HENALLUX
Année académique 2015-2016

Précisions sur le cours de théorie

Steven VON HAUS

Transaction

Première solution

```
public void transactionMethod(Book book1, Book book2)
{   Session session = sessionFactory.getCurrentSession();
    session.beginTransaction();
    session.save(providerConverter.bookModelToBookEntity(book1));
    session.update(providerConverter.bookModelToBookEntity(book2));
    session.getTransaction().commit();
}
```

Solution retenue

```
@EnableTransactionManagement
public class Application {
```

```
@Override
@Transactional
public void updatePassword(String login, String newPassword) {
    UserDb user = userRepository.findByUsername(login);
    user.setPassword(newPassword);
}
```

Transaction

Exemple 1

```
try {
    userDetailsService.createUser(true);
} catch (Exception e) {
    LOG.info("expected exception");
}
```

Résultat ?

a) Les entity0 et entity1 sont persistés car l'exception est correctement catchée

b) Aucun entity n'est persisté

c) Le programme ne compile pas

d) Le programme plante à l'exécution

```
@Override
@Transactional(TxType.REQUIRED)
public void createUser(boolean throwException) {
    UserDb entity0 = createUserDb("test1", "test2", true, "USER", true, true, true);
    UserDb entity1 = createUserDb("test2", "test2", true, "USER", true, true, true);
    UserDb entity2 = createUserDb("test3", "test2", true, "USER", true, true, true);
    UserDb entity3 = createUserDb("test4", "test2", true, "USER", true, true, true);
    UserDb entity4 = createUserDb("test1", "update", true, "USER", true, true, true);

    userRepository.save(entity0);
    userRepository.save(entity1);

    if(throwException) {
        throw new RuntimeException();
    }

    userRepository.save(entity2);
    userRepository.save(entity3);
    userRepository.save(entity4);

    entity0.setPassword("password changed");
}
```

Transaction

Exemple 2

```
userDetailsService.createUser(false);
test = userDetailsService.loadUserByUsername("test1");
LOG.info(test.getPassword());
```

```
@Override
@Transactional(TxType.REQUIRED)
public void createUser(boolean throwException) {
    UserDb entity0 = createUserDb("test1", "test2", true, "USER", true, true, true);
    UserDb entity1 = createUserDb("test2", "test2", true, "USER", true, true, true);
    UserDb entity2 = createUserDb("test3", "test2", true, "USER", true, true, true);
    UserDb entity3 = createUserDb("test4", "test2", true, "USER", true, true, true);
    UserDb entity4 = createUserDb("test1", "update", true, "USER", true, true, true);

    userRepository.save(entity0);
    userRepository.save(entity1);

    if(throwException) {
        throw new RuntimeException();
    }

    userRepository.save(entity2);
    userRepository.save(entity3);
    userRepository.save(entity4);

    entity0.setPassword("password changed");
}
```

Résultat ?

a) « test2 » est affiché

b) « password changed » est affiché

c) NullPointerException sur test.getPassword()

Transaction

Exemple 2

```
userDetailsService.createUser(false);
test = userDetailsService.loadUserByUsername("test1");
LOG.info(test.getPassword());
```

Résultat ?

a) « test2 » est affiché

b) « password changed » est affiché

c) NullPointerException sur test.getPassword()

```
@Override
@Transactional(TxType.REQUIRED)
public void createUser(boolean throwException) {
    UserDb entity0 = createUserDb("test1", "test2", true, "USER", true, true, true);
    UserDb entity1 = createUserDb("test2", "test2", true, "USER", true, true, true);
    UserDb entity2 = createUserDb("test3", "test2", true, "USER", true, true, true);
    UserDb entity3 = createUserDb("test4", "test2", true, "USER", true, true, true);
    UserDb entity4 = createUserDb("test1", "update", true, "USER", true, true, true);

    entity0 = userRepository.save(entity0);
    entity1 = userRepository.save(entity1);

    if(throwException) {
        throw new RuntimeException();
    }

    entity2 = userRepository.save(entity2);
    entity3 = userRepository.save(entity3);
    entity4 = userRepository.save(entity4);

    entity0.setPassword("password changed");
}
```

