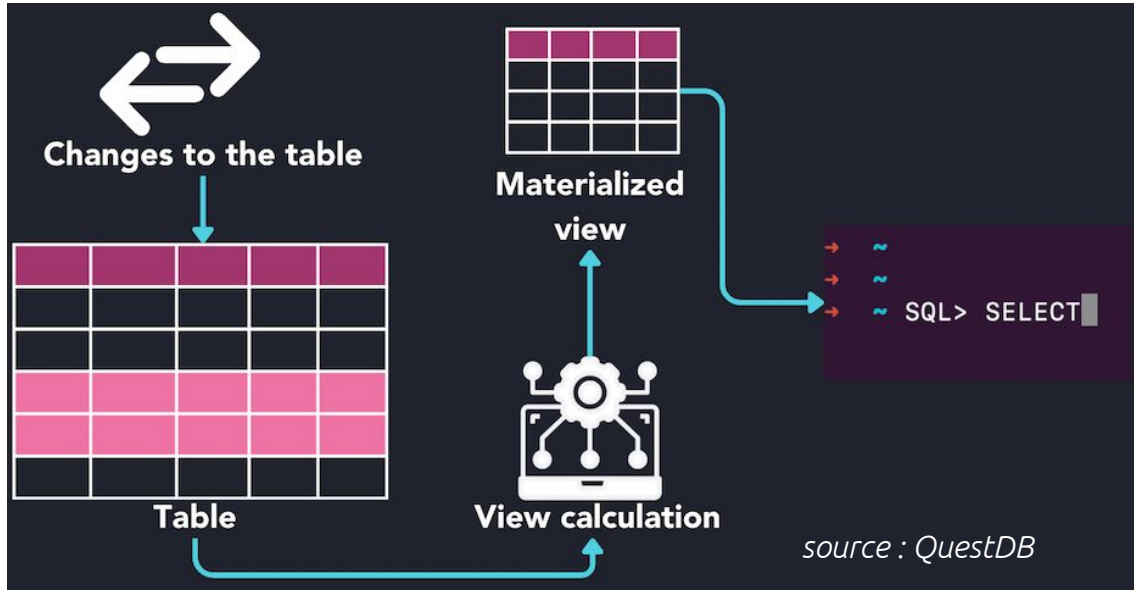
- L'**architecture CQRS** (**C**ommand **Q**uery **R**esponsibility **S**egregation) sépare les opérations qui modifient l'état du système (Command) des opérations qui lisent les données sans les modifier (Query).
  - On sépare entre les opérations de lecture (SELECT) de celles d'insertion, MAJ, et suppression.
  - Deux composantes (objet, service) au lieu d'une qui gère les deux types d'opérations à la fois :
    - **Command Handler** : exécute les requêtes de modification (executeUpdate dans JDBC).
    - **Query Handler** : exécute les requêtes de lecture (executeQuery dans JDBC).

- Séparation des responsabilités :
  - Meilleure gestion du business logic (séparation).
- Optimisation et scalabilité indépendante :
  - Optimiser chaque composant selon la charge.
  - La lecture est souvent plus fréquente.
- Flexibilité technologique :
  - Utiliser des technologies différentes et optimisées (traitements) pour chaque composant.
  - Utilisation de BD différentes (SQL, NoSQL).
- Utilisation des **Vues** dénormalisées :
  - Dans la même BD ou dans une BD différente.
  - Adaptation aux besoins de l'utilisateur (SELECT).

