# SPRING AOP
# (ASPECT ORIENTED PROGRAMMING)

2019-2020

ESPRIT thouraya.louati@esprit.tn UP JavaEE / .NET (Bureau E204)


Spring
Framework

# Objectifs et Acquis d'apprentissage



- À la fin de la séquence d'enseignement « **Spring AOP** (Aspect Oriented Programming) », les étudiants seront en mesure de :
  - **Décrire** et comprendre les éléments principaux de la **programmation orientée aspect (AOP)** qui ont fait la force de Spring et son positionnement par rapport à l'architecture Spring.
  - **Distinguer** les atouts de la programmation orientée aspect.
  - **Procéder** à l'implémentation et l'intégration de l'AOP dans un projet déjà implémenté.
  - **Reconnaître** les différents types ressources bibliographiques (documentation officielle de Spring AOP, ..)

# Plan et organisation de la séquence




- Séquence : **Spring AOP**
  - Charge : 3 heures
    - Évolution des modèles de programmation
    - Problématique:  cross cutting concerns
    - Solution: AOP
    - Définition de la Programmation Orientée Aspect **AOP**
    - **Avantages et inconvénients de l'AOP**
    - **AOP** vs **IoC** (ID)
    - Principes de l'AOP : **SoC** / **DRY** / **CrossCutting Concerns**
    - Implémentation : **JoinPoint, PointCut, Advice, Aspect, Weaving**
    - Type d'Advice: **Before, After, Around, After Returning, After Throwing**
    - Design Pattern **Proxy** implémenté par l'**AOP**
    - **TPs AOP**: Journalisation, Performance

# Introduction



**« Spring is an application Framework for the Java platform »**

Outils et de règles facilitant le développement d'applications


Principes généraux mis en œuvre dans Spring qui facilitent la vie des développeurs.

- Inversion du contrôle
- Injection de dépendances
- Programmation par aspects
- Approche MVC (Modèle – Vue - Contrôleur)

# Architecture

Javac
Java –jar
Class path
New
Charger état du Bean (request, sigleton)
Charger des bibliothèques

Instancier
Configurer
Assembler

Spring Boot

Annotations
@Configuration
@ComponentScan
---
@Component

Spring Core

Spring ApplicationContext

Sprign IOC Container

1 → Annotations

Configuration de l'Injection de Dépendance

Charger la configuration

POJO Java
Vos classes

Bean 1
@Service

Bean 2
@Controller

Bean 3
@Repository

Bean 4
@Component
@Aspect

Singleton
prototype
Request
Session
Global-session

JDBC - MySQL

2

3

Application configurée

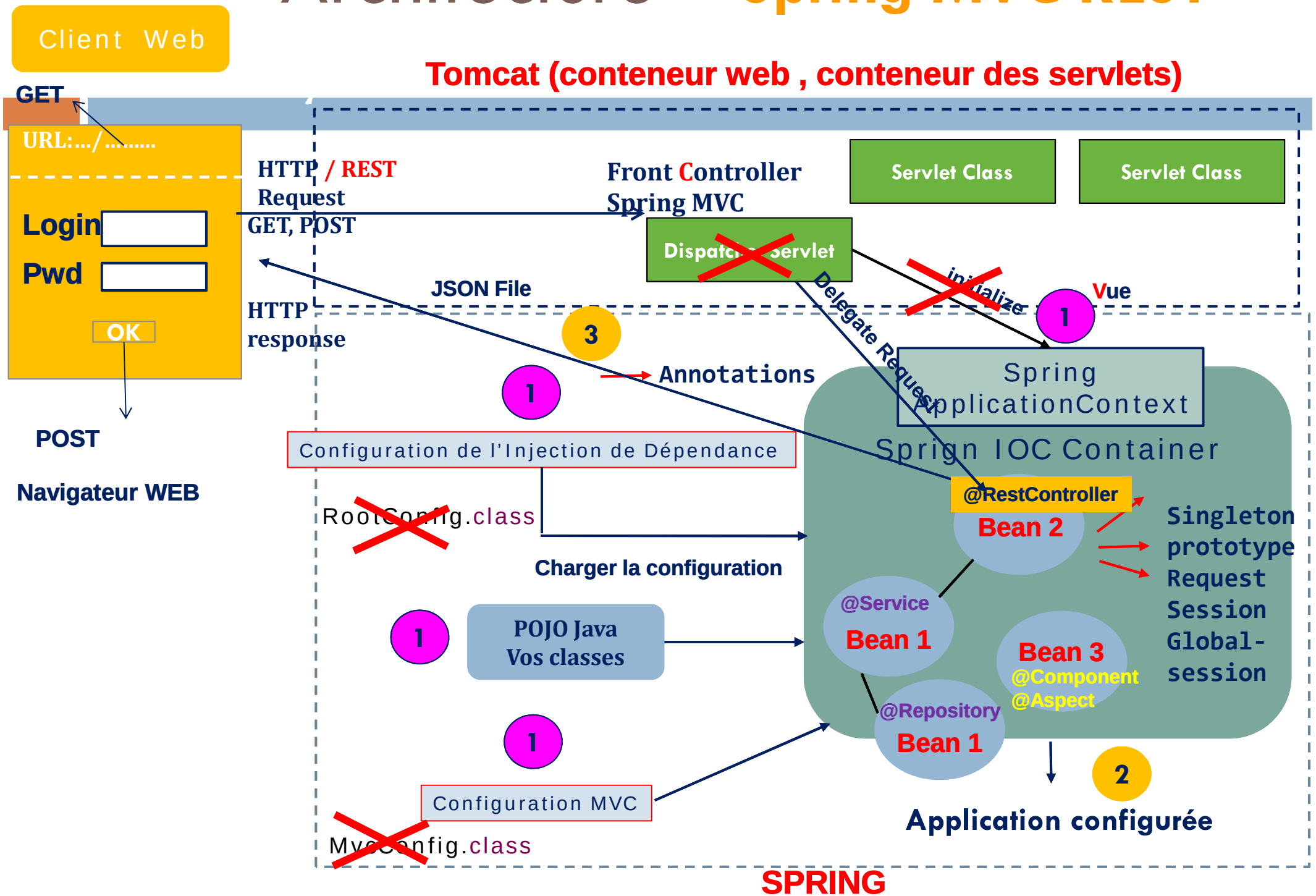
Spring Data

Spring MVC

SPRING

Sécurité
Logging
Joinpoint
Pointcut
Advice
Aspect
Weaving
Before
After returning
After throwing
After
Around