Transaction

Première solution

- Les transactions utilisent la notion de Proxy.
- Un appel interne provenant d'une méthode non-transactionnelle ne fonctionne pas !

```java
@Service
public class UserDetailsServiceImpl implements IUserDetailsService {

    @Autowired
    private UserRepository userRepository;

    public void initializeDb() { createUser(false); // will throw an Exception ! }

    @Override
    @Transactional(TxType.REQUIRED)
    public void createUser(boolean throwException) {
        UserDb entity0 = createUserDb("test1", "test2", true, "USER", true, true, true);
        userRepository.save(entity0);
    }
```

Design Pattern Proxy

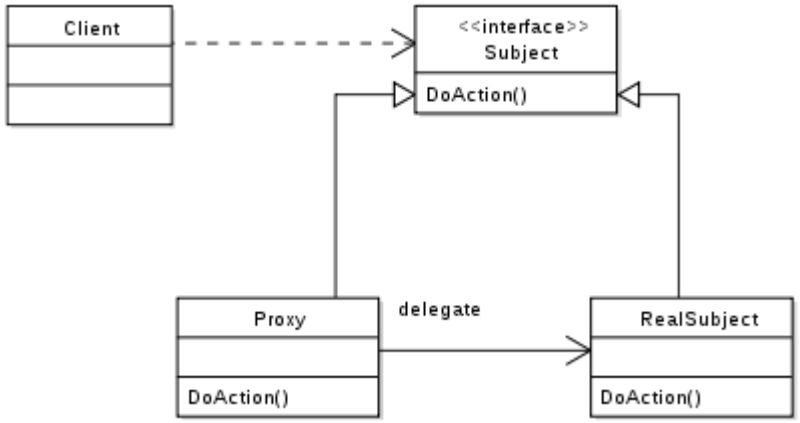
Objectif

- Désir de garder une grande cohésion au niveau de la classe. Une classe est responsable d'un unique aspect (ex: le business)
  - ❑ Quid si on veut ajouter des transactions ?
  - ❑ Du logging ?
  - ❑ Des transactions ?
  - ❑ Des indicateurs de performance ?

Design Pattern Proxy

Fonctionnement

- Le proxy implémente la même interface que la classe dont il doit étendre le comportement.
- Il contient une variable d'instance du type de la classe qu'il va appeler.
- Pour chaque méthode implémentée, le proxy appelle cette même méthode à l'aide de sa variable d'instance.
- => Le proxy peut réaliser des actions avant et après.



https://upload.wikimedia.org/wikipedia/commons/thumb/7/75/Proxy_pattern_diagram.svg/400px-Proxy_pattern_diagram.svg.png

Design Pattern Proxy

Exemple

- Le proxy pourrait ressembler à la figure suivante.
- Attention il s'agit d'un exemple, ce n'est pas du tout représentatif du proxy créé par Spring.

```
@Component
public class UserDetailsServiceProxy implements IUserDetailsService{

    @Autowired
    private SessionFactory sessionFactory;

    @Autowired
    UserDetailsServiceImpl userDetailsService;

    @Override
    public void createUser(boolean throwException) {
        Session session = sessionFactory.getCurrentSession();
        session.beginTransaction();
        userDetailsService.createUser(throwException);
        session.getTransaction().commit();
    }
```

Java Persistence API with Spring Data

```java
public interface UserRepository extends CrudRepository<User, Integer> {

    String USER_BY_GROUP = "select distinct U from User as U "
            + "inner join U.pKUser2Group.group as G "
            + "where G.name = :groupName";

    @Query(USER_BY_GROUP)
    List<User> findByGroupName(@Param("groupName") String groupName);
}
```

```java
@Embeddable
public class PKUser2Group {

    @Column(name = "USER_LOGIN", nullable = false)
    private String login;

    @ManyToOne(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name = "ID_GROUP")
    private Group group;
```

```java
@EmbeddedId
private PKUser2Group pKUser2Group;
```

