

L'injection de dépendances

Jean-Baptiste Nizet

Projet Github

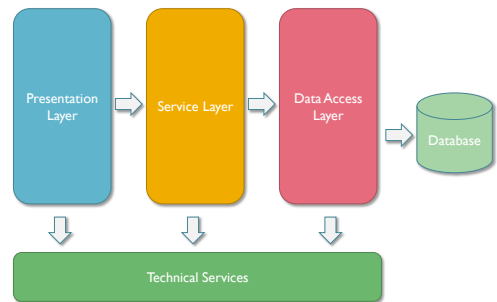
git clone https://github.com/Ninja-Squad/tpdi.git

Qui suis-je

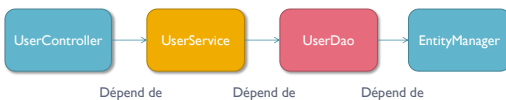
- jb@ninja-squad.com
- @jbnizet
- Co-fondateur de Ninja Squad
- Développeur Java depuis 1997
- Très actif sur StackOverflow.com




Architecture typique



Dépendance



new



```

public class UserController {
    private UserService userService = new UserService();


    public ModelAndView showUsers() {
        List<User> users = userService.findAllUsers();
        // ...
    }
}
  
```

Problèmes de new

Couplage
Accès à distance
Transactions
Sécurité

TESTS

Factory



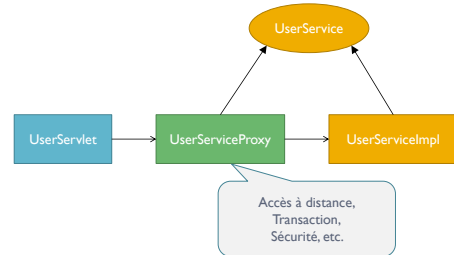
```
public class UserController {
    private UserService userService = UserServiceFactory.createUserService();

    public ModelAndView showUsers() {
        List<User> users = userService.findAllUsers();
        // ...
    }
}
```

Avantages de Factory

- Moins de couplage: on dépend de l'interface et pas de l'implémentation
- La factory peut retourner un proxy (remoting, transactions, sécurité)

Proxy




Inconvénients de Factory

Ecrire la factory
Ecrire le proxy
Boilerplate code

TESTS

Service locator



```
public class UserController {
    private UserService userService;

    public UserController() {
        try {
            InitialContext ctx = new InitialContext();
            UserServiceHome home =
                (UserServiceHome) ctx.lookup("java:/comp/env/userService");
            this.userService = home.create();
        } catch (NamingException e) {
            throw new RuntimeException("Impossible to locate user service", e);
        } catch (CreateException e) {
            throw new RuntimeException("Impossible to create user service", e);
        }
    }

    public ModelAndView showUsers() {
        List<User> users = userService.findAllUsers();
        // ...
    }
}
```

Problèmes de Service Locator

Boilerplate code

~~XML~~ XMHELL

Dépendances
peu claires

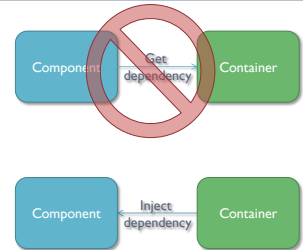
TESTS

Injection de dépendances : principe

IOC

Hollywood
principe

Don't call me,
I'll call you



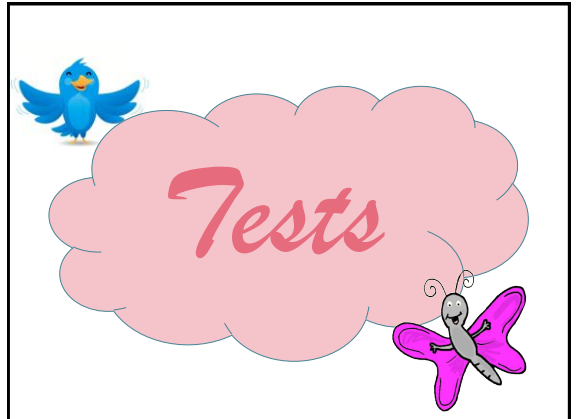
Injection de dépendances

```

public class UserController {
    private UserService userService;

    @Inject
    public UserController(UserService userService) {
        this.userService = userService;
    }

    public ModelAndView showUsers() {
        List<User> users = userService.findAllUsers();
        // ...
    }
}
  
```



Test (JUnit, Mockito)

```

public class UserControllerTest {
    private UserService mockUserService;
    private UserController userController;

    @Before
    public void prepare() {
        mockUserService = mock(UserService.class);
        userController = new UserController(mockUserService);
    }

    @Test
    public void testShowUsers() {
        List<User> users = new ArrayList<User>();
        when(mockUserService.findAllUsers()).thenReturn(users);

        ModelAndView result = userController.showUsers();

        assertEquals("users", result.getViewName());
        assertEquals(users, result.getModel().get("users"));
    }
}
  