# Architecture – Spring MVC REST



# Évolution des modèles de programmation

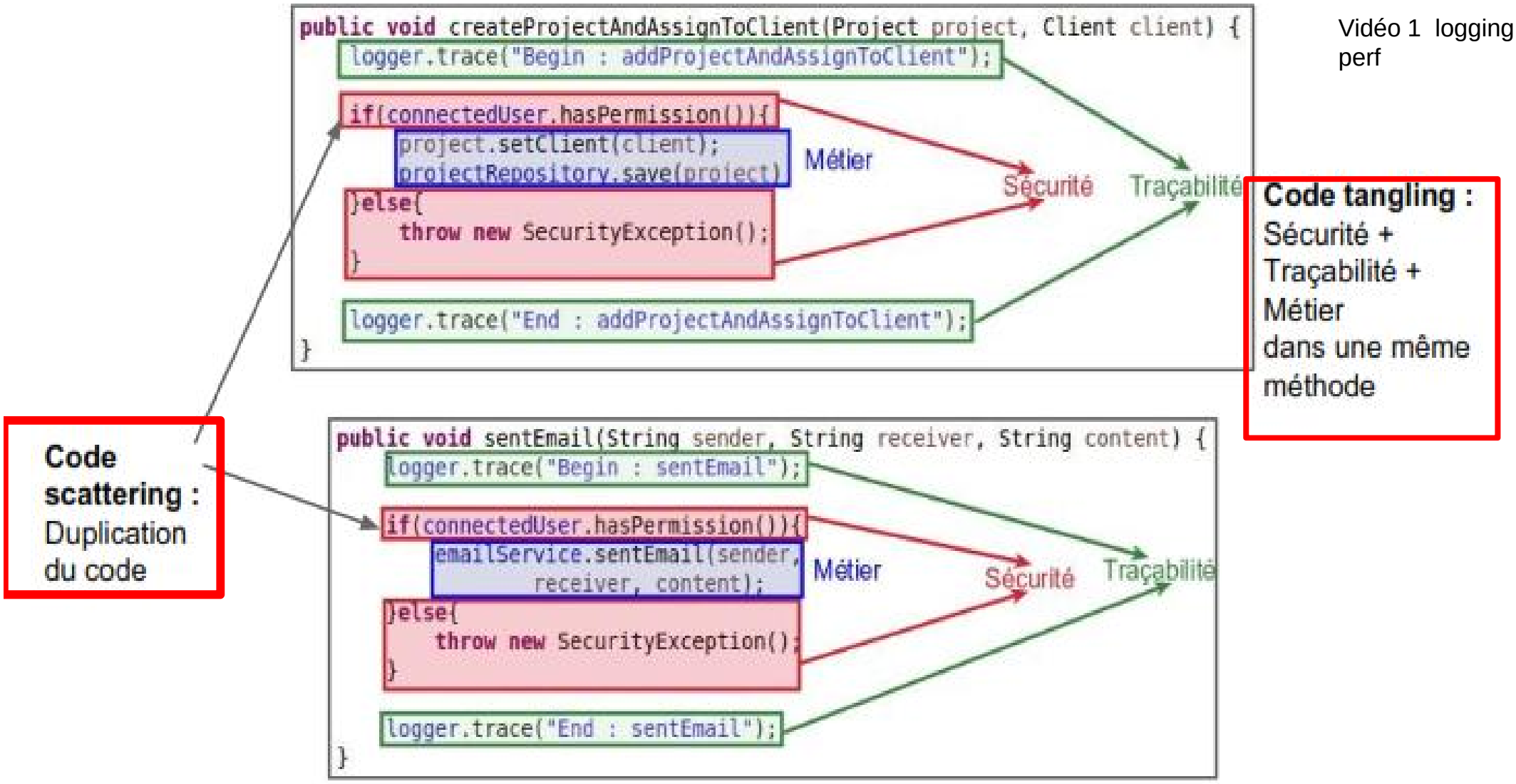


| Modèle | Préoccupation | Elément |
|---|---|---|
| Programmation procédurale | Découper le code en portions | Fonction, procédure |
| OOP: Programmation orientée objets | Données sous la forme d'objets | Classe |
| AOP: Programmation orientée aspects | Fonctionnalités transverses | Aspect |

+Maintenabilité
+Réutilisabilité

# Problématique : Cross Cutting Concerns

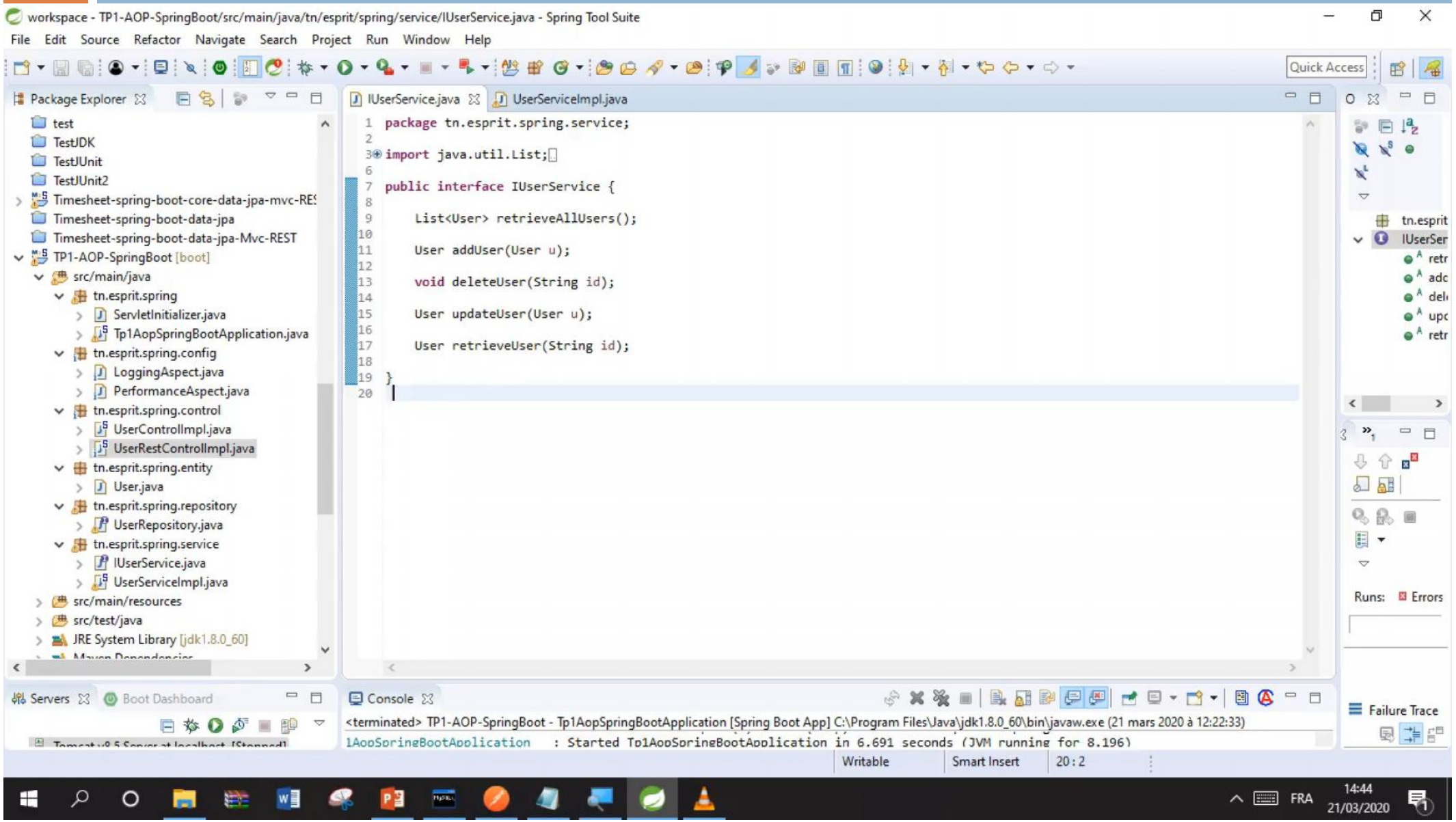





Vidéo 1 logging perf

# Problématique : Cross Cutting Concerns





# Problématique : Cross Cutting Concerns



- Exemple de « **cross-cutting concerns** »
  - Tracing, caching, Transaction, security, performance, monitoring, Error handling.
- Impact de « cross cutting concerns » sur notre système:
  - **Tangling:** Mélange du code métier avec du code de (sécurité, traçage,…).
  - **Scattering:** Duplication d'un bout de code dans plusieurs endroits. (tel que test de sécurité).

