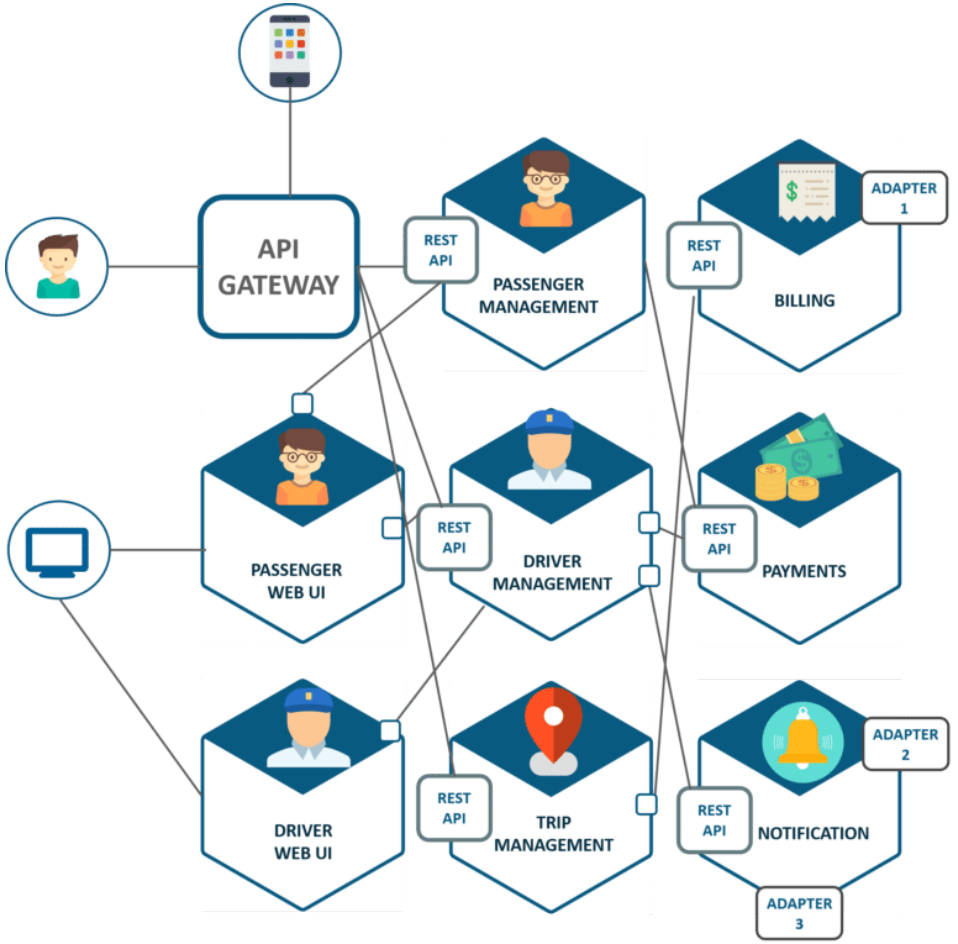
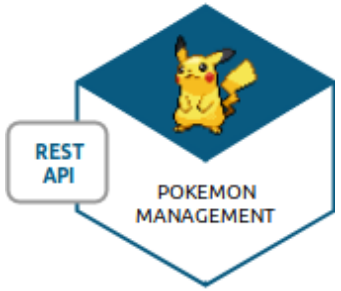
ALOM

UBER


API
GATEWAY
REST
API
PASSENGER
MANAGEMENT
REST
API
BILLING
ADAPTER
1
PASSENGER
WEB UI
REST
API
DRIVER
MANAGEMENT
REST
API
PAYMENTS
DRIVER
WEB UI
REST
API
TRIP
MANAGEMENT
REST
API
NOTIFICATION
ADAPTER
2
ADAPTER
3

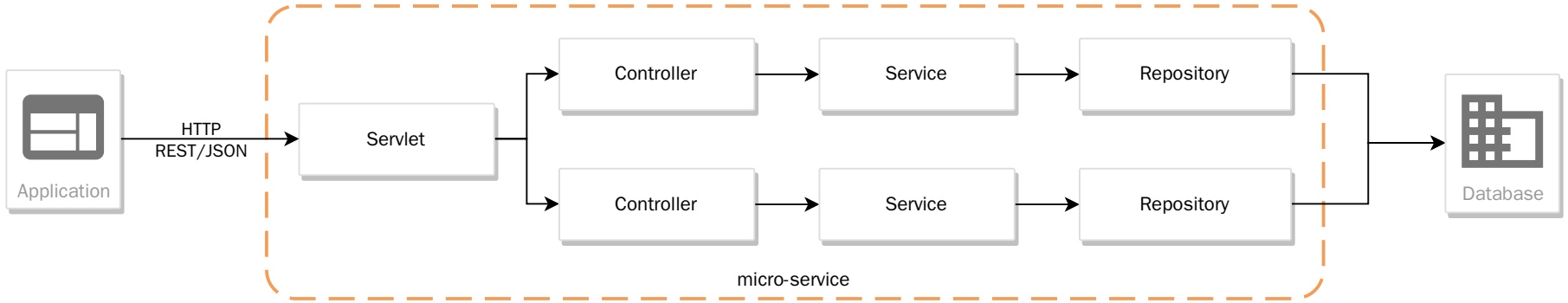
UN MICRO-SERVICE C'EST :