Spring context

Fonctionnement du @ComponentScan

- @ComponentScan -> Indique à Spring où chercher les **beans**
- Spring parcourt toutes les classes à la recherche des **annotations** @Service, @Component, @Repository, @Controller, @RestController, etc…
- Spring crée une **instance** (Singleton Design Pattern) dans le contexte pour chacune de ces classes

Spring context

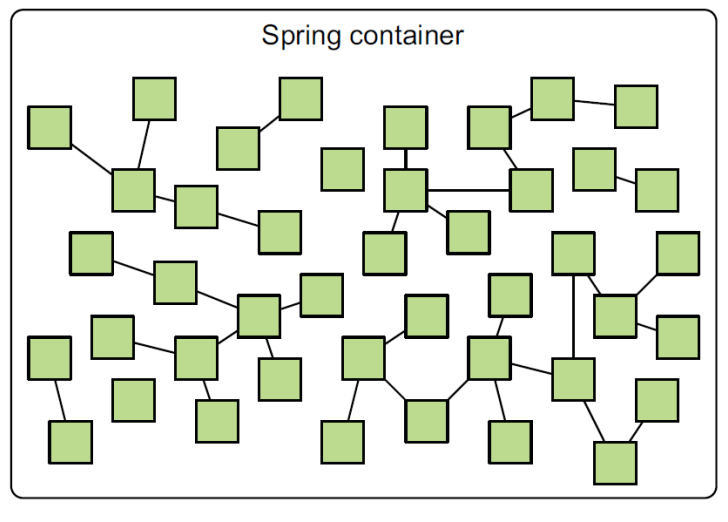
Fonctionnement du @ComponentScan

- @EnableAutoConfiguration-> Indique à Spring qu'il doit chercher les **classes de configuration**
- Spring parcourt toutes les classes à la recherche de l'annotation **@Configuration**
- Pour chacune de ses classes, Spring détecte chaque méthode annotée **@Bean**
- Spring crée une instance dans le contexte pour chacune de ces méthodes