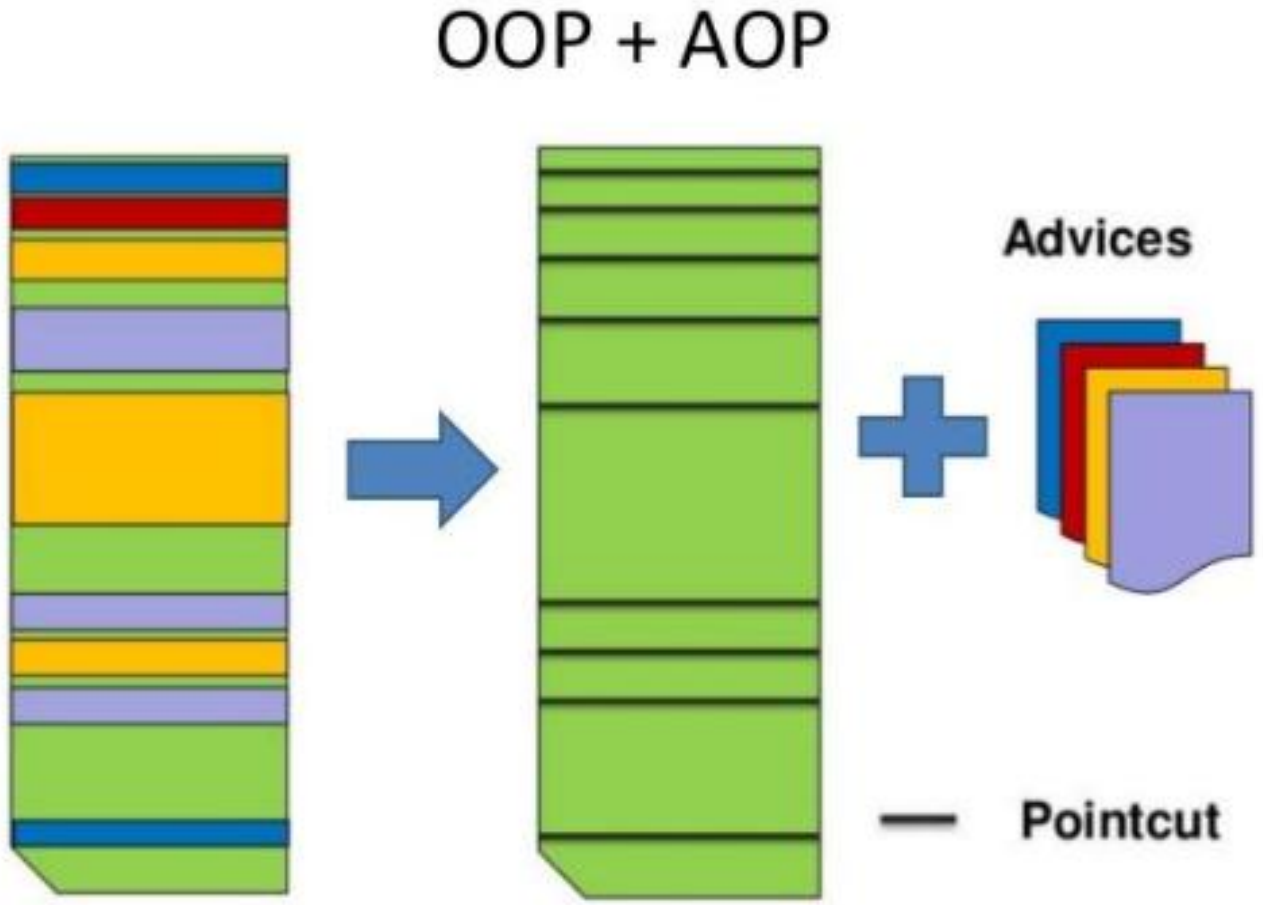
Problème de **maintenabilité.**

L'**AOP** peut être une **solution** à ces problèmes (**Tangling, Scattering**).

# Solution : Programmation Orientée Aspects AOP



- L'AOP vient pour factoriser les « cross-cutting concerns » sous forme d'aspect.



# SoC/ DRY/Cross cutting Concerns



- Separation of Concerns (**SoC**) : **Au lieu d'avoir un appel direct à** un module technique depuis un module métier, en AOP, le code du module en cours de développement est concentré sur le but poursuivi (la logique métier).
- L'AOP permet d'enrichir ce code métier avec ces fonctionnalités transversale (**Crosscutting Concerns**).
- Exemple : Ajout de logs dans une application existante.
- Don't Repeat Yourself (**DRY**) : **Cela évite la duplication de code.**

# Solution : Programmation Orientée Aspects AOP



- Permet de rajouter des comportements à des classes ou des méthodes existantes.
  - Ajouter des traces (logs),
  - Ajouter la gestion des transactions,
  - Ajouter la gestion de la sécurité,
  - Ajouter du monitoring,
  - Il s'agit de problématiques transverses (**Crosscutting concerns**), **en** général, techniques.

# Avantages et inconvénients de l'AOP



- Les avantages sont:
  - **Facilité de maintenance**, puisque les fonctionnalités transverses sont regroupées dans les aspects.
  - Particulièrement adapté pour les fonctionnalités techniques.
  - Permet une meilleure **modularité** du code et des applications ce qui augmente la réutilisation du code et la modularité des systèmes.
- Les inconvénients sont:
  - La lecture du code contenant les traitements ne permet pas de connaître les aspects qui seront exécutés (sans utiliser un outil).
  - Nécessite un temps de prise en main.

# Solution : Programmation Orientée Aspects AOP



- L'AOP peut-être utilisée:
  - Indirectement, lors de l'utilisation des annotations Spring, tel que, **@Configuration** et **@Transactional**.
  - Directement, pour mettre en œuvre ses propres Aspects : Spring facilite alors cette mise en œuvre.

# Mise en œuvre et implémentation

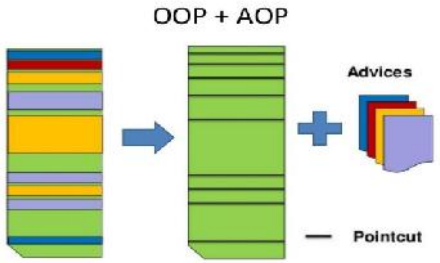
# Programmation Orientée Aspects AOP : Mise en œuvre



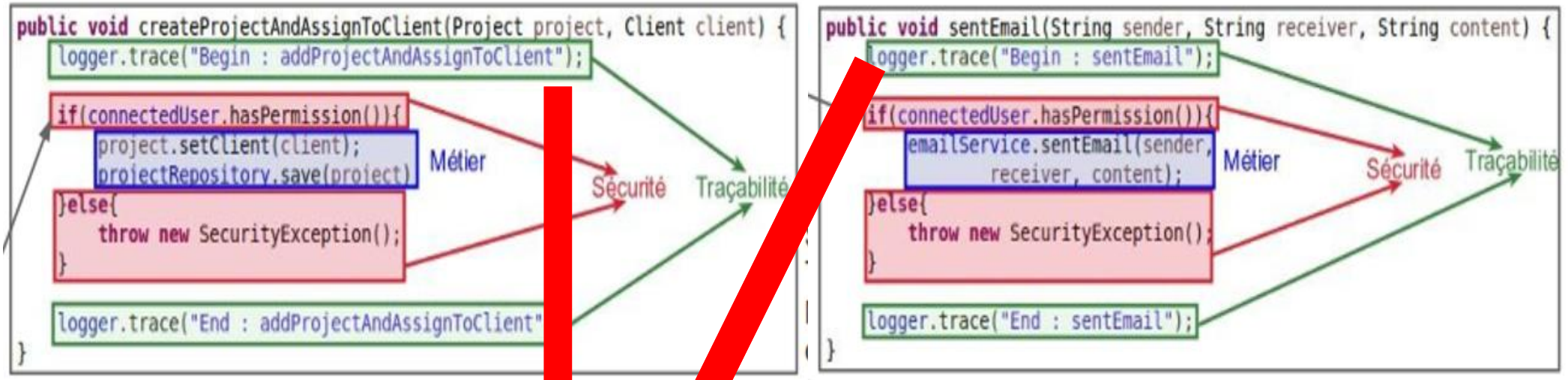
- L'AOP peut-être mise en œuvre via **Spring AOP** et/ou **AspectJ** de plusieurs manières:
  - Avec **AspectJ**, on utilise un tissage au chargement (Load Time Weaving) ou le compilateur AspectJ.
  - Avec **Spring AOP**, on a des limites telles que : ne peut appliquer des aspects **que sur** des beans spring.
  - Avec un mixte de **Spring AOP et AspectJ**

