

ISADEVOPS

Cohésion : mesure à quel point les éléments d'un module ou d'une classe travaillent ensemble pour réaliser une seule tâche ou responsabilité. Plus la cohésion est forte, plus le module est spécialisé et focalisé.

Exemple : Une classe

`PDFExporter` qui ne s'occupe que de générer des fichiers PDF à partir de données – et pas d'envoyer des mails, ni de lire des fichiers – est fortement cohésive.

Couplage : mesure le degré de dépendance entre deux modules. Moins deux modules sont dépendants, plus le couplage est faible, ce qui est souhaitable.

Exemple : Si une classe `OrderService` dépend directement de `DatabaseConnection`, elle est fortement couplée à l'implémentation. Si elle dépend d'une interface `OrderRepository`, elle est faiblement couplée, car on peut facilement changer la base de données ou simuler des appels en test.

Principes SOLID :

- Single Responsibility Principle : chaque classe s'occupe d'une chose
- Open/Closed Principle : Les entités doivent être **ouvertes à l'extension** mais **fermées à la modification**.
- Liskov Substitution Principle : Les classes dérivées doivent pouvoir être utilisées comme leurs classes de base sans altérer le comportement. Exemple :

```
class Bird {  
    void fly() { /* voler */ }  
}  
  
class Ostrich extends Bird {  
    void fly() { throw new UnsupportedOperationException(); } // ✗ Viol LSP  
}
```

- Interface Segregation Principle : Il vaut mieux avoir plusieurs **interfaces spécifiques** qu'une seule interface générale.

- Dependency Inversion Principle : Les classes doivent dépendre **d'abstractions**, pas de classes concrètes.

POJO (Plain Old Java Object) : classe Java simple qui ne dépend d'aucune bibliothèque externe (comme Spring, Jakarta, etc.).

Bean : Un Bean Spring est un objet géré par le conteneur Spring.

Il peut être n'importe quelle classe Java (souvent un POJO), mais il est instancié, injecté, et géré par Spring. Un Bean est un **composant actif** du système, souvent utilisé pour **services, logiques métiers, accès aux données, etc.** Pour qu'une classe soit un Spring Bean, elle doit :

- être annotée avec une annotation comme `@Component`, `@Service`, `@Repository`, ou déclarée dans une config `@Bean`
- être détectée par le scan du conteneur Spring

Serializable and no arg constructor. On peut set les bean dans une classe `@Configuration`.

`@Autowired` sur un attribut : ne pas faire ! Seulement dans les cas de test

Why ?

- **Injection cachée** → ce n'est **pas explicite** que la classe dépend de `MyDependency`.
- **Impossible de faire un test sans tricher** (pas d'injection via constructeur)
- **Pas immuable** → on peut modifier la dépendance plus tard (moins sûr)
- **Moins lisible / plus magique**
- **Difficile avec `final`** → tu ne peux pas marquer l'attribut comme `final`, donc tu perds en sécurité.

`@Autowired` sur une constructeur : OK

`@Value` :

@Value

- Injection of a value into a bean property
- Located before an attribute, setter, constructor parameter
- Example on a attribute

```
@Component  
public class MovieDAOCsv implements MovieDAO {  
  
    @Value("mymovies.txt")  
    private String filename;
```

```
@Value("${bank.host.baseurl}")  
private String bankHostandPort;
```

N-Tier architecture (souvent 3 ou 5 couches) : avec 3, présentation (UI), domain (logique) et Data Source (DB)

Rappel des 3 couches (modèle classique 3-tier) :		
Couche	Rôle principal	Exemples
⌚ Presentation Layer	Gère les interactions utilisateur / client	Controller , REST API , HTML , React , etc.
🧠 Domain Layer	Contient la logique métier (services, règles)	Service , Manager , UseCase classes
🗄 Data Layer	Gère l'accès aux données, persistance	Repository , DAO , JPA , DB access

Règles à suivre pour les controllers :

Golden rules of REST Controllers

1. Thou shalt not implement business logic in a controller. The controller is handling interoperability (transferring information back and forth), handling exceptions to return appropriate status codes, and coordinating call to business components. That's all.
2. Thou shalt not make a controller stateful. The controller should be stateless. It should not keep *conversational state* information between the client and the server (i.e., some information that would oblige to keep the controller object to be the same to handle several distinct calls from the same client to the server). This enables to handle network failure, and to scale horizontally.