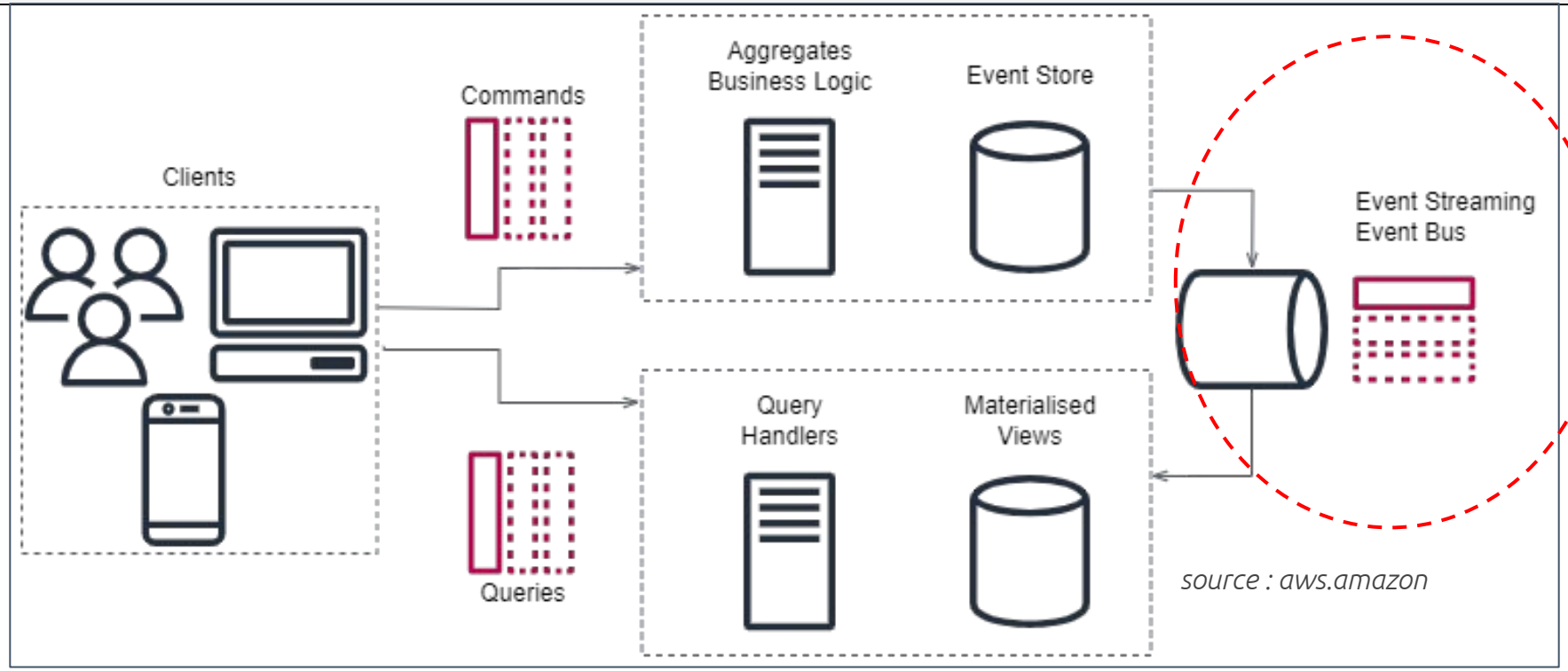


Vue

- Une **vue** est une définition stockée d'une requête SQL de type SELECT. Il s'agit d'une **table** virtuelle contenant des champs et des colonnes. Elle permet d'éviter de retaper une même requête longue et complexe (jointures multiples) à chaque fois (une vue = **alias** permanent d'une requête).
  - CREATE [ou REPLACE] VIEW inscriptions AS
    SELECT e.nom AS Nom_Etudiant, f.nom AS Nom_Formation FROM Etudiants e, Formations f
    WHERE e.id_formation = f.d;
  - → SELECT * from inscriptions;
- Une **vue matérialisée** stocke physiquement le résultat d'une requête → très rapide mais consomme de l'espace de stockage et peut devenir obsolète.
- Mise à jour des vues (dans le contexte CQRS) :
  - Le modèle d'écriture produit des événements : *etudiant_created*, *formation_validated*.
  - Le modèle de lecture met à jour les vues à partir de ces événements → *eventually consistent* !



