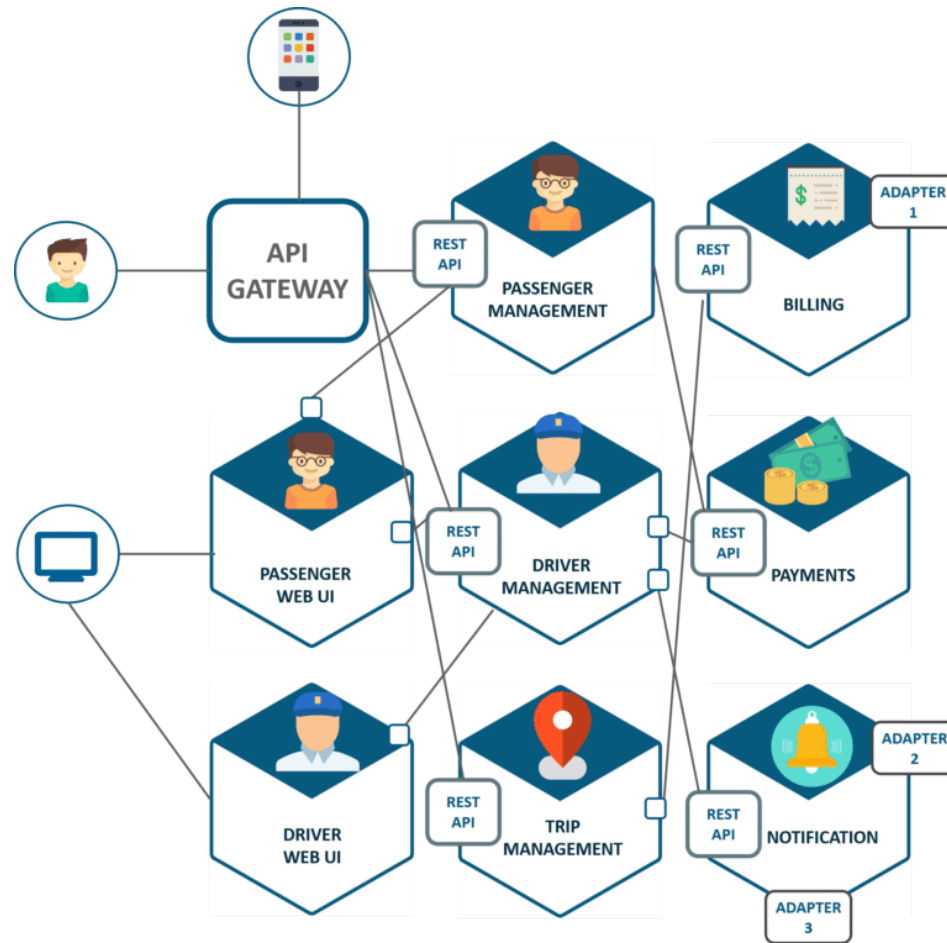


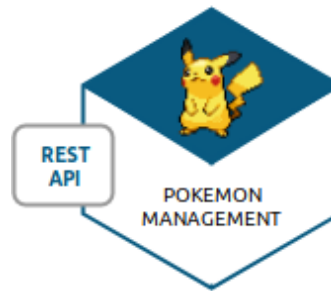
# ALOM





# UN MICRO-SERVICE C'EST :

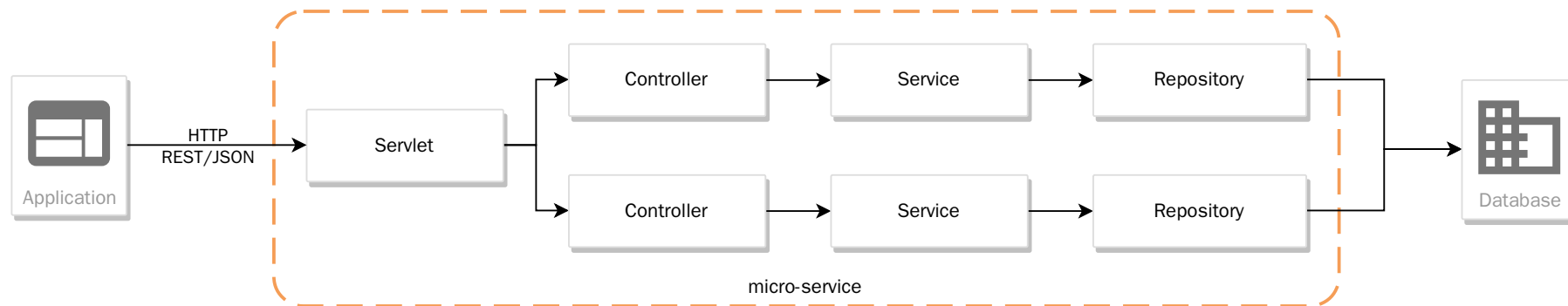
- Un ensemble de fonctionnalités du même domaine métier
- Un ou plusieurs canaux de communication
  - HTTP - REST/JSON
- Une source de données dédiée