Dans ce cours, on va s'intéresser à **la mise en œuvre AspectJ.**

# Implémentation






```
@Component
@Aspect
public class LoggingAspect {
private static final Logger logger = Logger.getLogger(LoggingAspect.class);

@Before("execution(* tn.esprit.esponline.service.UserServiceImpl.*(..))")
public void logMethodEntry(JoinPoint joinPoint) {
String name = joinPoint.getSignature().getName();
logger.info("In method " + name + " : ");
}
}
```

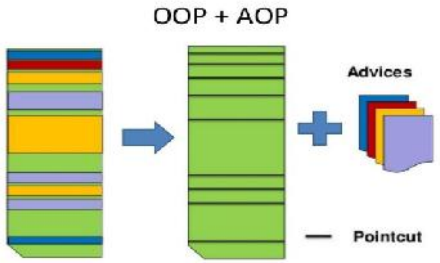
Type d'advice

Pointcut

Advice

Aspect

# Implémentation






```
@Component
@Aspect
public class LoggingAspect {
private static final Logger logger = Logger.getLogger(LoggingAspect.class);

@Before("execution(* tn.esprit.esponline.service.UserServiceImpl.*(..))")
public void logMethodEntry(JoinPoint joinPoint) {
String name = joinPoint.getSignature().getName();
logger.info("In method " + name + " : ");
}
}
```

Type d'advice

Pointcut

Advice

Aspect

```
@Service
public class UserServiceImpl implements UserService {
@Autowired
UserDAO userDao;
public String hello(String msg) { try {
Thread.sleep(200);
} catch (InterruptedException e) {
e.printStackTrace();}
String s = "Bonjour Esprit " + msg;  return s;  }
```

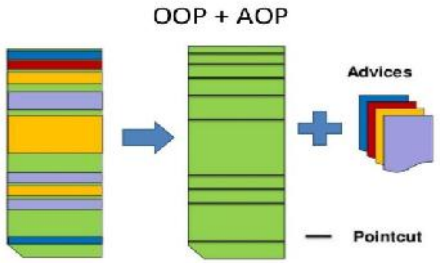
JoinPoint

# Solution : AOP



- L'AOP utilise le Design Pattern **Proxy.**
  - Un proxy est une classe se substituant à une autre classe. Le proxy implémente la même interface que la classe à laquelle il se substitue.
  - Dans notre cas, Spring va créer une classe "**proxy" qui implémente IUserService et** va l'injecter à la place du bean "**userServiceImpl".**
  - Cette classe proxy contient les aspects et les méthodes de l'interface.

# Implémentation






```
@Component
public class IdentityControllerImpl implements IidentityController {

@Autowired
@Qualifier("clientInfoSOAPServiceImpl")
private IClientInfoService clientInfoService;

private Logger logger = LoggerFactory.getLogger(IdentityControllerImpl.class);



@Override
public String getFullNameByClientId(int clientId){
//Appeler le webservice SOAP
return "Nom et prenom : " +
clientInfoService.getFirstNameByClientId(clientId) + " " +
clientInfoService.getLastNameByClientId(clientId);
}
}
```

Qui est injecté ?
→ C'est le proxy qui est injecté au moment de l'exécution.

# Implémentation de l'AOP



- **Joinpoint** (**point d'exécution**): L'endroit où l'on veut qu'un aspect s'applique; comme l'appel d'une méthode ou le lancement d'une exception.
- **Pointcut** (**point de coupe**): C'est l'expression qui permet de sélectionner un ou plusieurs **Joinpoints**. Par exemple, « toutes les méthodes public dans un paquet précis ».
- **Advice:** Le code que l'on veut rajouter. On peut ajouter ce code avant, après, autour de la méthode.
- **Aspect:** Une classe qui encapsule une fonctionnalité transverse et est composé d'un ou plusieurs **Pointcut** et **Advice**. La classe est annotée par @Aspect.
- **Weaving**(tissage): action d'insertion des aspects.

# Les types d'advice



- Spring AOP propose 5 types d'advices:
  - **before:** Le **code de l'advice** est exécuté **avant l'exécution de la méthode.** Il n'est pas possible d'empêcher l'invocation de la méthode sauf si une exception est levée dans l'advice.
  - **after returning:** Le **code de l'aspect** est exécuté **après l'exécution de la méthode** à condition qu'aucune exception n'est levée.
  - **after throwing:** Le **code de l'aspect** est exécuté lorsqu'une exception est levée suite à l'invocation de la méthode.
  - **after:** Le code de l'aspect est exécuté après l'exécution de la méthode, même si une exception est levée.
  - **around:** Le code de l'aspect permet de lancer l'exécution de la méthode et ainsi de réaliser des traitements avant et après (peut aussi arrêter la propagation de l'exception).

# After returning



```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

    @AfterReturning(
        pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
        returning="retVal")
    public void doAccessCheck(Object retVal) {
        // ...
    }

}
```

# After throwing



```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

    @AfterThrowing(
        pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
        throwing="ex")
    public void doRecoveryActions(DataAccessException ex) {
        // ...
    }

}
```

# After (Finally)



```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.After;

@Aspect
public class AfterFinallyExample {

    @After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
    public void doReleaseLock() {
        // ...
    }

}
```

# Around



```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class AroundExample {

    @Around("com.xyz.myapp.SystemArchitecture.businessService()")
    public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
        // start stopwatch
        Object retVal = pjp.proceed();
        // stop stopwatch
        return retVal;
    }

}
```

# Pointcut expressions

"execution(Modifiers-pattern? Ret-type-pattern
Declaring-type-pattern? Namepattern(param-
pattern)
Throws-pattern?)"



```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern)
            throws-pattern?)
```

- "?" veut dire optionnel
  - Modifiers-pattern? : public, private ...
  - Ret-type-pattern : le type de retour.
  - Declaring-type-pattern? : nom de la classe y compris le package.
  - Name-pattern : nom de la méthode.
  - Throws-pattern? : l'exception.
  - **".." veut dire, 0 ou plusieurs paramètres**

# Exemples de PointCut expressions



- the execution of any public method:

```
execution(public * *(..))
```

"..." veut dire, 0 ou plusieurs paramètres

- the execution of any method with a name beginning with "set":

```
execution(* set*(..))
```

- the execution of any method defined by the `AccountService` interface:

```
execution(* com.xyz.service.AccountService.*(..))
```

- the execution of any method defined in the service package:

```
execution(* com.xyz.service.*.*(..))
```

- the execution of any method defined in the service package or a sub-package:

```
execution(* com.xyz.service..*.*(..))
```