CQRS



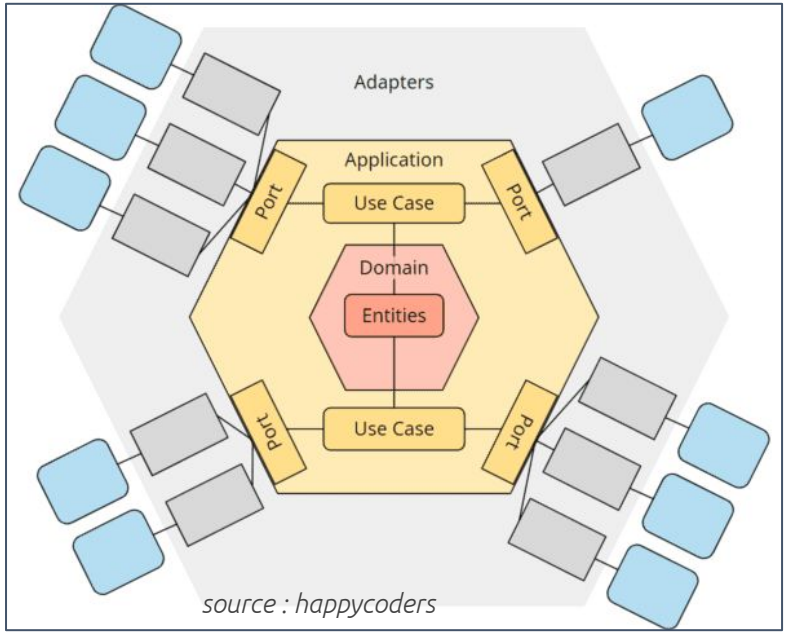
Le **broker** reçoit les messages de modification (create, update, delete) relatifs aux tables, les persistent pendant une durée limitée, et les achemine aux consommateurs (subscribers) intéressés : exactement une fois (éviter la duplication des traitements), et dans l'ordre (crucial pour la consistance). Grâce à la persistance, les applications disposent de temps pour propager les modifications (tolérance aux pannes, gestion de la charge, cohérence éventuelle).

Architecture Hexagonale (Ports & Adapters)

- L'architecture hexagonale (*Ports and Adapters*) permet de créer des composants faiblement couplés. Elle repose sur la séparation du *core* de l'application des systèmes externes (BD, UI, services tiers) :
  - Noyau (Core) : contient la logique métier pure et ne dépend pas de l'extérieur.
  - Ports : les **interfaces** (contrats) définies et exposées par le noyau pour communiquer en entrée (recevoir des commandes pour lancer un processus métier : créer un étudiant) et en sortie (émettre des commandes : requête vers une BD, appeler un service tiers).
  - Adaptateurs : les **implémentations** (situées à l'extérieur du noyau) concrètes des ports pour agir comme des intermédiaires entre le noyau et l'extérieur (adapter le format des données, traduire des objets en requêtes SQL, …).
- Isolation complète de la logique métier.
- Scalabilité et maintenabilité faciles : on ne change pas la logique métier, mais uniquement les adaptateurs, pour migrer par exemple de MySQL vers MongoDB.
  - **Enregistrement d'un nouvel étudiant** : le noyau exprime un besoin, le port de sortie définit l'opération (quoi), l'adaptateur fournit l'implémentation (comment).

Architecture Hexagonale (Ports & Adapters)

```java
interface PortEnregistrement {
    void enregistrer(Etudiant etud);
}

class AdaptateurEnregistrementMySQL implements PortEnregistrement {

    @Override
    public void enregistrer(Etudiant et) {
        // connexion JDBC, construction de la requête SQL
        // executeUpdate("INSERT INTO Etudiants...")
    }
}

class AdaptateurEnregistrementMongoDB implements PortEnregistrement {

    @Override
    public void enregistrer(Etudiant et) {
        // connexion MongoDB, construction de la requête
        // db.insertOne({})
    }
}
```

