UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

Relatórios Técnicos
do Departamento de Informática Aplicada
da UNIRIO
n° 0003/2012

Propagação de Identidade com o Uso do Framework Spring

Felipe Leão
Leonardo Guerreiro Azevedo
Fernanda Baião
Claudia Cappelli

Departamento de Informática Aplicada

UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
Av. Pasteur, 458, Urca - CEP 22290-240
RIO DE JANEIRO – BRASIL

Projeto de Pesquisa

Grupo de Pesquisa Participante

Patrocínio

PETROBRAS

Relatórios Técnicos do DIA/UNIRIO, No. 0003/2012
Editor: Prof. Gleison Santos Agosto, 2012

Propagação de Identidade com o Uso do Framework Spring

Felipe Leão, Leonardo Guerreiro Azevedo, Fernanda Baião, Claudia Cappelli

Núcleo de Pesquisa e Prática em Tecnologia (NP2Tec)
Departamento de Informática Aplicada (DIA) – Universidade Federal do Estado do Rio de Janeiro (UNIRIO)

{felipe.leao, azevedo, fernanda.baiao, claudia.cappelli }@uniriotec.br

Abstract. Information security is a critical issue for organizations and comprises several perspectives. Identity propagation and data access control are some of those important issues in current n-tiers information systems architectures. This work describes the design and implementation of an architecture for effectively assuring information security, based on standard technologies allied to modifications to standard objects of the Spring framework, intending to propagate the user identity, in a non-intrusive way, from client applications until the DBMS where the authorization rules are enforced using a RBAC approach. Experimental testes were executed to evaluate the proposal concerning concurrent execution and performance.

Keywords: Database security, Identity Propagation, Dependency Injection, Data Access Control, Authorization Rules, TPC-H Benchmark.

Resumo. Segurança da informação é uma questão importante para organizações e inclui várias perspectivas. Propagação de identidade e controle de acesso a dados são algumas destas questões considerando arquiteturas de sistemas de informação em múltiplas camadas. Este trabalho descreve o projeto e a implementação de uma arquitetura para efetivamente garantir segurança de informação, baseado em tecnologias padrões incluindo modificações em objetos padrões do *framework* Spring para propagação da identidade do usuário, de uma forma não intrusiva, das aplicações cliente até o SGBD onde a regra de autorização deve ser garantida usando uma abordagem RBAC. Testes experimentais foram realizados a fim de avaliar a proposta quando a execução concorrente e de desempenho.

Palavras-chave: Segurança em Banco de Dados, Propagação de Identidade, Injeção de Dependência, Controle de Acesso a Dados, Regras de Autorização, Benchmark TPC-H.

Sumário

Figuras

1 Introdução

Existem diversos cenários de arquitetura distribuída dos sistemas de informação atuais, tais como os cenários apresentados por [Azevedo *et al.*, 2009; Leão *et al.*, 2011b]: cliente acessa banco diretamente executando um processo identificável; cliente acessa o banco diretamente executando um processo não identificável; cliente acessa servidor de aplicação que acessa o banco; e cliente acessa servidor de aplicação via serviços web. Nestes cenários, as camadas de um sistema (aplicação cliente, servidor de aplicação, componente de acesso a dados, servidor de dados) encontram-se implementadas em componentes independentes e alocadas a unidades de processamento distintas.

Os principais mecanismos para implementação do controle de acesso de usuários nestes cenários são a propagação de identidade entre as camadas do sistema de informação e a aplicação de regras de autorização. Estes mecanismos fazem com que a identidade do usuário seja considerada quando do acesso ao dado de modo que sejam manipulados apenas os dados que o usuário tem acesso e também que a aplicação tenha conhecimento do que este determinado usuário pode fazer com a informação acessada.