DTO (Data Transfer Object) : **objet simple** utilisé pour **transporter des données** entre différentes couches d'une application (ex: entre le backend et le frontend), **sans logique métier**.

Il contient généralement :

- Des **champs publics ou privés avec getters/setters**
- Aucun comportement (juste des données)
- Parfois des annotations (ex: `@JsonProperty` , `@NotNull`)

Objective: avoid tight coupling on data and ensure data integrity/security

- decouple the domain models from the presentation layer (allowing both to change independently)
- encapsulate the serialization logic
- potentially reduce the number of method calls (lowering the network overhead)

Design: simply convert in and out some internal data model to a specific data model for transport outside of the application

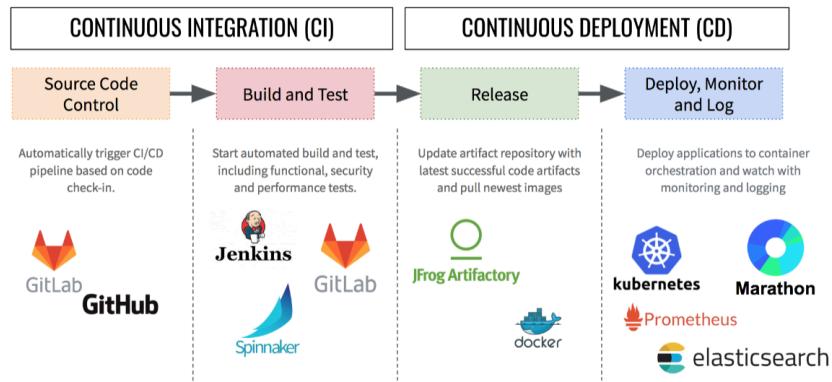
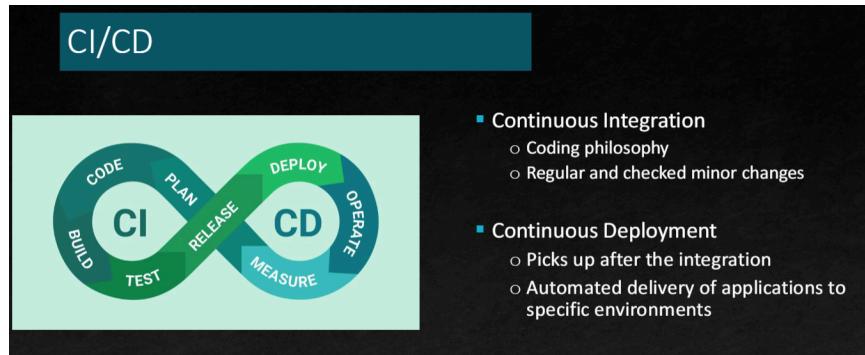
Implementation:

- Could be very simple (removing some attributes)
- Could be more complex (building a different representation/model from several objects)

Repositories : technical facade to the data source

Connectors : proxy interfaces to external systems (with DTO as input/output)

CI/CD :



REST : REpresentational State Transfer. Media type is JSON by default, standard HTTP methods/verbs for transactions

ORM : une technique (et souvent une librairie) qui permet de mapper des objets Java (ou d'un autre langage) à des tables d'une base de données relationnelle.

Il permet de manipuler les données via des objets, sans écrire de SQL directement. Une entité User liée à la DB par exemple.