```

Field injection

```

public class UserServiceImpl implements UserService {
    @Inject
    private UserDao userDao;

    @Override
    public List<User> findAllUsers() {
        return userDao.findAllUsers();
    }
}
  
```

Constructor Injection

```
public class UserServiceImpl implements UserService {
    private UserDao userDao;

    @Inject
    public UserServiceImpl(UserDao userDao) {
        this.userDao = userDao;
    }

    @Override
    public List<User> findAllUsers() {
        return userDao.findAllUsers();
    }
}
```

Method / Setter Injection

```
public class UserServiceImpl implements UserService {
    private UserDao userDao;

    @Inject
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }

    @Override
    public List<User> findAllUsers() {
        return userDao.findAllUsers();
    }
}
```

Field Injection – Avantages / Inconvénients

- Compact
- Aucune action possible à l'injection
- La classe n'est pas testable facilement

Constructor Injection – Avantages / Inconvénients

- La plus logique
- Les champs peuvent être final
- Les dépendances sont claires
- Pas d'injection optionnelle
- Impossible dans certains cas (servlets par exemple)
- Les sous-classes doivent connaître les dépendances de leurs super-classes
- Moins pratique pour les tests qui n'ont besoin que d'une dépendance

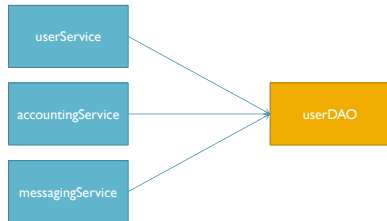
Setter Injection – Avantages / Inconvénients

- La seule qui fonctionne dans certains cas
- Facilite la configuration par XML
- L'objet passe par un état invalide transitoire

Configuration

- XML (Spring)
- Code et annotations (Spring, Guice, CDI)
- Classpath scanning (CDI)

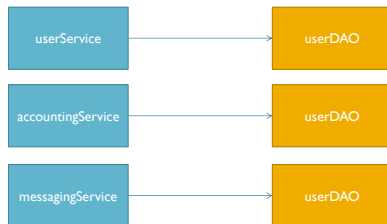
Scopes : Singleton



Singleton

- Attention: doit être thread-safe
- Nécessaire pour maintenir un état global en mémoire
- Spring : scope par défaut

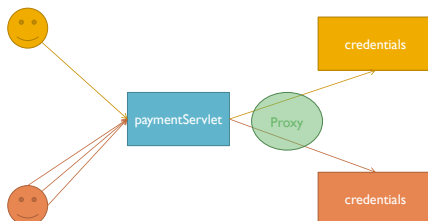
Scopes : Prototype



Prototype

- Pas de problème de concurrence
- Nécessaire pour maintenir un état contextuel

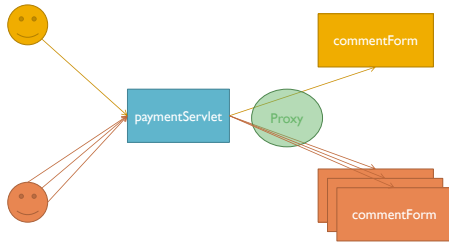
Scopes : Session



Session

- Doit être thread-safe
- Utile pour sauvegarder des informations valables pour la durée de vie de la session (nom, prénom, droits, etc.)

Scopes : Request



Les transactions

- Méthode transactionnelle avec JPA :

```
public void transfer(BigDecimal amount, Long fromAccountId, Long toAccountId) {
    em = emFactory.createEntityManager();
    em.getTransaction().begin();
    try {
        // do the transfer
        em.getTransaction().commit();
    }
    catch (Exception e) {
        em.getTransaction().rollback();
    }
    finally {
        em.close();
    }
}
```

Problèmes de cette approche

- Boilerplate code
- Risques d'erreurs
- Comment faire si une méthode transactionnelle appelle une autre méthode transactionnelle ?
- Code fonctionnel lié à JPA. Comment faire si une méthode doit utiliser JDBC et JPA ?

Solution : les proxies

```
@Transactional
public void transfer(BigDecimal amount, Long fromAccountId, Long toAccountId) {
    // do the transfer
}
```

- Le conteneur injecte un proxy
- Ce proxy
 - démarre la transaction
 - appelle la vraie méthode tranfer()
 - committe / rollbacke la transaction

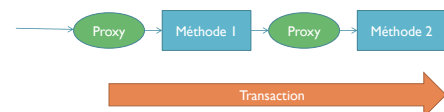
Propagation de la transaction

- Si une méthode transactionnelle appelle une autre méthode transactionnelle, le proxy détecte qu'une transaction est déjà en cours
- Plusieurs modes de propagation possibles, dont
 - REQUIRED
 - REQUIRES_NEW