Portanto, uma solução para o aspecto de controle de acesso em aplicações passa por implementar mecanismos relacionados à definição dos perfis e de regras de autorização; à autenticação do usuário; à propagação de identidade do usuário autenticado entre as camadas do sistema até o servidor de dados e, finalmente, uma vez que o usuário atual do sistema de informação tenha sido reconhecido pelo SGBD (Sistema de Gerenciamento de Banco de Dados), à execução das regras de autorização em tempo de execução. A implementação de mecanismos relacionados à definição de perfis e regras de autorização foi tratada por Azevedo *et al*. [2010] é empregada neste trabalho. A implementação de mecanismos de autenticação do usuário está fora do escopo deste trabalho, sendo considerado que as informações existentes do usuário na aplicação está devidamente autenticada.

Este trabalho tem o objetivo de apresentar solução para propagação de identidade considerando o cenário "cliente acessa servidor de aplicação que acessa o banco", ilustrado na Figura 1. A proposta de solução é baseada na tecnologia Spring.

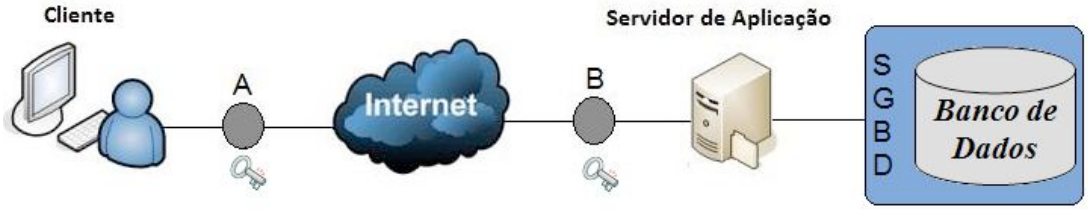

Figura 1 – Cenário de conexão com banco de dados "Cliente acessa servidor de aplicação que acessa o banco"

Este relatório foi produzido pelo Projeto de Pesquisa em Autorização de Informação como parte das iniciativas dentro do contexto do Projeto de Pesquisa do Termo de Cooperação entre UNIRIO/NP2Tec e a PETROBRAS/TIC-E&P/GIDSEP.

Esse relatório está organizado em 6 capítulos, sendo o Capítulo 1 a presente introdução. O Capítulo 2 apresenta o *framework* Spring. O Capítulo 3 apresenta a arquitetura utilizada como base para especificação desta solução. O Capítulo 4 apresenta a proposta de solução para propagação de identidade, enquanto que o Capítulo 5 apresenta as conclusões do trabalho.

2 Framework Spring

O Spring[1] é um *framework open source* criado por Rod Johnson, que tem como objetivo prover múltiplas funcionalidades a desenvolvedores de aplicações Java. Ele se tornou conhecido por facilitar a utilização do padrão de projeto *Dependency Injection*[2]. Spring é modular, permitindo que se utilize somente as partes que se julgarem necessárias.

O *framework* oferece funcionalidades para, dentre outras coisas, Inversão de Controle (*Inversion of Control,* base para o padrão *Dependency Injection*), Programação Orientada a Aspectos (AOP – *Aspect-Oriented Programming*), Acesso Remoto (*Remote Access*) e MVC (Model-View-Controller) [Walls e Braidenbach, 2007][Johnson *et al.*, 2010; Prasanna, 2009].

2.1 Injeção de Dependência com Spring

Segundo Prasanna [2009] e Walls e Breidenbach [2007], o framework suporta, principalmente, injeções através de métodos *setter* (*Setter Injection*) e construtores, tendo sido criado para diminuir a complexidade no desenvolvimento de aplicações corporativas. Burke e Monson-Haefel [2006] observam ainda que com o uso de Spring é possível utilizar objetos *Java Bean* para realizar atividades onde antes era necessário o uso de EJBs (componentes do framework EJB).

Injeção de Dependência é um processo no qual os objetos definem suas próprias dependências, ou seja, outros objetos com os quais ele trabalha. Estas dependências podem ser definidas através de argumentos do construtor, em um método *factory* ou através de propriedades que são ajustadas em uma instância do objeto após sua construção [Johnson *et al.*, 2010]. As dependências devem ser indicadas em tempo de implementação, mas serão dinamicamente fornecidas em tempo de execução pelo container IoC (responsável por proporcionar a Inversão de Controle – *Inversion of Control*) correspondendo à configuração realizada pelo desenvolvedor.