ElementCollection :

`@ElementCollection` is a JPA annotation used when you want to **store a collection of basic types** (like `String`, `Integer`, etc.) or **embeddable objects** that **don't need their own table with an ID**.

It tells JPA:

“Hey, this collection belongs to this entity, but it’s stored in a separate table.”

When to use it?

Use `@ElementCollection` when:

- You want to store a list, set or map of **primitive types or simple objects**.
- These values don't deserve a full `@Entity` class (they have no identity).

```
@Entity
public class User {

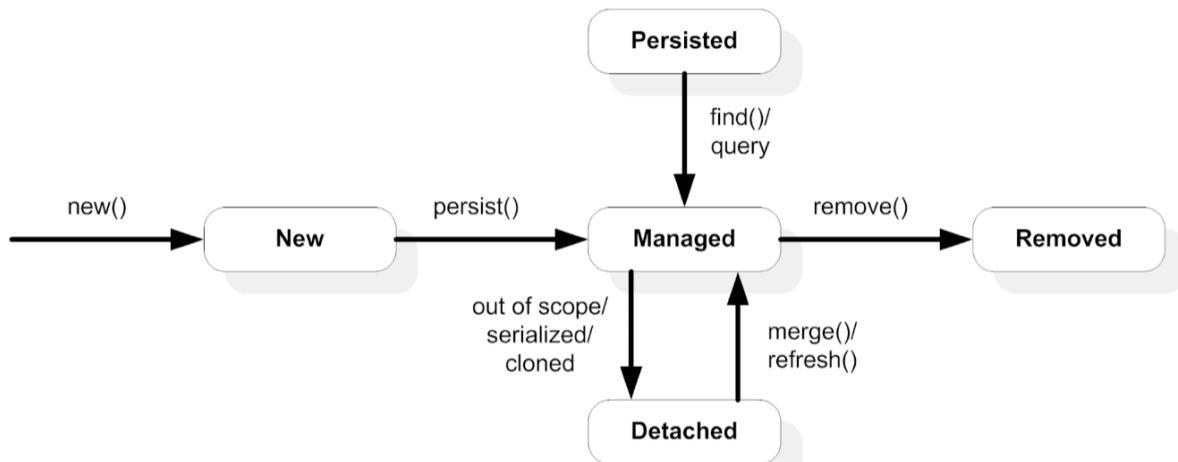
    @Id
    private Long id;

    private String name;

    @ElementCollection
    private List<String> phoneNumbers;
}
```

JPA Entity :

JPA Entity Lifecycle



End of transaction → entity detached

Spring Date hides the persistent context (`entityManager.merge` etc)

FetchTypes :

⌚ What is <code>FetchType</code> in JPA?	
<code>FetchType</code> controls when related data is loaded from the database:	
FetchType	Description
<code>EAGER</code>	Load immediately with the entity
<code>LAZY</code>	Load only when accessed (on demand)

You can use it on relationships like:

- `@OneToMany`
- `@ManyToOne`
- `@OneToOne`
- `@ManyToMany`
- `@ElementCollection`

.LAZY (Lazy loading)

java

```
@OneToMany(fetch = FetchType.LAZY)
private List<Order> orders;
```

Only fetches `orders` when you call `getOrders()`.

Uses a proxy behind the scenes.

💡 Good for performance if you don't always need the related data.

⚡ EAGER (Eager loading)

java

```
@ManyToOne(fetch = FetchType.EAGER)
private Customer customer;
```

Automatically fetches `customer` as soon as the parent entity is loaded.

Can cause **performance issues** (lots of joins or unnecessary data fetched) → the "N+1 problem".

💡 TL;DR

FetchType	When it fetches	Use when...
LAZY	On access (lazy load)	You don't always need the data
EAGER	Immediately (auto join)	You always need related data

Default Lazy loading behavior

Relationship	Default behavior	Entity retrieved
One-to-One	EAGER	Single entity
One-to-Many	LAZY	Collection
Element-Collection	LAZY	Collection
Many-to-One	EAGER	Single entity
Many-to-Many	LAZY	Collection

@Transactional tells Spring:

"Run this method inside a database transaction."