- Un ensemble de fonctionnalités du même domaine métier
- Un ou plusieurs canaux de communication
  - HTTP - REST/JSON
- Une source de données dédiée



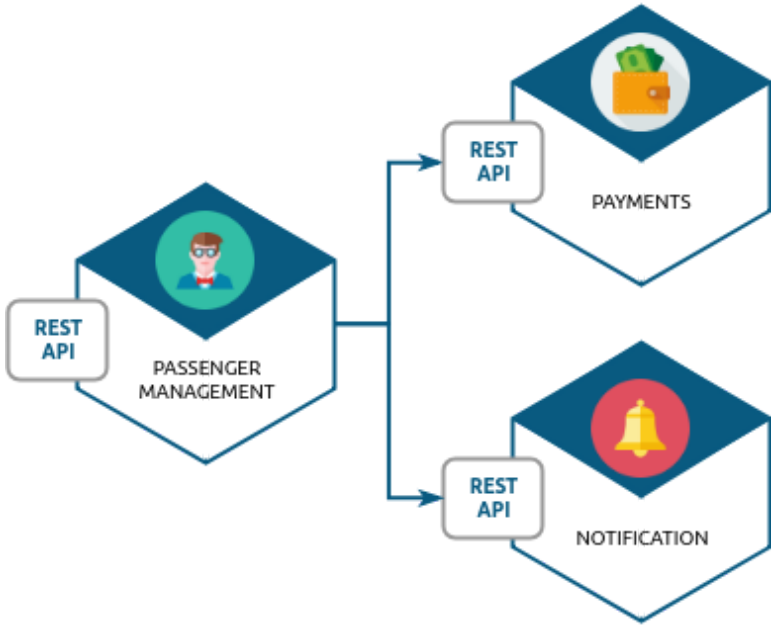
🔎 UN MICRO-SERVICE JAVA

On s'appuie sur les technologies connues: les servlets !

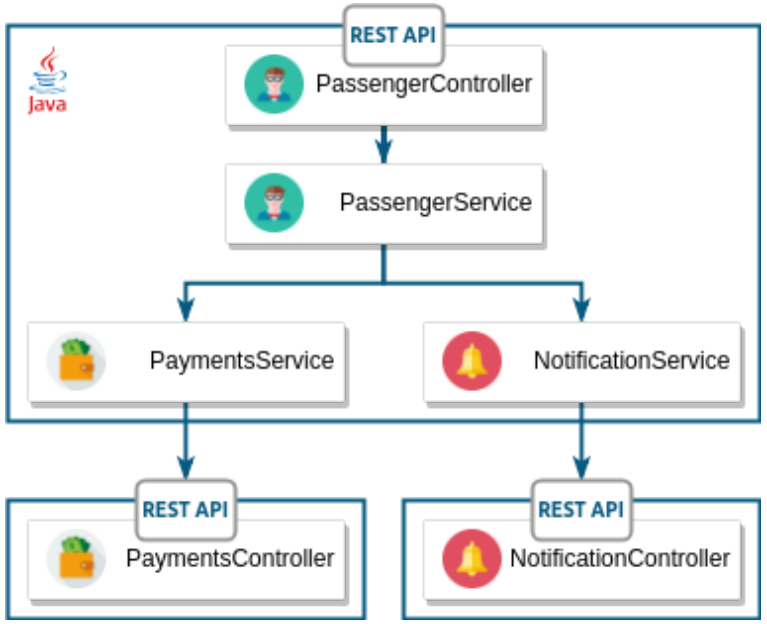


DEPENDENCY INJECTION

🚗 UN MORCEAU D'UBER



LA VISION ARCHITECTURE



🤔 LE CODE DU NOTIFICATIONSERVICE

```
class NotificationService {
    private MailService mailService = new MailService();
    private Paypal paypal = new Paypal();
    void notify(Event event){
        this.mailService.sendMail(event.passenger(), event.body());
    }
    void payForTrip(Trip t){
        this.paypal.requiredPayment(t.passenger().email(), t.cost());
    }
}
class MailService {
    void sendMail(String to, String content){
        [...]
    }
}
```

❓ Quel est le problème de ce code?