As duas variantes mais importantes da Injeção de dependência são *Setter Injection* (Injeção por método *Set)* e *Constructor Injection* (Injeção por Construtor de Classe), detalhadas na seção a seguir. Posteriormente é explicado o uso de anotações para realização de injeções.

2.1.1 *Setter Injection* (Injeção por método *Set*)

A injeção por método *set* recebe este nome porque é realizada pelo container através de métodos *set* dos atributos de uma classe, invocados após a construção de um objeto através de um construtor sem argumentos [Johnson *et al.*, 2010].

Como exemplo, considere a classe **Cliente**, que possui apenas dois atributos, *cpf* e *endereco*, descrita pela Figura 2. A classe descreve os dois atributos, com seus respectivos métodos *set* e um construtor *default*. Quando o Spring instancia um objeto desta classe não é necessário que seja informado parâmetro algum, dado que seu construtor não possui argumentos. Entretanto, alternativamente, o Spring permite que valores para suas propriedades (*cpf* e *endereco*) sejam injetadas no momento em que a instância é criada.

[1] http://www.springsource.org/

[2] http://martinfowler.com/articles/injection.html

```
class Cliente{
    private String cpf;
    private Endereco endereco;

    public void setCpf (String cpf){
        this.cpf = cpf;
    }

    public void setEndereco (Endereco endereco){
        this.endereco = endereco;
    }

    public Cliente(){}
}
```

Figura 2 - Classe Cliente

A Figura 3 mostra como é configurada a injeção via métodos *Setter*. Quando o *bean* referenciado pelo id "cliente" é instanciado, o container irá procurar pelos métodos *set* para as duas propriedades a serem injetadas, no caso os métodos *setCpf* e *setEndereco*, passando como parâmetros, respectivamente, o valor descrito no atributo *value* e o *bean* referenciado pelo id *enderecoComercial*. O *bean* *enderecoComercial* é então instanciado para ser passado como parâmetro, e uma sequência de injeções via método *set* é executada.

```
<beans>
    <bean id="cliente" class="br.uniriotec.propid.Cliente">
        <property name="cpf" value="123.456.789-09" />
        <property name="endereco" ref="enderecoComercial"/>
    </bean>

    <bean id="enderecoComercial" class="br.uniriotec.propid.Endereco">
        <property name="estado" value="RJ" />
        <property name="cidade" value="Rio de Janeiro" />
        <property name="rua" value="Avenida República do Chile" />
        <property name="numero" value="65" />
        <property name="cep" value="28.999-999" />
    </bean>
</beans>
```

Figura 3 - Injeção através de método *Setter*

2.1.2 *Constructor Injection* (Injeção por Construtor de Classe)

A injeção de dependência através de construtores de classe é realizada pelo container através da invocação do construtor que comporte argumentos representando as dependências do objeto [Johnson *et al.*, 2010].

Considere uma nova versão da classe Cliente mostrada pela Figura 2. Nesta nova versão (Figura 4) a classe Cliente ainda possui os mesmos dois atributos (*cpf* e *endereco*) e seus métodos *setter*; entretanto, ela possui ainda um construtor sobrecarregado que recebe dois argumentos, uma *String* que representa o CPF do cliente e um objeto *Endereco*, representando o endereço do cliente.

```
class Cliente{
    private String cpf;
    private Endereco endereco;

    public void setCpf (String cpf){
        this.cpf = cpf;
    }

    public void setEndereco (Endereco endereco){
        this.endereco = endereco;
    }

    public Cliente(){}

    public Cliente(String cpf, Endereco endereco){
        this.cpf = cpf;
        this.endereco = endereco;
    }
}
```