If anything goes wrong (like an exception), **rollback** all changes automatically.

@Commit in a test : every transaction that is made

Il y a 2 mois dans # si4-isadevops - isa_4_1_SpringTesting.pdf

9

king (the Bank Proxy)

10

with 2 databases (dev, test)

11

// SpringBoot (not Spring) will initialize JPA and DB connection with the following properties:
 // spring.datasource.username
 // spring.datasource.password
 // spring.datasource.url

Test default DB : H2 database
An in-memory Java-implemented DB

src/main/resources/persistence.properties

```
# postgres DB # IN DOCKER COMPOSE SHOULD BE OVERRIDDEN BY ENV VARIABLES
# POSTGRES_USER
spring.datasource.username=postgresuser
# POSTGRES_PASSWORD
spring.datasource.password=postgrespass
spring.datasource.url=jdbc:postgresql://${POSTGRES_HOST}/tcf-db
spring.datasource.driver-class-name=org.postgresql.Driver

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.show-sql=true
spring.jpa.generate-ddl=true

spring.jpa.open-in-view=false
```

src/test/resources/persistence.properties

```
# H2 in-memory DB
# The default setup in SpringBoot could be used
# we show here the equivalent setup of the default configuration

spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
# DB_CLOSE_EXIT set as false to keep H2 database alive at cucumber runtime :)
spring.datasource.url=jdbc:h2:mem:db;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE
spring.jpa.database-platform: org.hibernate.dialect.H2Dialect
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.h2.console.enabled=true
spring.h2.console.path=/console/
spring.jpa.open-in-view=false
```

Ensures Controllers do not start a default transaction (bad conceptually and bad for performance)

Could have been omitted

(last blue sentence is important : ensure...)



Why this happens in tests

In many unit/integration tests:

- You load an entity (e.g., via `repository.findById(...)`)
- The transaction ends
- Later in the test, you access a **lazy-loaded field** → **Boom, session is closed.**

Add transactional to the test to stop this from happening or always fetch eagerly

Exemple test Controller

Mocking the business component

```
@Test
void recipesRestTest() throws Exception {
    when(mockedCat.listPreMadeRecipes())
        .thenReturn(Set.of(Cookies.CHOCOLALALA, Cookies.DARK_TEMPTATION)); // only 2 of the 3 enum values

    mockMvc.perform(get(RecipeController.BASE_URI)
        .contentType(APPLICATION_JSON))
        .andDo(print())
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.isArray()"))
        .andExpect(jsonPath("$.size()", hasSize(2)))
        .andExpect(jsonPath("$.item(0).name()", hasItem("CHOCOLALALA")))
        .andExpect(jsonPath("$.item(1).name()", hasItem("DARK_TEMPTATION")));
}
```

Hitting the API on the route

Usage of Builders, Matchers, Handlers

On peut ajouter un port pr brancher un serveur de test si besoin

```

        Add a first response in the queue
        ↗
@Test
void recipesSetTest() throws Exception {
    // Given
    mockWebServer.enqueue(new MockResponse()
        .setBody("[\"CHOCOLALALA\", \"DARK TEMPTATION\", \"SOO_CHOCOLATE\"]")
        .addHeader("Content-Type", "application/json"));

    // When-Then
    assertEquals(EnumSet.allOf(CookieEnum.class), client.recipes());
}

// Verify the request was made to the correct endpoint
RecordedRequest recordedRequest = mockWebServer.takeRequest();
assertEquals("/recipes", recordedRequest.getPath());
assertEquals("GET", recordedRequest.getMethod());
}

```

Calling and Asserting

Additional verification if

`@Commit` is a **Spring Test annotation** used to say:

"Commit the transaction after the test finishes"

(instead of rolling it back like usual)

💡 By default in Spring tests:

- Each test runs inside a **transaction**
- That transaction is **rolled back automatically** at the end
- This keeps your database clean between tests — which is good ✓

💡 Use cases for `@Commit` :

- When you're **manually verifying the DB content** after the test (e.g. debugging)
- When one test sets up data used by another test (⚠️ not ideal, but possible)
- When you want to **reproduce a bug** involving committed state

⚠️ Important Notes:

- `@Commit` only works **with `@Transactional`** tests.