Spring context

@Autowired

- Besoin de lier (wire) tous ces composants



- => @Autowired

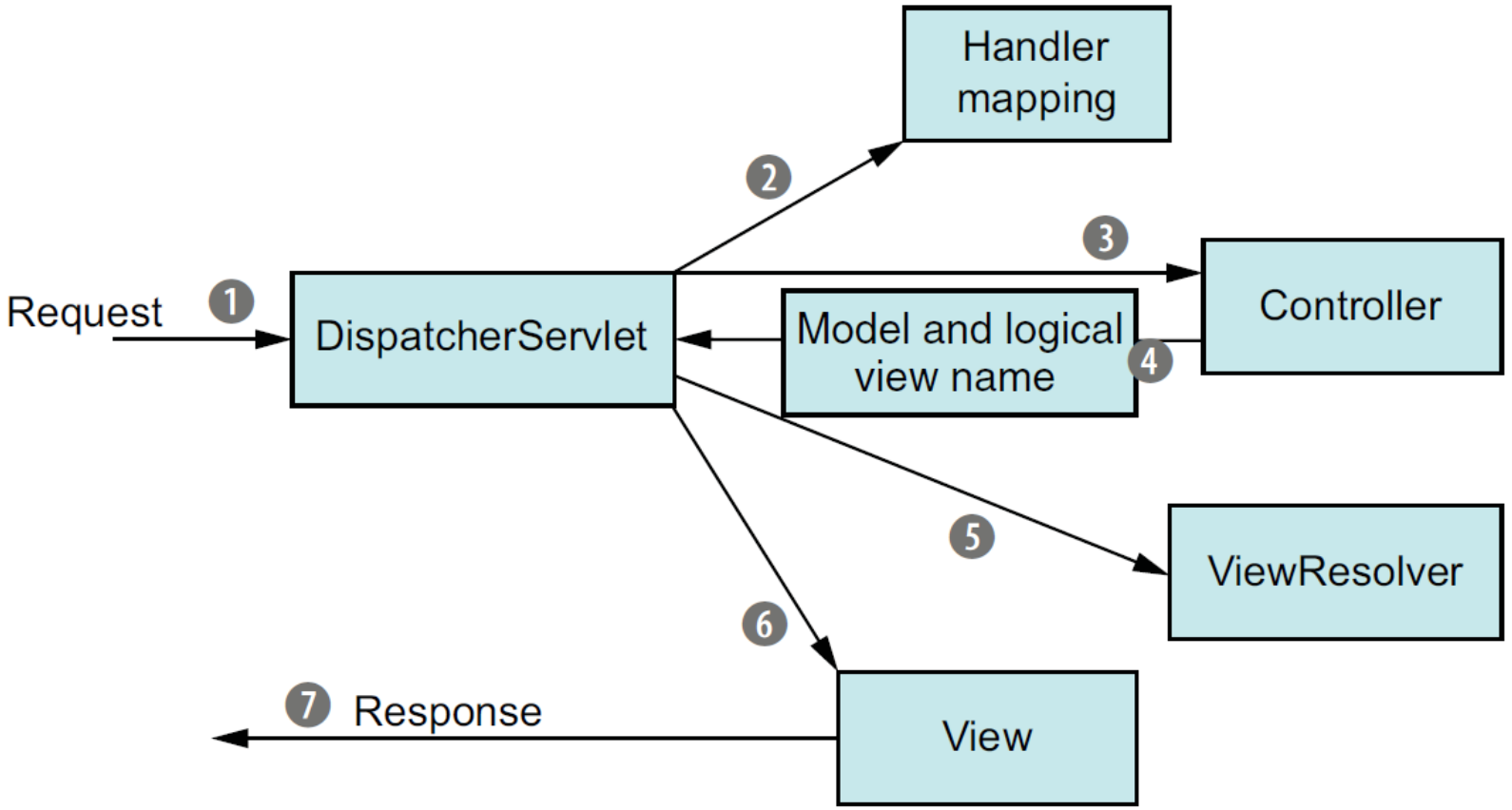
Spring context

@Autowired

- Variables d'instance
- Constructeur
- Setter

- Même effet au final mais…
  - ❑ Préférable au niveau du constructeur
  - ❑ Complexité de la classe visible
  - ❑ **Testability**
  - ❑ Readability
  - ❑ Si un jour la classe n'est plus dans le contexte Spring => Rien à changer

DispatcherServlet



Dispatcher Servlet

Fonctionnement

- **@ComponentScan** pour identifier les controllers (cf: slides précédents)
- **@RequestMapping** pour identifier les méthodes traitant les requêtes
- Découple la logique du controller de la vue

Dispatcher Servlet

Fonctionnement

1) Le DispatcherServlet est le composant au coeur de Spring MVC. Il est l'implementation direct du design pattern "Front controller":

Une unique servlet endosse la responsabilité d'orchestrer toutes requêtes entrantes au sein d'une application. Cet unique point d'entrée délègue l'ensemble du traitement aux différents composants de l'application.

Dispatcher Servlet

Fonctionnement

2) Un ou plusieurs “handler mapping” sont consultés par le DispatcherServlet afin d’identifier le bon controller ainsi que la bonne méthode à l’aide des annotations @Controller et @RequestMapping

3) Le dispatcher servlet appelle la méthode du controller qui a été désigné en passant en argument les différents de la requête (param, model, headers, etc)

Dispatcher Servlet

Fonctionnement

4) Le controller intéragit avec les différents composants de l'application afin de fournir au DispatcherServlet le Model complété ainsi que le nom de la vue qui sera retournée

5) Le DispatcherServlet fournit le nom de la vue ainsi que le Model au View Resolver (ex: TilesViewResolver) qui va se charger de construire la vue à renvoyer (JSP).

Dispatcher Servlet

Fonctionnement

6) La vue utilise le Model afin de produire la réponse renvoyée au client (html, headers,etc)

Conclusion: Beaucoup d'étapes sont réalisés mais vous n'avez pas à vous en préoccuper lors de l'utilisation de Spring MVC.