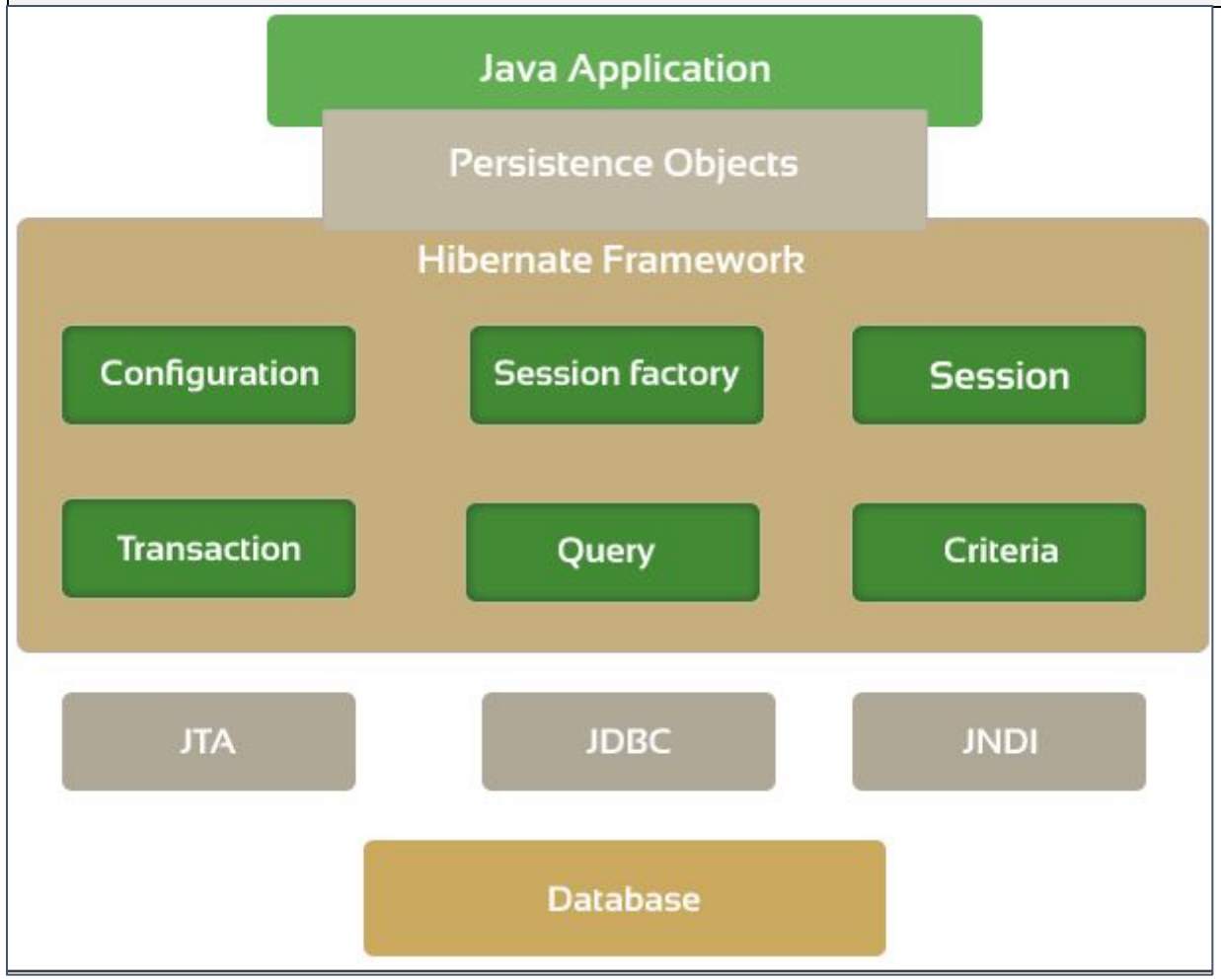


source : happycoders

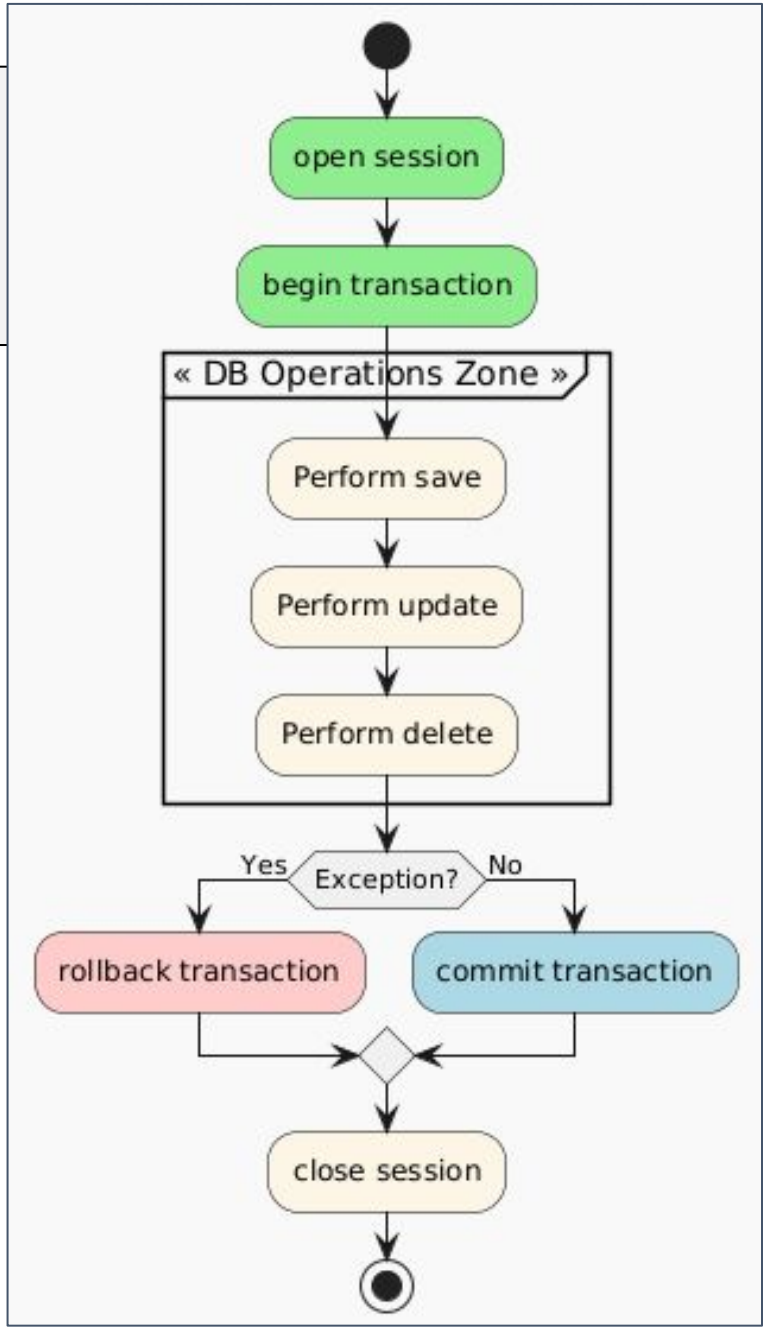
JPA Hibernate

- La **mapping Objet-Relationnel** (ORM : *Object-Relational Mapping*) est le processus de faire correspondre des objets Java et des tables de BD.
  - Interagir avec la BD sans écrire des requêtes SQL (générées par l'ORM).
- L'**API de Persistance Java** (JPA : *Java Persistence API*) est la spécification qui définit comment persister des données dans Java.
- **Hibernate** est le framework ORM Java les plus populaire qui implémente la spécification JPA.
  - Encapsulation des requêtes SQL.
  - Gestion des transactions et des relations entre les entités.
- Un fichier de configuration XML (hibernate.cfg.xml) est une des méthodes pour configurer **Hibernate** : il contient les détails de connexion, les classes à mapper, et d'autres paramètres.
- Une classe Java est mappée à une table via l'annotation @Entity :
  - @Table (name = "nom_table") : spécifie le nom de la table si différent du nom de la classe.
  - @Id : marque la clé primaire de l'entité.
  - @GeneratedValue(strategy = GenerationType.IDENTITY) : indique que le ID est généré automatiquement par la BD (AUTO_INCREMENT !).