- The opposite of `@Commit` is `@Rollback` (which is actually the default).
- Be cautious: tests that leave behind data can cause flaky test behavior.

Propagation :

Behavior	Description
REQUIRED	The called method have to be executed in an existing transactional context. If this last does not exist, it will be created.
SUPPORTS	The called method may be executed in an existing transactional context. In case where no transactional context exists, the method will be executed even so (without transactional context).
MANDATORY	The called method have to be executed in a transactional context. In the opposite case an exception will be thrown.
REQUIRES_NEW	The called method requires the creation of a new transaction.
NOT_SUPPORTS	The called method does not support transaction. If a transactional context exists, it will be suspended.
NEVER	The called method should never be executed in a transactional context. In the opposite case an exception will be thrown.
NESTED	If a transactional context exists while a method is called, this last will be executed in a nested transaction.

Mandatory is the important one, inforce safety

Embeddable :

@Embeddable

```
public class Address {  
    protected String street;  
    protected String city;  
    protected String zipcode;  
}
```

does not need
an ID

@Entity

```
public class User {  
    @Id  
    protected Long id;
```

Shared
Identifier

@Embedded

```
protected Address address;
```

Cascade types :

Cascade Type	What it does
PERSIST	Saves related entities when parent is saved
MERGE	Merges changes of child when parent is merged
REMOVE	Deletes child entities when parent is deleted
REFRESH	Reloads child from DB when parent is refreshed
DETACH	Detaches child when parent is detached from persistence context
ALL	Applies all of the above

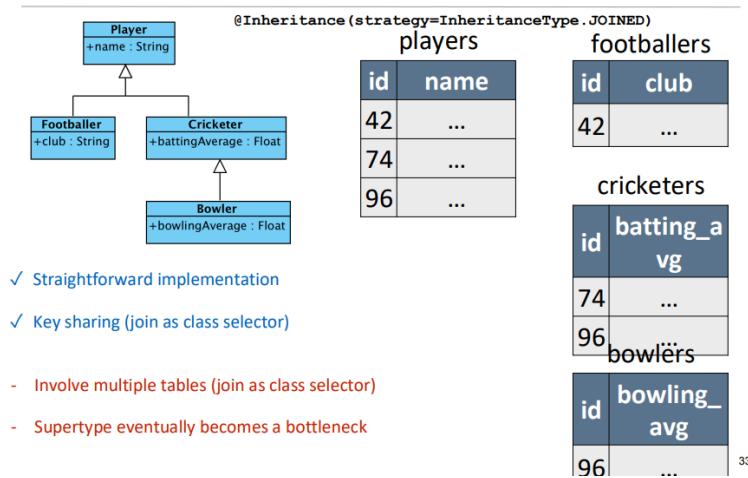
@Inheritance(strategy=InheritanceType.JOINED):

This annotation is used in **JPA** to map **inheritance hierarchies** (i.e., Java class inheritance) into **database tables**.

It tells JPA **how to store parent and child classes in relational tables**.