REQUIRED

```
@Transactional(propagation = Propagation.REQUIRED)
public void transfer(BigDecimal amount, Long fromAccountId, Long toAccountId) {
    // do the transfer
}
```

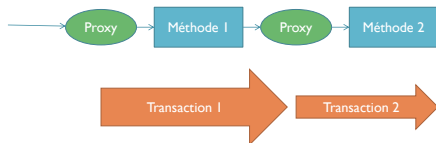
- La deuxième méthode est exécutée dans la même transaction que la première



REQUIRES_NEW

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void transfer(BigDecimal amount, Long fromAccountId, Long toAccountId) {
    // do the transfer
}
```

- La deuxième méthode est exécutée dans une nouvelle transaction
- La première transaction est suspendue pendant la deuxième, puis reprend



Les frameworks

- Il existe plusieurs frameworks d'injection de dépendances, et notamment
 - Spring : l'un des premiers, sans doute le plus utilisé
 - Guice : a introduit l'utilisation d'annotations et de code Java plutôt que le XML utilisé par Spring. Spring a depuis rattrapé son retard
 - CDI : le standard dans JEE6, mais est arrivé très tard

Spring

- C'est celui que nous allons utiliser
- Version 3.2
- Nous allons tout configurer avec du code Java et des annotations

Beans et ApplicationContext

- Les composants injectables par Spring sont appelés des beans.
- Le composant Spring qui gère l'injection est appelé l'ApplicationContext
- Il faut configurer l'ApplicationContext pour lui dire quels beans existent

Bootstrapping

- Presque tout se passe au démarrage de l'application
- On fournit des beans, annotés
- On dit à l'ApplicationContext où les trouver
- Spring établit le graphe des dépendances des beans, instancie les beans, crée les proxies nécessaires, et injecte les beans les uns dans les autres.
- On demande un bean à l'ApplicationContext, et on l'utilise

A quoi ressemble un bean?

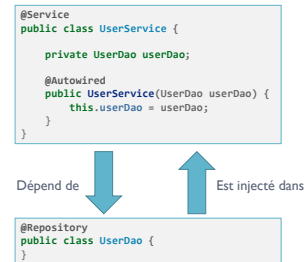
```
@Component
public class MyFirstBean {
    // constructeur sans argument!
}
```

- Spring doit instancier votre bean. Il lui faut donc un constructeur sans argument.

Stereotypes

- Autres annotations possibles :
 - @Repository : pour les DAOs, dont le rôle est de gérer la persistance
 - @Service : pour les services, souvent transactionnels, qui utilisent les DAOs
 - @Controller : pour les controllers de Spring MVC, qui gère les requêtes dans une application web

Dépendances



Configurer Spring

- Créer une classe annotée avec @Configuration
- Le plus simple : laisser Spring trouver les beans annotés dans le classpath

```

package com.ninja_squad.springdemo;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.ninja_squad.springdemo")
public class AppConfig {
}

```

Configurer le contexte

```

public static void main(String[] args) {
    ApplicationContext ctx =
        new AnnotationConfigApplicationContext(AppConfig.class);

    MyFirstBean myFirstBean = ctx.getBean(MyFirstBean.class);
    myFirstBean.doStuff();
}

```

Configuration des beans

- On peut vouloir créer un bean à partir d'une classe externe au projet, non annotée.
 - Exemples : DataSource, Executor, etc.
- On peut aussi vouloir configurer un bean
- Créer une méthode de fabrique dans la classe de configuration
- Utiliser l'annotation @Bean sur cette méthode de fabrique

Exemple de bean externe

```

@Configuration
@ComponentScan("com.ninja_squad.springdemo")
public class AppConfig {
    @Bean
    public ExecutorService batchExecutorService() {
        return Executors.newFixedThreadPool(4);
    }
}

```


Beans Spring

- Spring fournit un grand nombre de classes de beans rendant divers services techniques :
 - `TransactionManager` pour gérer les transactions et leur propagation
 - `DataSource` pour accéder à une base de données
 - `LocalEntityManagerFactoryBean` pour utiliser JPA
 - Etc.

Configuration des beans Spring

```
public class AppConfig {  
    @Bean  
    public LocalContainerEntityManagerFactoryBean emf() {  
        LocalContainerEntityManagerFactoryBean result =  
            new LocalContainerEntityManagerFactoryBean();  
        result.setPersistenceUnitName("TP_JPA");  
        result.setDataSource(dataSource());  
        result.setPersistenceProviderClass(HibernatePersistence.class);  
        return result;  
    }  
  
    @Bean  
    public DataSource dataSource() {  
        return new DriverManagerDataSource("jdbc:hsqldb:hsqldb://localhost/TP_JPA",  
            "sa",  
            "");  
    }  
  
    @Bean  
    public PlatformTransactionManager txManager() {  
        return new DataSourceTransactionManager(dataSource());  
    }  
}
```