Figura 4 - Classe Cliente com construtor sobrecarregado

Com esta nova versão do construtor da classe, é possível utilizar o Spring para construir o objeto realizando a injeção das propriedades diretamente através do construtor da classe, como mostrado pela Figura 5.

```
<beans>
    <bean id="cliente" class="br.uniriotec.propid.Cliente">
        <constructor-arg name="cpf" value="123.456.789-09"/>
        <constructor-arg name="endereco" ref="enderecoComercial"/>
    </bean>

    <bean id="enderecoComercial" class="br.uniriotec.propid.Endereco">
        <property name="estado" value="RJ" />
        <property name="cidade" value="Rio de Janeiro" />
        <property name="rua" value="Avenida República do Chile" />
        <property name="numero" value="65" />
        <property name="cep" value="28.999-999" />
    </bean>
</beans>
```

Figura 5 - Injeção de Dependência através de Construtor

No trecho ressaltado na Figura 5, a *tag* utilizada (`<constructor-arg>`) tem por intenção informar um argumento que deve ser passado ao construtor da classe. O container irá localizar o construtor da classe capaz de receber um argumento *String* e outro da classe *Endereco*, neste caso referenciado pelo *bean enderecoComercial*, e injetará os valores relacionando o atributo *name* da tag ao nome do atributo na assinatura do construtor.

A tag `<constructor-arg>` poderia ainda omitir alguns detalhes ou especificá-los ainda mais. Sendo os valores passados como argumentos de dois "tipos" diferentes (no caso um é uma String enquanto o outro é da classe "Endereco") a especificação do nome do argumento é desnecessária, pois o container conseguiria fazer a associação ao passar os valores para o construtor da classe. Outra possível forma de fazer a associação de valores a variáveis específicas da assinatura do construtor seria o uso do índice do argumento, exemplificado na Figura 6.

```
<bean id="cliente" class="br.uniriotec.propid.Cliente">
    <constructor-arg index="0" value="123.456.789-09"/>
    <constructor-arg index="1" ref="enderecoComercial"/>
</bean>
```

Figura 6 - Injeção de Dependência através de Construtor com parâmetro *index*

Observe que a diferença na configuração entre injeção por *setter* e injeção por construtor está no fato da primeira usar propriedades (*properties*) e a segunda usar o argumento de construtor (*constructor-arg*). Obviamente, a segunda requer um construtor com argumentos, enquanto que a primeira não.

2.1.3 Uso de anotações para realizar Injeção de Dependências

Assim como em muitos *frameworks*, o Spring disponibiliza um conjunto de anotações capazes de simplificar o desenvolvimento de aplicações. Algumas destas anotações auxiliam a injeção de dependências em objetos java. Duas das principais anotações são a `@Required`, a `@Autowired` e `@Inject`, pertencentes ao pacote `org.springframework.beans.factory.annotation`.

As três anotações possibilitam a simplificação da configuração dos beans Spring. Todas devem ser utilizadas nas classes antes dos métodos *setter*, desta forma é possível omitir no arquivo de configuração as *tags* <*property*> que referenciam outros beans do Spring para injeção através de métodos *Set*. Um exemplo de uso da anotação *@Autowired* pode ser visto na Figura 7, que exibe a versão original da classe Cliente mostrada na seção 2.1.1 porém com a anotação antecedendo o método *setEndereco*.

```
class Cliente{
    private String cpf;
    private Endereco endereco;

    public void setCpf (String cpf){
        this.cpf = cpf;
    }

    @Autowired
    public void setEndereco (Endereco endereco){
        this.endereco = endereco;
    }

    public Cliente(){}

}
```

Figura 7 - Classe Cliente utilizando a anotação *@Autowired*

A classe Cliente requisita, através da anotação *@Autowired*, que uma dependência sua seja satisfeita. O container irá identificar a classe requisitada pelo método e procurará um *bean* no arquivo de configuração que implemente a classe desejada. O *bean* encontrado será então injetado no objeto recém instanciado. A Figura 8 mostra como ficaria a configuração do Spring para instanciar um *bean* da classe Cliente. Deve-se observar que o xml já não mais referencia que uma propriedade "endereco" deve ser injetada no *bean*.

```
<beans>
    <bean id="cliente" class="br.uniriotec.propid.Cliente">
        <property name="cpf" value="123.456.789-09"/>
    </bean>

    <bean id="enderecoComercial" class="br.uniriotec.propid.Endereco">
        <property name="estado" value="RJ" />
        <property name="cidade" value="Rio de Janeiro" />
        <property name="rua" value="Avenida República do Chile" />
        <property name="numero" value="65" />
        <property name="cep" value="28.999-999" />
    </bean>
</beans>
```