Strategy	Tables	Data Stored	Performance
JOINED	1 per class	Parent + children separately	✓ Good for write-heavy, normalized schema
SINGLE_TABLE	1 table total	All fields + discriminator	✓ Fast reads, ✗ lots of nulls
TABLE_PER_CLASS	1 per subclass	Each subclass is standalone	✗ Poor for querying by parent type

Solution #2: Class-Table Inheritance



- ✓ Straightforward implementation
- ✓ Key sharing (join as class selector)
- Involve multiple tables (join as class selector)
- Supertype eventually becomes a bottleneck

33

Revisions basées sur les questions annales

21:07 Mardi 8 avril

Do List | Représentation Dyna... | Attention Multi Têtes | Revisions

Révisions

→ ORM & persistence

ORM : objet relational mapping

entity needs :

- an empty constructor
- a proper equals method that relies on business elements
- a proper hashCode method

Relationship:

- OneToOne 1-1
- OneToMany 1-n
- ManyToOne n-1
- ManyToMany n-n

client à plusieurs commandes

manyToOne (mapp...) @onetomany (mappedBy = customer)

Cascade : permet de propager certaines opérations d'une entité vers ses enfants associés

PERSIST : quand tu sauvegarde le parent les enfants aussi

MERGE : utilise quand update

REMOVE : pour tout supprimer

REFRESH : remet les données à jour depuis la base

DETACH : détache les entités du contexte de persistance

ALL

21:07 Mardi 8 avril

Do List Représentation Dyna... Attention Multi Têtes Revisions

5) Stateless : sans état
↳ le serveur ne garde aucune information sur le client entre 2 requêtes, chaque requête doit être complète et indépendante.
↳ moins de mémoire utilisée
↳ facile à écheler horizontalement

7) le principe de cascading permet de propager automatiquement certaines opérations d'une entité vers ses entités liées

8) Artifactory : gère un dépôt pour stocker, versionner et partager des artifacts
↳ centraliser tous les fichiers build
↳ tracer et versionner les releases
↳ gérer les accès

9) Transaction avec mandatoriness
↳ si une méthode à sa c'est quelle doit être appelée dans une méthode parente qui a déjà commencé une transaction.
↳ elle ne peut pas être une transaction elle-même sinon une exception est levée

ex :

```
@Service
public class PaiementService {
    @Transactional
    public void traiterCommande() {
        vérifier_stock()
        débiter_compte();
    }
    @Transactional(Mandatory)
    public void débiter_compte() {
        compte_repo.debiter(..);
    }
}
```

cas correct : paiementService.traiterCommande()
cas faux : paiementService.débiterCompte() exception levée

21:07 Mardi 8 avril

Do List | Représentation Dyna... | Attention Multi Têtes | Revisions

1) Isa-devops
Isa : ingénierie des systèmes d'information et application
↳ couvre la conception, le développement et la gestion des systèmes d'information
DevOps : développement + opération,
↳ approche visant à unifier le Dev et l'admin système et une partie consiste à exploiter pour automatiser et accélérer le cycle de vie des applications

2) DTO : data transfer object
↳ objet utilisé pour transporter des données entre différentes couches (back / front)
* Avantages :

- évite l'exposition direct des entités JPA
- permet de contrôler les données exposées
- allège les objets transférés

* Inconvénients :

- besoin d'écrire des classes supplémentaires
- doit être synchronisé avec les entités
- peut allonger le code

3) 3 types de tests différents :
- tests unitaires : teste une seule unité de code (une méthode isolée)
- test d'intégration : vérifie que plusieurs composants fonctionnent bien entre eux (controller + service + Repo)
- test E2E : simule un scénario complet du front jusqu'à la base.

4) CI/CD : continuous integration / continuous deployment
automatiser les tests et la validation du code dès qu'il est poussé
↳ déploiement automatisé vers un dépôt, un environnement préprod jusqu'en prod
∞ : plan → code → build → test → release → deploy → operate
→ monitor

21:37 Mardi 8 avril

Apprentissage Auto... Tokenization Embedding Revisions

Diagramme de classe

→ en iso devops on met pas les méthodes uniquement les attributs

→ je savais pas ⚡ et ⚡

```

classDiagram
    class ListeDeLecture
    class Morceau
    ListeDeLecture "1..*" --> "1..*" Morceau
  