# AOP par la pratique

# Exemple : Tracer tous les appels SOAP



- Ajouter la dépendance Maven:

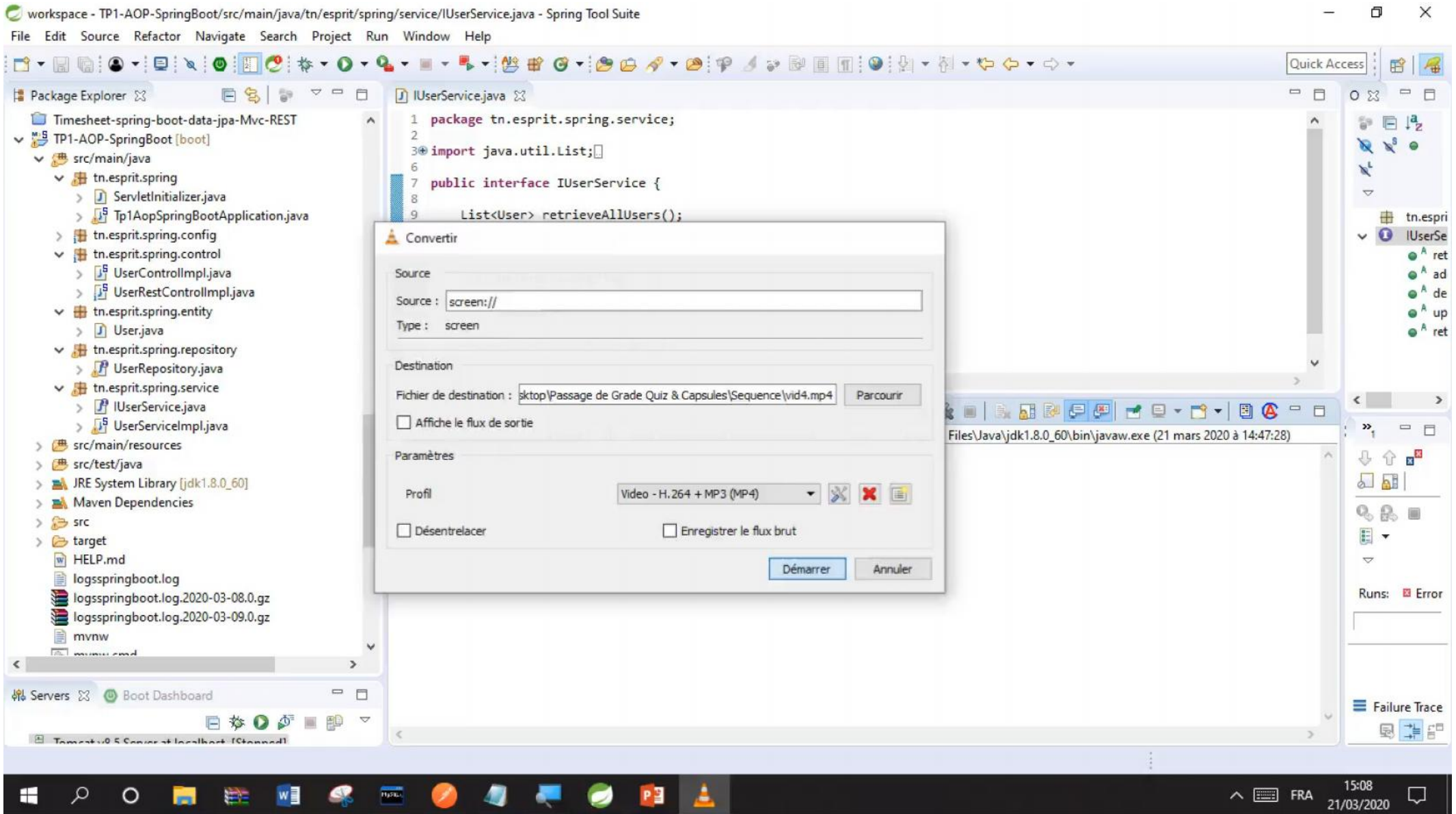
```
<dependency>
<groupId>org.aspectj</groupId>
<artifactId>aspectjweaver</artifactId>
<version>1.9.2</version>
</dependency>
```

- Activer l'AOP dans le projet (exemple: configuration par annotation).

```
@Configuration
@ComponentScan(basePackages =
{"tn.esprit.esponline.control",
"tn.esprit.esponline.service",
"tn.esprit.esponline.dao", "tn.esprit.esponline.config"})
@EnableAspectJAutoProxy
public class BeansConfiguration { }
```

# Exemple : Tracer tous les appels SOAP





# Exemple : Tracer tous les appels SOAP



- Définir l'aspect

# Exemple : Tracer tous les appels SOAP



- Exécution

Spring va créer une classe “proxy” qui implémente **IClientInfoService** et va l’injecter à la place du bean “**clientInfoSOAPServiceInf**”.

```
[04-07-2018 19:27:27.699] [DEBUG] [o.s.aop.framework.JdkDynamicAopProxy.getProxy(118)] - Creating JDK dynamic
proxy: target source is SingletonTargetSource for target object
[tn.esprit.service.ClientInfoSOAPServiceImpl@32f5d7d]
[04-07-2018 19:27:27.751] [INFO ] [tn.esprit.aspect.Tracking.trackSOAPCalls(23)] - One SOAP service is called !
[04-07-2018 19:27:27.752] [INFO ] [tn.esprit.aspect.Tracking.trackSOAPCalls(23)] - One SOAP service is called !
[04-07-2018 19:27:27.752] [INFO ] [tn.esprit.presentation.ClientView.main(43)] - Nom et prenom : Walid <From
SOAP> YAICH <From SOAP>
```

C’est plutôt le proxy qui est injecté

```
@Component("identityControllerBean")
public class IdentityControllerImpl implements IidentityController {

    @Autowired
    @Qualifier("clientInfoSOAPServiceImpl")
    private IClientInfoService clientInfoService;

    @Override
    public String getFullNameByClientId(int clientId){
        //Appeler le webservice SOAP
        return "Nom et prenom : " +
                clientInfoService.getFirstNameByClientId(clientId) + " " +
                clientInfoService.getLastNameByClientId(clientId);
    }
```