Figura 8 - Criação do *bean* no arquivo de configuração do Spring

As anotações *@Required* e *@Inject* funcionam de forma similar ao *@Autowired*. Entretanto, o primeiro especifica uma dependência que deve ser obrigatoriamente satisfeita, caso contrário a aplicação levantará uma exceção, enquanto a segunda especifica uma dependência opcional. O *@Autowired* é mais versátil pois pode especificar dependências obrigatórias ou opcionais (caso no qual deve receber o atributo "*required=false*"). A Anotação *@Autowired* pode ainda ser utilizada em métodos construtores ou diretamente em propriedades ao invés do seu método *setter* respectivo. Quando utilizado em um método que possui mais de um argumento o container considerará todos os argumentos como dependências, tentando localizar beans que satisfaçam todos estes argumentos.

2.2 Exposição de um serviço Spring utilizando HTTP Invokers

Spring fornece classes de integração que visam suportar a exposição remota de *beans* através de várias tecnologias. O suporte remoto facilita o desenvolvimento de serviços remotos implementados diretamente através de POJOs padrão do Spring. Atualmente, o *framework* Spring suporta 6 mecanismos para exposição e acesso de *beans* remotos, os quais são: Remote Method Invocation (RMI), HTTP Invoker (próprio do Spring), Hessian, Burlap, JAX-RPC e JMS. Considerando o cenário utilizado para pesquisa e a solução a ser proposta, este trabalho abordará somente o mecanismo HTTP Invoker.

O Spring suporta a exposição de serviços usando RMI de forma transparente. A configuração é semelhante à configuração de um EJB remoto, exceto pelo fato de que não existe nenhum padrão para propagação de contexto seguro ou para propagação de transação. No entanto, o Spring provê pontos para contexto de invocações adicionais ao utilizar RMI *invoker* pelo uso de outros *frameworks* de segurança ou para credenciais seguras customizadas. Através do uso do `RmiServiceExporter`, um serviço pode ser exposto, e ele pode ser acessado usando `RmiProxyFactoryBean`. O `RmiServiceExporter` explicitamente suporta a exposição de serviços que não são conformes com RMI via RMI *invokers*. Johnson *et al.* [2010] detalham as configurações necessárias para expor e invocar serviços RMI utilizando Spring.

O HTTP Invoker permite serialização via HTTP similar ao RMI. Este mecanismo possui vantagens quando utilizado em um cenário onde tipos complexos são utilizados como parâmetros ou retornos e não podem ser serializados utilizando

mecanismos como Hessian[3] e Burlap[4] [Johnson *et al.*, 2010] (capítulo 19). A exposição do serviço se dá através da inserção de um *bean* no arquivo de configuração do Spring cujo atributo *name* é precedido por uma barra ("/") e a classe implementada é a `HttpInvokerServiceExporter`, pertencente ao pacote `org.springframework.remoting.httpinvoker`. Este *bean* deverá possuir duas propriedades, uma para referenciar o *bean* que de fato implementa a lógica do negócio e outra para indicar a interface de negócio a ser exposta. Um exemplo de implementação para a exposição de um serviço pode ser vista na Figura 9.

```
<bean name="/MeuServico"
    class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
  <property name="service" ref="idService"/>
  <property name="serviceInterface" value="pacote.InterfaceServico"/>
</bean>
```

Figura 9 – Configuração para exposição de um bean Spring como serviço através de HTTP Invoker

Do lado cliente da aplicação, a invocação de um *bean* exposto como serviço através de HTTP Invoker no servidor de aplicação também se dá através do Spring. Entretanto, no lado cliente, utiliza-se a classe `HttpInvokerProxyFactoryBean`, pertencente ao pacote `org.springframework.remoting.httpinvoker`.