JPA Hibernate



source : medium



JPA Hibernate

```xml
<?xml version='1.0' encoding='utf-8'?>
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hiberdb</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">1002</property>

        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

        <property name="hibernate.hbm2ddl.auto">update</property>
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>

        <mapping class="ma.ac.umi.jobintech.Etudiant"/>

    </session-factory>
</hibernate-configuration>
```

```java
@Entity
@Table(name = "Etudiant")
public class Etudiant {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String nom;
    private String prenom;
    private float age;

    // constructeur par défaut (obligatoire pour Hibernate)
    public Etudiant() {}
```

JPA Hibernate

```
SessionFactory factory = new Configuration()
        .configure("hibernate.cfg.xml")
        //.addAnnotatedClass(Etudiant.class) //<mapping class=""/>
        .buildSessionFactory();

Session session = null;
Transaction transaction = null;

try {
    session = factory.openSession();
    transaction = session.beginTransaction();

    Etudiant etu1 = new Etudiant("name","prename", 44);
    session.save(etu1);

    Etudiant etu2 = session.get(Etudiant.class, etu1.getId());
    etu2.setNom("Updated");
    session.update(etu2);

    Etudiant etu3 = session.get(Etudiant.class,7);
    session.delete(etu3);

    transaction.commit();
```

JPA Hibernate

- Quand le nom d'un champ dans la base de données est différent de l'attribut de la classe Java, on utilise l'annotation : @Column(name = "nom_champ_table").
  - On peut fournir les informations utiles à la génération du **DDL** si Hibernate est utilisé pour créer automatiquement les tables : @Column(name="nom", length=20, nullable=false).
- On peut interagir avec la BD en utilisant **HQL** (*Hibernate Query Language*) : un langage orienté objet qui manipule les objets Java, non pas les tables de la BD (Hibernate traduit ensuite en SQL).
  - **Récupérer toutes les lignes** :

```
List<Etudiant> list = session.createQuery("FROM Etudiant", Etudiant.class).list();
```

  - **Filtrer** :

```
Etudiant etu = session.createQuery("FROM Etudiant e WHERE e.id = :id", Etudiant.class)
                      .setParameter("id", 21).uniqueResult();
```

  - **Projeter** :

```
List<String> noms = session.createQuery("SELECT e.nom FROM Etudiant e", String.class).list();
```

  - **Trier** :

```
List<Etudiant> list = session.createQuery("FROM Etudiant e ORDER BY e.age ASC", Etudiant.class).list();
```

JPA Hibernate

- Les annotations de jointure servent à définir les relations entre les entités et leur mapping dans la BD.
- @OneToOne : une liaison 1-1 entre deux entités. La FK est dans la table de la classe annotée.

```
@OneToOne
@JoinColumn(name = "mon_entité_id")
private Entité mon_entité;
```

- @ManyToOne : une liaison N-1 (plusieurs entités liées à une seule autre). La FK est dans la table de la classe annotée (côté “N”).

```
@ManyToOne
@JoinColumn(name = "formation_id")
private Formation formation;
```

- On peut écrire une requête HQL avec jointure pour récupérer les objets liés :

```
List<Object[]> results = session.createQuery("FROM Etudiant e INNER JOIN e.formation f").list();
for (Object[] res : results) {
        Etudiant e = (Etudiant) res[0];
        Formation f = (Formation) res[1];
        System.out.println(e.getNom() + " :: " + f.getNom());
 }
```

Exercice

Projet Hibernate-MySQL-CQRS-ActiveMQ

Mettre en place un projet Java utilisant Hibernate/JPA, CQRS et messaging avec ActiveMQ pour gérer les inscriptions d'étudiants dans des formations.
Le projet doit séparer les responsabilités d'écriture (Command) et de lecture (Query) et garantir que la vue des inscriptions deviendra éventuellement à jour après chaque modification.

- Etudiant (id, nom, prenom, formation_id)
- Formation (id, nom)
- Inscriptions (id, etudiant_id, formation_id, nom_complet_etudiant, nom_formation) : table SQL pour simuler une vue matérialisée (*ici, dans la même BD*).

- Modèle d'écriture (Command) : gère la modification des tables Etudiant et Formation et notifie le service de lecture (produit/publie un événement) via **ActiveMQ** (*violation de SRP !*).
- Modèle de lecture (Query) : consomme les événements, met à jour la vue Inscriptions et fournit les requêtes de lecture (les lectures se font uniquement via la vue Inscriptions) (*violation de SRP !*).

- Les **factory** de connexions à la BD et à **ActiveMQ** sont centralisées : les objets SessionFactory (Hibernate) et ActiveMQConnectionFactory sont des **singletons**.