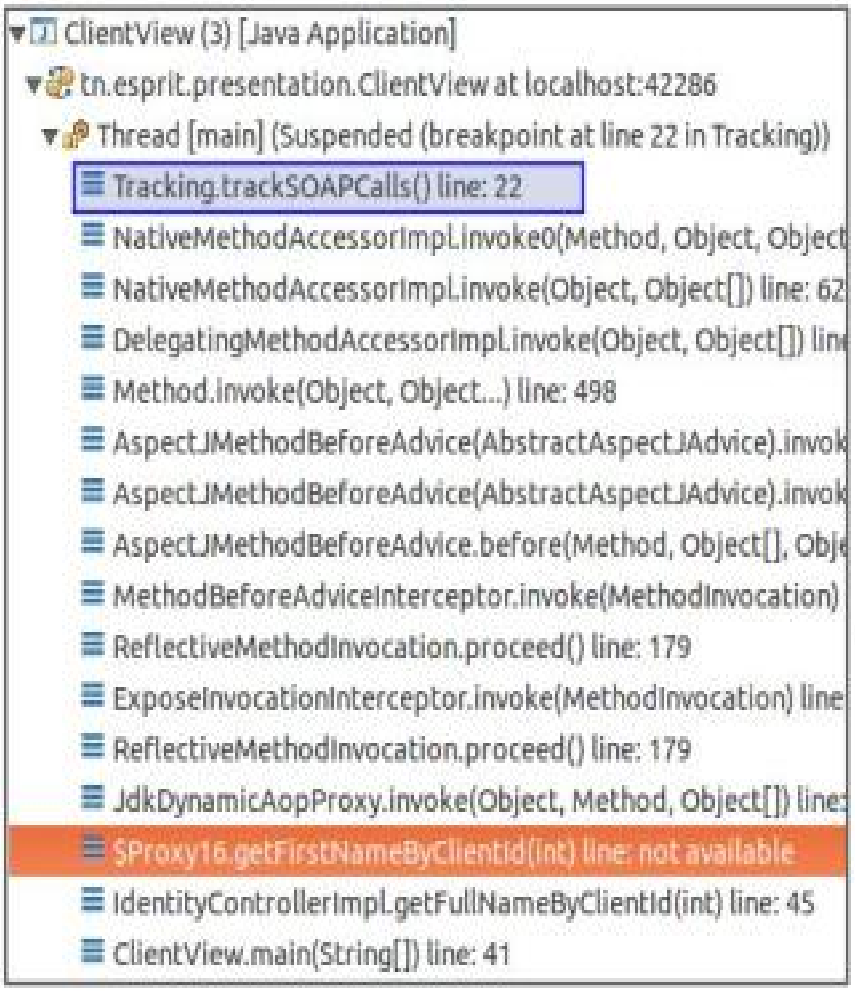
Joinpoint

```
@Component
public class ClientInfoSOAPServiceImpl
        implements IClientInfoService{

    @Override
    public String getFirstNameByClientId(int clientId){
        //Appel a un webservice SOAP
        return "Walid <From SOAP>";
    }

    @Override
    public String getLastNameByClientId(int clientId){
        //Appel a un webservice SOAP
        return "YAICH <From SOAP>";
    }
```

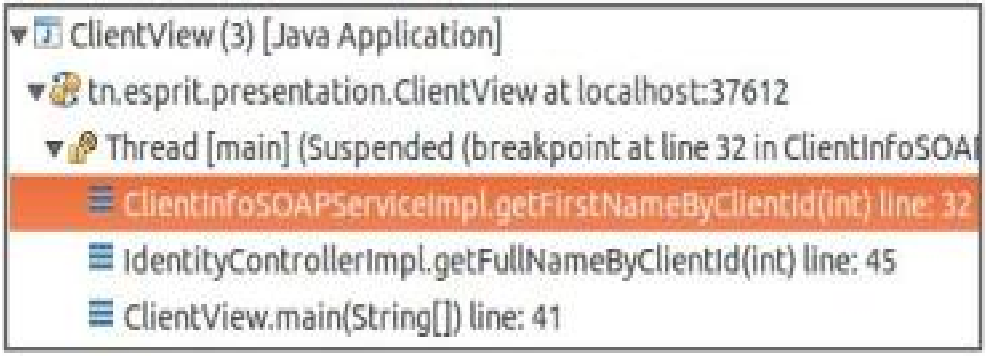
# Exemple : Tracer tous les appels SOAP



Avec AOP




Sans AOP



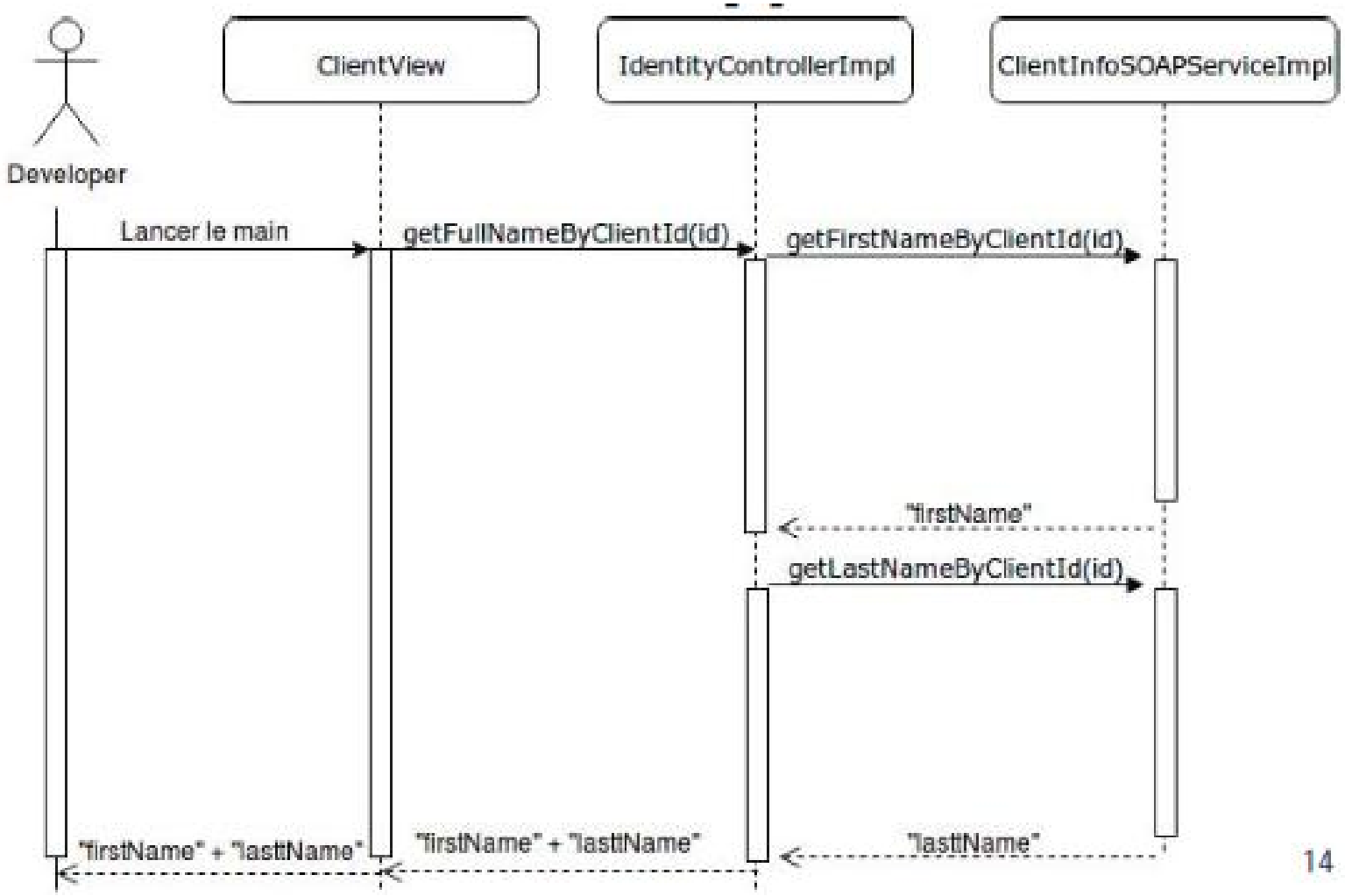

Avec L'aspect qu'on a rajouté, c'est plutôt la méthode **getFirstNameByClientId** de la classe **$Proxy16** qui est appelée en premier lieu !