```

Ici on peut avoir plusieurs listes de lecture avec au moins un morceau dedans
 ↳ si on vient à supprimer la liste de lecture les morceaux seront pas supprimés.

```

classDiagram
    class Album
    class Morceau
    Album "1" --> "1..*" Morceau
  
```

Ici 1 album est unique et peut contenir 1 ou plus morceaux
 ↳ si on supprime l'album les morceaux le sont aussi (correspondance avec ~~cas de type Remove~~)

1) Différences entre objet métier et composant métier

- représente une entité /ée ou / représente la logique métier (classe avec attribut)
- une donnée
- une logique métier souvent des services

Différences entre objet métier et objet

un objet peut être une classe un dto, un objet métier mais pas l'inverse

21:37 Mardi 8 avril

Apprentissage Auto... Tokenization Embedding Revisions

82 %

3) stateful : il garde un état interne entre les appels (ex compteur)
↳ l'inconvénient pas de threadsafe → incohérence, difficile à débug ou à scalar.
↳ avantage peut être utile si tu as besoin de maintenir une session utilisateur

4) Intercepteur c'est quoi ?
↳ composant qui agit avant ou après une requête
ex : authentification, logging des requêtes, gestion des headers ou cookie.

5) Avantages docker-compose (hormis lancer plusieurs images en même temps) :

- définir les réseaux internes entre les services
- configuration de variable d'environnement
- gérer des volumes
- lancer tout un environnement de dev (avec des règles)
- script de déclenchement automatique ↗
- orchestration (dépendance, restart) ↙