# UN MICRO-SERVICE JAVA

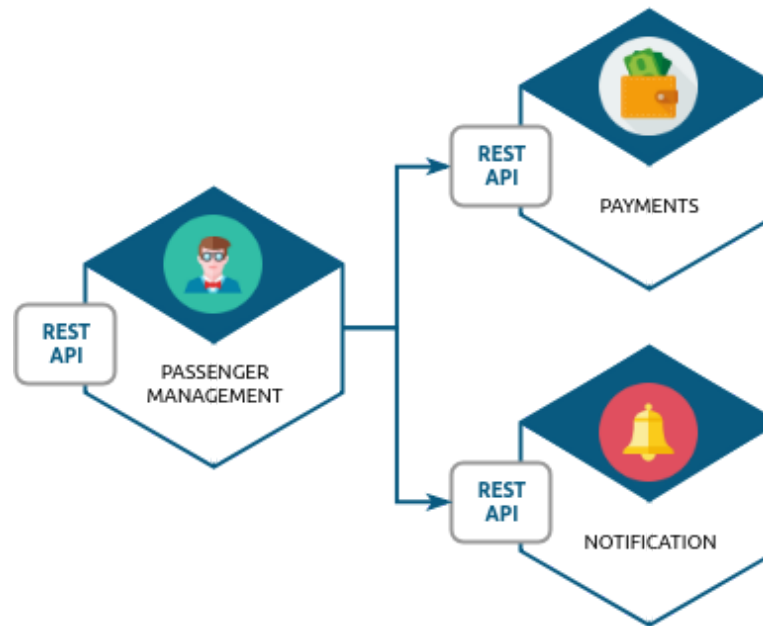
On s'appuie sur les technologies connues: les servlets !





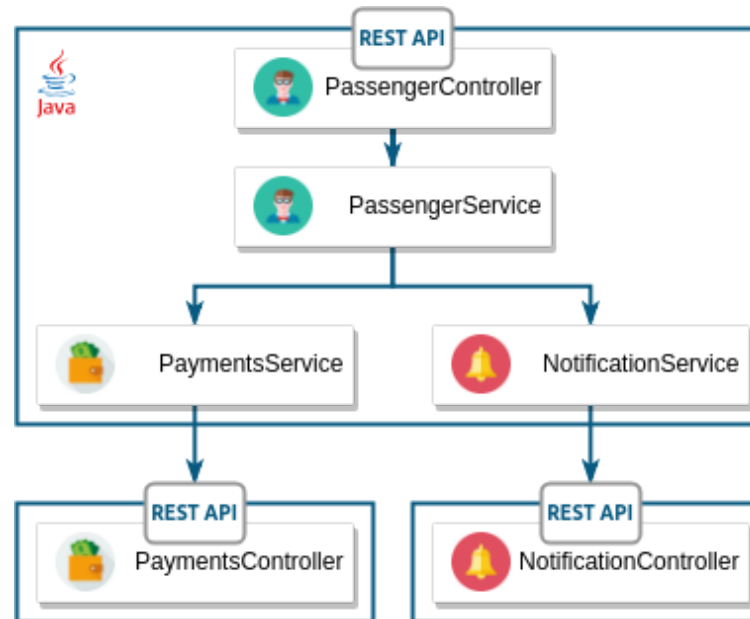
# DEPENDENCY INJECTION

# 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 Responsibility  
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 !



```
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){
        [...]
    }
}
```



# INJECTION DE DÉPENDANCES

- 3 moyens :
- Par le constructeur (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/java>

# SPRING-BOOT

"Extension" à Spring

Fournit la configuration automatique des frameworks  
les plus utilisés:

- spring-web
- spring-security
- jpa
- mongodb
- ...

# SPRING-BOOT

Les applications sont packagées dans des `jar` auto-portés.

Le `jar` contient le code de l'application, plus les librairies (spring-core, spring-web, etc...).

Les applications contiennent un `main` Java, qui démarre Spring-Boot.

Adapté à un écosystème  containers

# SPRING-BOOT

Un `pom.xml` parent:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.3.4</version>
  <relativePath/>
</parent>
```

Liste toutes les versions de dépendances  
possibles/compatibles.

# spring-boot-starter-web

- configure `spring-web`
- embarque un tomcat-embedded
- démarre l'application 'courante' dans le tomcat
- écoute sur le port `8080`
- configure `jackson-databind` pour retourner du JSON pour les contrôleurs annotés `@RestController`

# spring-boot-starter-web

Utilisation dans un projet :

```
<dependencies>
  ...
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  ...
</dependencies>
```

On ne précise pas la version, elle est dans le `pom.xml` parent 🤖 !

# SPRING-BOOT

On déclare un `main` java dans une classe annotée :

```
@SpringBootApplication
public class MyApp {

    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }

}
```

# 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  
dépendances spaghetti 🍝 entre les classes

# 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 contrôler la visibilité des classes  
(`package-private` ❤️). 💡 `@Repository` ne sort pas public pour forcer l'utilisation des `@Service`

# GUIDELINES GÉNÉRALES

- ⚠ jamais de lien direct entre `@Controller` et `@Repository`
  - Sauf CRUD très simple. Attention à la gestion de transactions
- ⚠ jamais de `@Autowired` sur des attributs de classe
  - Impossible à tester unitairement
  - Ne fonctionne pas sur les JVM renforcées avec un SecurityManager
  - <https://docs.oracle.com/en/java/javase/21/docs/api/java/lang/instrument/package-summary.html>

# TP



Jouer avec spring-boot !

# FIN DU COURS

Cours suivant :

Persistance des données avec JPA