# Exemple : Tracer tous les appels SOAP



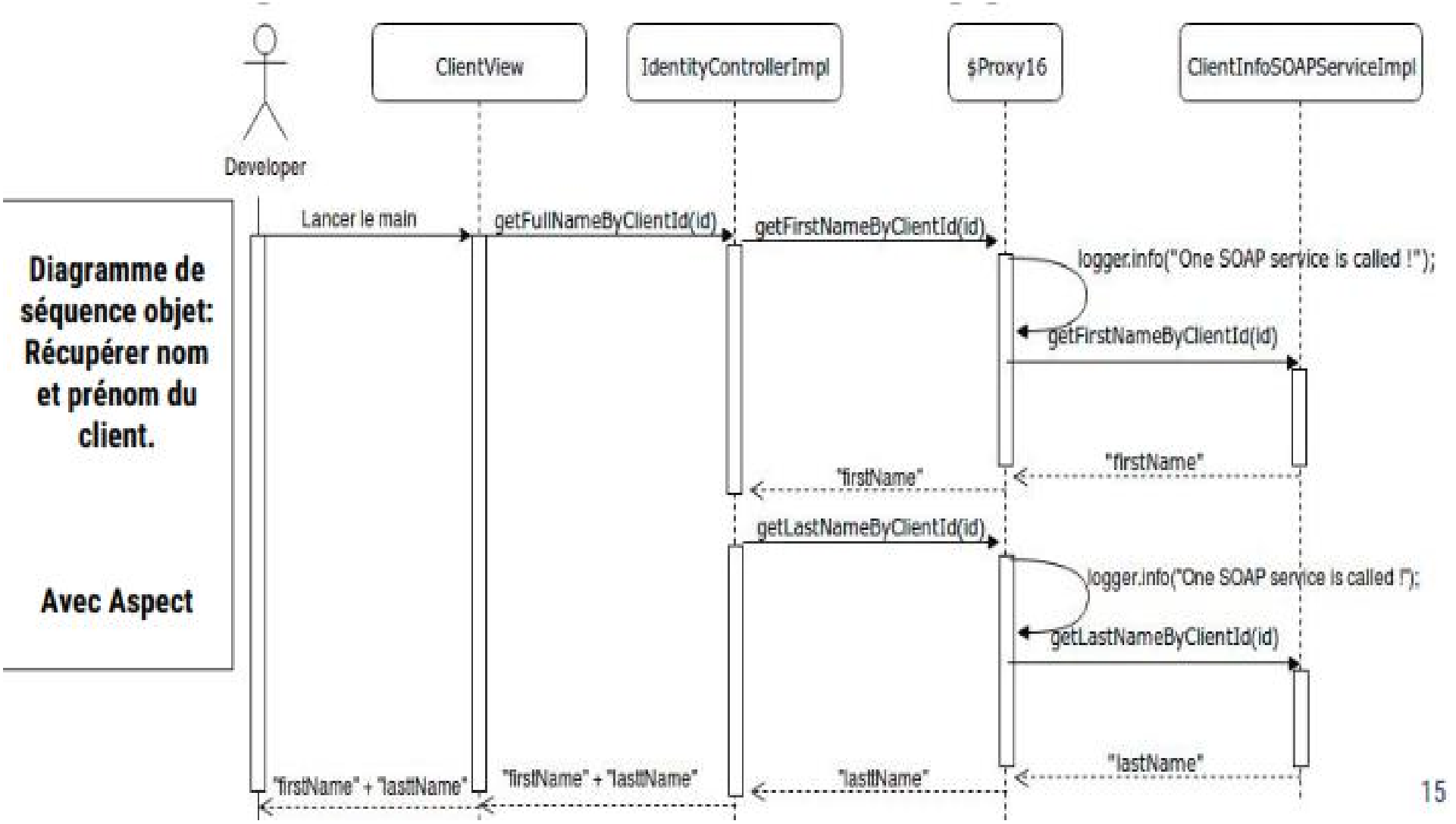
**Diagramme de séquence objet : Récupérer nom et prénom du client.**

**Sans Aspect**



# Exemple : Tracer tous les appels SOAP





# Exemple : Tracer tous les appels SOAP



Jusque là, on a ce message "One SOAP Service is called" avant chaque appel a un service soap.

On souhaite mettre à jour l'aspect avec l'information suivante :
Quel est le service SOAP qui a été appelé ?

```
@Component //Sinon l'aspect ne sera pas détecté
@Aspect
public class Tracking {
    private Logger logger = LoggerFactory.getLogger(getClass());

    @Before("execution(* tn.esprit.service.ClientInfoSOAPServiceImpl.*(..))")
    public void trackSOAPCalls(JoinPoint joinPoint){
        logger.info("This SOAP Service " + joinPoint.getSignature().getName() + " is called !");
    }
}
```

# TP

# Exercice 1 : Advice



- Quelle assertion est correcte :
  - L'AOP injecte des Beans dans les objets Java que nous manipulons
  - L'élément de base de la Programmation Orientée Aspect (AOP) est l'Aspect
  - L'élément de base de la programmation procédurale est l'Objet
  - L'élément de base de la programmation Orientée Objet (OOP) est la Fonction

# Exercice 2 : Advice



- Indiquer quel est **l'Aspect, le JoinPoint et le PointCut, l'Advice et le type d'advice :**
- Attention : l'une de ces notions ci-dessus ne se trouve pas dans le code ci-dessous, laquelle?

```
package tn.esprit.esponline.config;
import ……
@Component
@Aspect
public class LoggingAspect {
private static final Logger logger = Logger.getLogger(LoggingAspect.class);
@Before("execution(* tn.esprit.esponline.service.*.*(..))")
public void logMethodEntry(JoinPoint joinPoint) {
String name = joinPoint.getSignature().getName();
logger.info("In method " + name + " : ");
}
}
```

# Exercice 3 : Advice



- Expliquer les **PointCut suivants :**

```
@Before("execution(* tn.esprit.esponline.service.*.*(..))")

@Before("execution(public * *(..))")

@Before("execution(* set*(..))")

@Before("execution(* tn.esprit.esponline..*.*(..))")
```

# TP2

# TP1 : Logs



- Nous allons créer un projet qui contiendra du code simple.

Par la suite, nous allons enrichir ce projet avec un Aspect (Journalisation ou Logs) :

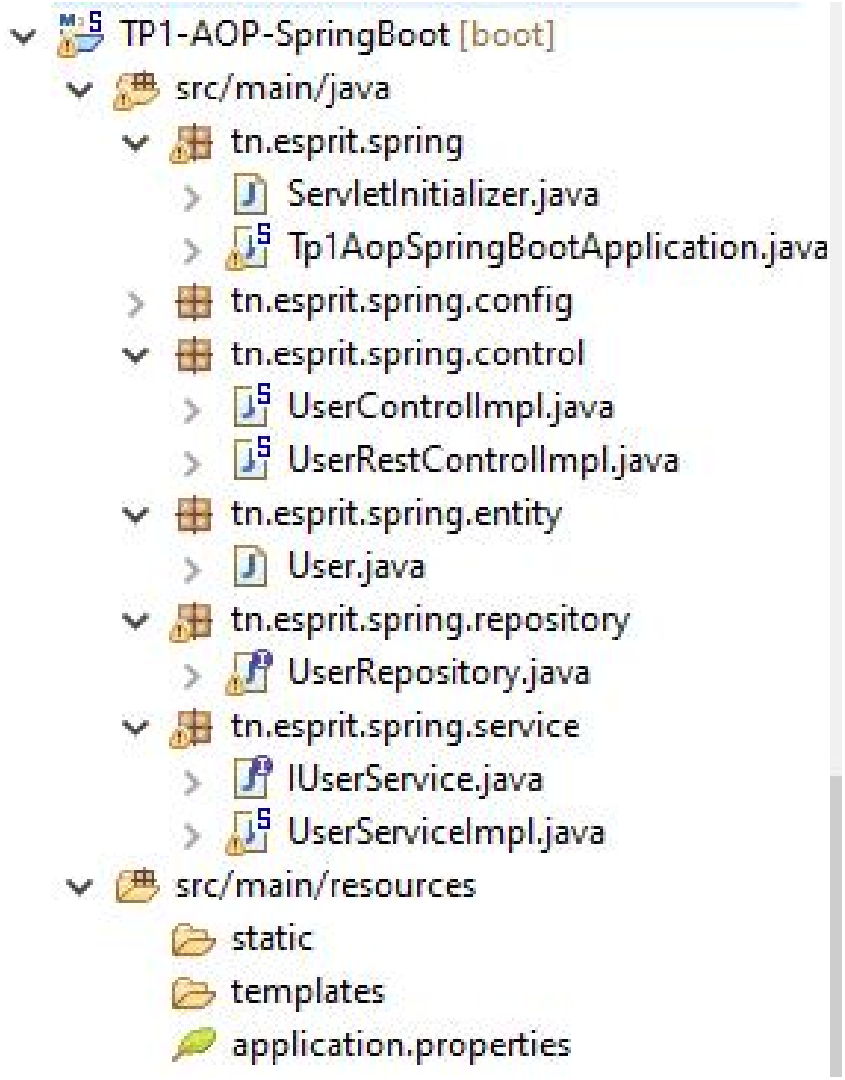
- Créer un projet : **tp1-spring-aop**, en choisissant **«New Spring Stater Project» ou en utilisant un projet Spring Boot déjà existant.**
- Mettre à jour le **pom.xml** avec la dépendance **aspectjweaver.**

```
<dependency>
<groupId>org.aspectj</groupId>
<artifactId>aspectjweaver</artifactId>
<version>1.9.2</version>
</dependency>
```