👮 S.O.L.I.D PRINCIPLES

- S : Single Responsability
  Une classe doit avoir une seule responsabilité
- O : Open/Closed
  Ouvert à l'extension, mais fermé à la modification
- L : Liskov Substitution
  Pouvoir utiliser un sous-type
- I : Interface Segregation
  Présenter plusieurs interfaces spécifiques
- D : Dependency Inversion
  Dépendre d'abstractions, et non d'implémentations

👮 IS IT S.O.L.I.D ?

```
class NotificationService {
    private MailService mailService = new MailService();
    private Paypal paypal = new Paypal();
    void notify(Event event){
        this.mailService.sendMail(event.passenger(), event.body());
    }
    void payForTrip(Trip t){
        this.paypal.requiredPayment(t.passenger().email(), t.cost());
    }
}
class MailService {
    void sendMail(String to, String content){
        [...]
    }
}
```

S ❌

O ❌

L ❌

I ❌

D ❌

🚑 REFACTORING !

```java
public interface NotificationService {
    void notify(Event event);
}
class NotificationServiceImpl implements NotificationService {
    private MailService mailService;
    void setMailService(MailService mailService){
        this.mailService = mailService;
    }
    void notify(Event event){
        this.mailService.sendMail(event.passenger().mail(), event.body());
    }
}
public interface MailService {
    void sendMail(String to, String body);
}
class MailServiceImpl implements MailService {
    void sendMail(String to, String body){
        [...]
    }
}
```

S	✅
O	✅
L	✅
I	✅
D	✅

Rendre notre code S.O.L.I.D :

- Le rend testable
- Le rend compatible avec de l'injection de dépendances

INJECTION DE DÉPENDANCE

Laisser la plateforme fournir les dépendances:

- En fonction du contexte
- En fonction des composants disponibles

Nécessite des efforts de conception objet!

INJECTION DE DÉPENDANCES

AVEC spring boot

```
public interface NotificationService {
    void notify(Event event);
}
@Service
class NotificationServiceImpl implements NotificationService {
    private MailService mailService;
    @Autowired
    NotificationService(MailService mailService){
        this.mailService = mailService;
    }
    void notify(Event event){
        this.mailService.sendMail(event.passenger().mail(), event.body());
    }
}
public interface MailService {
    void sendMail(String to, String body);
}
@Service
class MailServiceImpl implements MailService {
    void sendMail(String to, String body){
        [...]
    }
}
```

MIAGE Lille

INJECTION DE DÉPENDANCES

3 moyens :

- Par le construteur (recommandé)
- Par les setters (bof, non-immutable)
- Par les attributs de classe (déconseillé)

⚠️ JAMAIS DE `@Autowired` SUR DES ATTRIBUTS DE CLASSE

- Impossible à tester unitairement
- Risque de démultiplier les dépendances d'une classe (par fa
- Ne fonctionne pas sur les JVM renforcées avec un Security M
  - https://docs.oracle.com/en/java/javase/21/docs/api/jav

ARCHITECTURE D'UNE APPLICATION EN COUCHES

```
.
├── pom.xml
└── src
    └── main
        ├── java
        │   └── fr
        │       └── univ_lille
        │           └── alom
        │               ├── Application.java  // main
        │               ├── config
        │               ├── controllers
        │               │   ├── PokemonController.java
        │               │   └── TrainerController.java
        │               ├── model
        │               │   ├── Pokemon.java
```

⚠️ implique que tout le code soit `public`

Dépendances entre packages claires, mais

ARCHITECTURE D'UNE APPLICATION EN FEATURE

```
.
├── pom.xml
└── src
    └── main
        ├── java
        │   └── fr
        │       └── univ_lille
        │           └── alom
        │               ├── Application.java
        │               ├── config
        │               ├── pokemons
        │               │   ├── PokemonController.java
        │               │   ├── Pokemon.java
        │               │   ├── PokemonRepository.java
        │               │   └── PokemonService.java
```

👍 permet de controller la visibilité des classes (`package-private` 💙)

GUIDELINES GÉNÉRALES

- ⚠️ jamais de lien direct entre `@Controller` et `@Reposit`
  - Sauf CRUD très simple. Attention à la gestion de transac
- ⚠️ jamais de `@Autowired` sur des attributs de classe
  - Impossible à tester unitairement
  - Ne fonctionne pas sur les JVM renforcées avec un Secur
  - https://docs.oracle.com/en/java/javase/21/docs/api/jav

TP

Jouer avec spring-boot !

FIN DU COURS

Cours suivant :

Persistance des données avec JPA