8 sur 8

Questions/à revoir

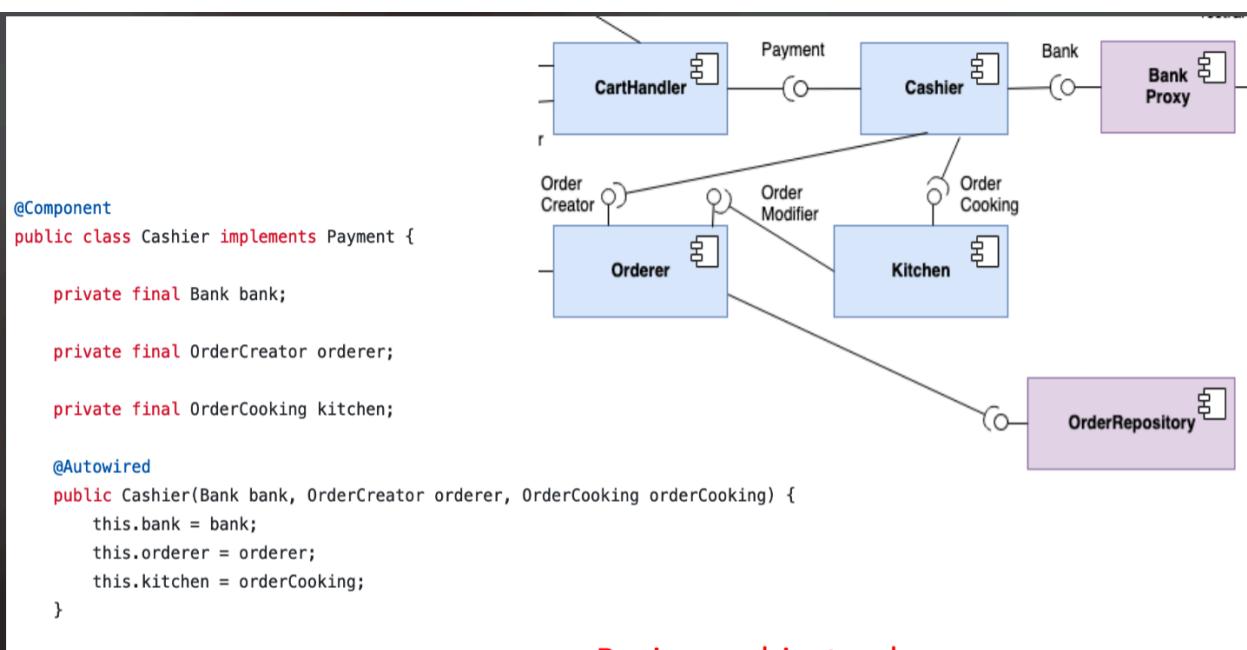
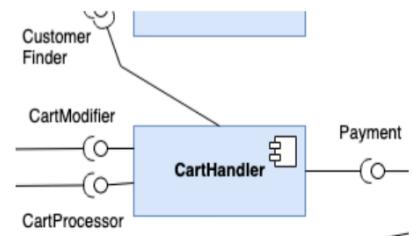
Diagramme de component exemple :

```
@Service
public class CartHandler implements CartModifier, CartProcessor {
    private static final Logger LOG = LoggerFactory.getLogger(CartHandler.class);

    private final Payment payment;

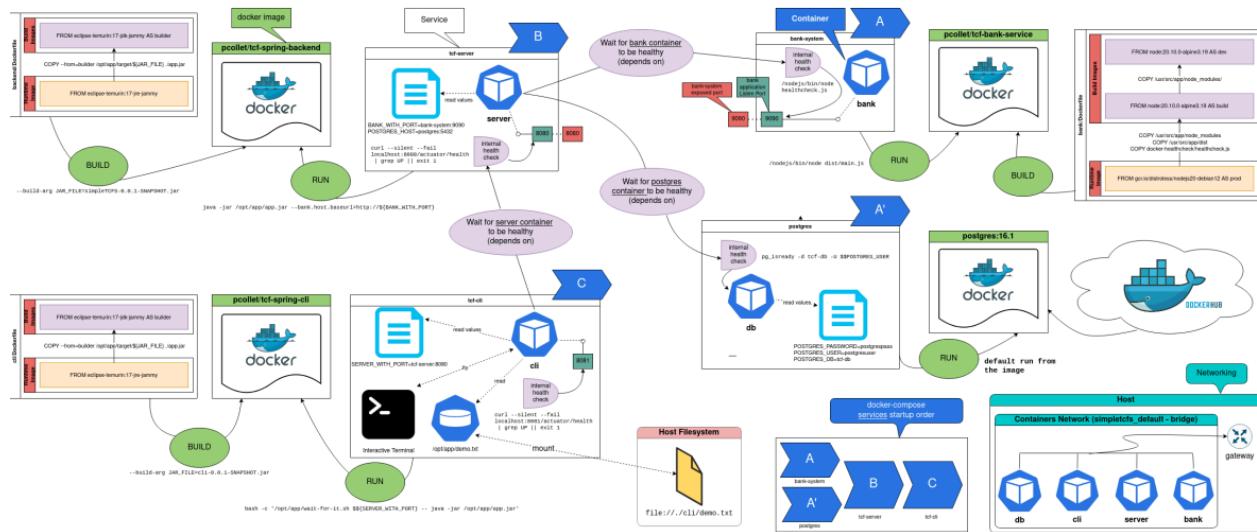
    private final CustomerFinder customerFinder;

    @Autowired
    public CartHandler(Payment payment, CustomerFinder customerFinder) {
        this.payment = payment;
        this.customerFinder = customerFinder;
    }
}
```



TCF Architecture (slide 1.4)

To ease understanding of the docker composition provided, the following schema illustrates all main elements:



Cascade Type	What it does
PERSIST	Saves related entities when parent is saved
MERGE	Merges changes of child when parent is merged
REMOVE	Deletes child entities when parent is deleted
REFRESH	Reloads child from DB when parent is refreshed
DETACH	Detaches child when parent is detached from persistence context
ALL	Applies all of the above

Intégration continue : chaque fois qu'un développeur pousse du code, il est automatiquement testé, compilé, analysé.

Préparer le code pour être déployé automatiquement sur un environnement (staging, préprod, etc.), mais le **déploiement final est encore manuel**.