https://mvnrepository.com/

# TP1 : Logs

```
public interface IUserService {

List<User> retrieveAllUsers();

User addUser(User u);

void deleteUser(String id);

User updateUser(User u);

User retrieveUser(String id);

}
```



# TP1 : Logs



- Bean Service UserServiceImpl .java :
  - Les méthodes seront à créer dans le bean Service :

```
@Service
public class UserServiceImpl implements IUserService {

@Autowired
UserRepository userRepository;
private static final Logger logger = LogManager.getLogger(UserServiceImpl.class);
@Override
public List<User> retrieveAllUsers() {
return (List<User>) userRepository.findAll(); }
```

  - Classe de configuration BeansConfiguration.java :

Ajouter l'annotation **@EnableAspectJAutoProxy sur la classe** Tp1AopSpringBootApplication. Elle permet l'activation de Spring AOP :

```
@SpringBootApplication
@EnableAspectJAutoProxy
public class Tp1AopSpringBootApplication {
public static void main(String[] args) {
SpringApplication.run(Tp1AopSpringBootApplication.class, args);}}
```

# TP1 : Logs



- Ajouter un **Aspect de Journalisation** (logs) et ré-exécuter le test unitaire : Dans le paquet config, ajouter l'aspect :

```
package tn.esprit.spring.config;

@Component
@Aspect
public class LoggingAspect {
private static final Logger logger =
LogManager.getLogger(LoggingAspect.class);
@Before("execution(* tn.esprit.spring.service.UserServiceImpl.*(..))")
public void logMethodEntry(JoinPoint joinPoint) {
String name = joinPoint.getSignature().getName();
logger.debug("In method " + name + " : ");
}
@After("execution(* tn.esprit.spring.service.UserServiceImpl.*(..))")
public void logMethodExit(JoinPoint joinPoint) {
String name = joinPoint.getSignature().getName();
logger.debug("Out of " + name );
}}
```

# TP1 : Logs



- Lancer Postman et exécuter le EndPoint REST suivant:
  **http://localhost:8081/SpringMVC/servlet/retrieve-all-users**
- Exécution:

```
2020-03-20 12:02:20.821[0;39m [32mDEBUG[0;39m [35m5816[0;39m [2m---[0;39m [2m[nio-8081-exec-1][0;39m
[36mtn.esprit.spring.config.LoggingAspect    [0;39m [2m:[0;39m In method retrieveAllUsers :
[2m2020-03-20 12:02:20.982[0;39m [32m INFO[0;39m [35m5816[0;39m [2m---[0;39m [2m[nio-8081-exec-1][0;39m
[36mo.h.h.i.QueryTranslatorFactoryInitiator [0;39m [2m:[0;39m HHH000397: Using ASTQueryTranslatorFactor
Hibernate: select user0_.id as id1_0_, user0_.f_name as f_name2_0_, user0_.l_name as l_name3_0_ from
user user0_
[2m2020-03-20 12:02:21.376[0;39m [32mDEBUG[0;39m [35m5816[0;39m [2m---[0;39m [2m[nio-8081-exec-1][0;39m
[36mtn.esprit.spring.config.LoggingAspect    [0;39m [2m:[0;39m Out of retrieveAllUsers
```

# TP2

# TP2 : Mesure de Performance



- En vous inspirant du TP1, créer dans le même projet un aspect qui permet de calculer et afficher dans les logs, la durée d'exécution de chaque méthode appelée.

```
2020-03-20 12:19:05.859[0;39m [32mDEBUG[0;39m [35m2956[0;39m [2m---[0;39m [2m[nio-8081-exec-2][0;39m
[36mtn.esprit.spring.config.LoggingAspect    [0;39m [2m:[0;39m In method retrieveAllUsers :
[2m2020-03-20 12:19:06.771[0;39m [32m INFO[0;39m [35m2956[0;39m [2m---[0;39m [2m[nio-8081-exec-
2][0;39m [36mo.h.h.i.QueryTranslatorFactoryInitiator [0;39m [2m:[0;39m HHH000397: Using
ASTQueryTranslatorFactory
Hibernate: select user0_.id as id1_0_, user0_.f_name as f_name2_0_, user0_.l_name as l_name3_0_ from
user user0_
[2m2020-03-20 12:19:07.378[0;39m [32mDEBUG[0;39m [35m2956[0;39m [2m---[0;39m [2m[nio-8081-exec-
2][0;39m [36mt.e.spring.config.PerformanceAspect     [0;39m [2m:[0;39m Method execution time: 1517
milliseconds.[tn.esprit.spring.entity.User@6b57c675, tn.esprit.spring.entity.User@15d0846e]
[2m2020-03-20 12:19:07.378[0;39m [32mDEBUG[0;39m [35m2956[0;39m [2m---[0;39m [2m[nio-8081-exec-
2][0;39m [36mtn.esprit.spring.config.LoggingAspect    [0;39m [2m:[0;39m Out of retrieveAllUsers
```

# TP3

# TP3: Mesure de Performance



- Reprendre l'exercice **DependencyInjection_TODO (configuration par annotation).**
- Tracer les **temps de réponse** des **méthodes qui commencent par get** et qui se trouvent **sous le package tn.esprit.controller**
- Il faut créer un autre aspect qui s'appelle Performance.

# Références



- J'ai souvent eu recours aux supports disponibles sur le WEB et notamment au cours des personnes suivantes :
  - **Documentation officielle de Spring** : http://spring.io
  - **Documentation officielle de Spring AOP :** https://docs.spring.io/spring/docs/2.0.x/reference/aop.html
  - **Documentation officielle de JavaEE8** https://www.oracle.com/java/technologies/java-ee-glance.html
  - **Cours Mourad Hassini,** assistant technologue à ESPRIT
  - **Cours Walid Yaich,** enseignant vacataire à ESPRIT

# SPRING AOP
# (ASPECT ORIENTED PROGRAMMING)

2019-2020

ESPRIT thouraya.louati@esprit.tn UP JavaEE / .NET (Bureau E204)