Injetando Repositorios (Repository) em entidades de maneira transparente com Spring

Tenho conversado com alguns amigos sobre os projetos que tenho participado no emprego novo e uma das dúvidas que surge é como utilizar repositórios em entidades pra tornar o domínio um pouco mais rico.

Obviamente, este POST não tem o intuito de levantar nenhuma discussão sobre como você deve tratar sua camada de persistência e o seu domínio, mas sim, mostrar como é possível injetar repositórios dentro de entidades de maneira transparente usando Spring.

Para saber mais a respeito do conceito por trás desta solução, leia este POST (http://blog.caelum.com.br/2007/06/09/repository-seu-modelo-mais-orientado-a-objeto/).

O problema

Ao recuperar uma entidade (user, por exemplo) de um repositório, as dependências desta entidade devem ser injetadas manualmente dentro do método find.

Exemplo:

```
public User findById(Long id) {
   User user = entityManager.find(User.class, id);
   user.setUserRepository(this);
   user.setRoleRepository(...);
   user.setXXXRepository(...);
}
```

Você talvez queira injetar só um repositório, mas talvez não. Imagine, se ao invés de todos estes setters eu apenas usasse a annotation @ResolveDependencies no método e todas as dependências (no caso, repositórios) fossem injetados automaticamente na entidade. Um outro problema é que a cada novo repositório mais um set você tem que efetuar no método find e isso daria um pouco mais de trabalho. Além disso, muitas vezes seus repositórios já estão sendo gerenciados pelo container de injeção de dependências e você não gostaria de criar uma nova instância a cada find que efetuar.

A solução

A classe ApplicationContext do Spring possui recursos para resolver as dependências de um objeto não gerenciado pelo Spring. Desta forma, se na minha entidade User eu possuir um atributo chamado UserRepository anotado pela annotation @Autowired (Spring) ou @Inject (JSR-330), através da chamada do método ctx.getAutowireCapableBeanFactory().autowireBean(user), todas as dependências que o container for capaz, ele injetará. Onde, ctx é uma instância de ApplicationContext, user é uma instância de User e UserRepository é uma instância de UserDAO gerenciada pelo Spring.

A seguir, as classes citadas:

UserRepository.java

```
public interface UserRepository {
```

```
	public User findById(Long id);
	public void save(User user);

}
```

UserDAO.java

```
@Named("userRepository")
public class UserDAO implements UserRepository {

	@Override
	@ResolveDependencies
	public User findById(Long id) {
		return new User(id);
	}

	@Override
	public void save(User user) {
		System.out.println("salvando user id=[" + user.getId() + "]");
	}

}
```

A classe UserDAO, foi anotada pela annotation @Named que faz parte da JSR-330 (Dependency Injection) suportada pelo Spring, o parâmetro passado (userRepository) é a String que identifica o DAO dentro do container de injeção de dependência (no caso, Spring). Toda classe anotada por esta anotação, será gerenciada pelo Spring.

O método save, simplemente imprime uma mensagem com o id do user.

O método findById, foi anotado pela a annotation @ResolveDependencies, que utilizaremos para indicar que a entidade retornada por este método deve ter todas as suas dependências resolvidas (injetadas).

A mágica

Como foi dito anteriormente, o Spring tem condições de resolver dependências de objetos não gerenciados pelo mesmo, desta forma, utilizaremos este recurso para o exemplo acima:

SpringUtils.java

```
public abstract class SpringUtils {

	public static void autowire(Object obj, ApplicationContext ctx) {
		ctx.getAutowireCapableBeanFactory().autowireBean(obj);
	}

}
```

Esta classe abstrata possui um método chamado autowire que, simplesmente, injeta as dependências (contidas no container, no caso, ApplicationContext) em um dado objeto (obj).

Para juntarmos tudo, utilizaremos AOP para interceptarmos os métodos anotados por @ResolveDependencies e, através da classe SpringUtils, resolvermos as dependências do objeto de retorno, conforme demostra a classe a seguir:

DetachedSpringBeanDependencyResolver.java

```
@Aspect
@Named
public class DetachedSpringBeanDependencyResolver {

	@Inject
	@Named("applicationContext")
	private ApplicationContext applicationContext;

	@AfterReturning(value="@annotation(br.com.submundojava.ResolveDependencies)",
 returning="retVal")
	public void resolve(Object retVal) throws Throwable {

		if(retVal != null)
			SpringUtils.autowire(retVal, applicationContext);

	}

}
```

Aqui complica um pouco, mas é bem simples:

@Aspect, sucintamente informa que esta classe utilizará AOP.
@Named, um objeto gerenciado pelo Spring.
@Inject + @Named("applicationContext"), como esta classe também será gerenciada pelo Spring e no próprio Spring existe uma instância de ApplicationContext (obtida pelo nome applicationContext) podemos então injetar este contexto dentro do aspecto.
@AfterReturning, indica que irá interceptar todos os métodos anotados pela annotation @ResolveDependencies.
Método resolve(retVal), acho que fica óbvio que retVal é o valor retornado pelo método anotado por @ResolveDependencies. No caso o retorno é uma instância de User.
SpringUtils.autowire, resolve todas as dependências do objeto retornado contidas em applicationContext.

Para finalizar, a entidade User, a anotação @ResolveDependencies e o application-context.xml

User.java

```
public class User {

	private Long id;

	@Inject
	@Named("userRepository")
	private UserRepository userRepository;

	public User(Long id) {
```

```
		this.id = id;
	}

	public User(Long id, UserRepository userRepository) {
		this.id = id;
		this.userRepository = userRepository;
	}

	public Long getId() {
		return id;
	}

	public void save() {

		if(userRepository == null)
			throw new IllegalStateException("userRepository == null");

		userRepository.save(this);

	}

}
```

ResolveDependencies.java

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ResolveDependencies {
}
```

application-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns:context="http://www.springframework.org/schema/context"
	xmlns:aop="http://www.springframework.org/schema/aop"
	xsi:schemaLocation="http://www.springframework.org/schema/aop http://
www.springframework.org/schema/aop/spring-aop-3.0.xsd
		http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans-3.0.xsd
		http://www.springframework.org/schema/context http://
www.springframework.org/schema/context/spring-context-3.0.xsd">

	<!-- Procura por classes anotadas nos sub-pacotes do base-package -->
	<context:component-scan base-package="br.com.submundojava" />

	<!-- Ativa suporte a aspectos -->
	<aop:aspectj-autoproxy />

</beans>
```

Executando

```
public static void main(String argz[]) {

		ApplicationContext ctx = new
ClassPathXmlApplicationContext("classpath:application-context.xml");

		UserRepository repository = ctx.getBean("userRepository",
UserRepository.class);

		User user = repository.findById(1L);
		user.save();

}
```

A saída deverá ser algo parecido com:

```
salvando user id=[1]
```

Conclusão

O que tentei demonstrar aqui é uma maneira simples de resolver dependências de objetos que não fazem parte do container de injeção de dependências. Esta abordagem está sendo utilizada atualmente nos meus projetos e tem solucionado uma porção de problemas.

Para ter uma melhor visualização você pode abaixar o projeto aqui. Certifique-se de ter java5+ e maven2 instalados.

Dúvidas, críticas ou sugestões. Deixe um comentário.

Abraços.