O arquivo de configuração do Spring deverá comportar uma referência ao *bean* remoto, que deverá implementar a classe `HttpInvokerProxyFactoryBean` e possuir duas propriedades: o endereço remoto do serviço e a referência para a interface de negócio[5] do serviço. A Figura 10 mostra um exemplo de implementação para acessar o serviço considerado na Figura 9.

```
<bean id="servicoRemoto"
    class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  <property name="serviceUrl" value="http://remotehost:8080/remoting/MeuServico"/>
  <property name="serviceInterface" value="pacote.InterfaceServico"/>
</bean>
```

Figura 10 – Configuração para consumo de um bean Spring como Serviço através de HTTP Invoker

É recomendado que a interface de negócio, por ser utilizada tanto pelo lado cliente quanto pelo lado servidor, seja armazenada em um local compartilhado.

2.3 Escopos de *beans* do Spring

Quando a definição de um *bean* é criada, na realidade está sendo feita a configuração a ser utilizada pelo container para gerar uma instância de certa classe. Através da configuração é possível definir diversos aspectos, como, por exemplo, parâmetros ou *beans* que devem ser injetados na nova instância. Uma característica de grande importância a ser definida é o escopo da nova instância. Esta abordagem permite que determinados padrões sejam aplicados a objetos sem que código tenha que ser adicionado às respectivas classes. O Spring suporta nativamente 5 diferentes

[3] O Hessian é um protocolo binário para web services que tem como proposta eliminar a necessidade de frameworks grandiosos. Por ser binário é capaz de enviar dados binários sem a necessidade de adição de complementos ao protocolo.

[4] Burlap é um protocolo simples, baseado em XML, para a conexão de webservices. Se destaca pela sua leveza, sendo por vezes utilizado em pequenos clientes, como *applets*, ou aplicações J2ME.

[5] Interface de Negócio é a interface implementada pelo serviço a fim de descrever a assinatura dos métodos que podem ser invocados a partir de um componente externo que pretenda acessá-lo.

escopos (dos quais 3 estão disponíveis apenas quando se faz uso de um *ApplicationContext* para aplicações Web). Os tipos são [Johnson *et. al.*, 2010]:

- Singleton: Uma única instância do *bean* é criada para toda a aplicação, ou seja, sempre que uma nova instância for requisitada ao container, a mesma referência será devolvida à aplicação. O Spring utiliza este escopo como padrão para seus *beans*. Portanto, caso a definição do escopo seja omitida, o *bean* será instanciado com escopo Singleton.
- Prototype: Uma nova instância da classe é criada para cada requisição feita ao container do Spring.
- Request: Uma única instância do *bean* é criada para o ciclo de vida de uma determinada requisição HTTP, ou seja, cada requisição terá suas próprias instâncias do *bean* e estas instâncias se comportarão como Singleton durante o ciclo de vida da requisição. Este tipo de escopo somente é válido em contextos de aplicações web.
- Session: Similar ao escopo "request", uma única instância de um *bean* é criada para o ciclo de vida de uma sessão HTTP. Este tipo de escopo somente é válido em contextos de aplicações web.
- Global Session: Similar aos escopos "request" e "session", uma única instância do *bean* é criada para o ciclo de vida de uma sessão global HTTP. Tipicamente utilizado em contextos de *portlet*. Este tipo de escopo somente é válido em contextos de aplicações web.

3 Arquitetura do utilizada como base para a solução

Esta seção tem o objetivo de ilustrar a arquitetura que representa aplicação real omitida por questões de sigilo. Esta aplicação foi utilizada como base para a solução para propagação de identidade considerando o cenário 3 da arquitetura de aplicações descrito por Azevedo *et al.* [2009].

3.1 Arquitetura da aplicação

A aplicação tem arquitetura que segue o cenário 3 de conexão com o banco de dados denominado "Cliente acessa servidor de aplicação que acessa o banco", ilustrado na Figura 1. No entanto, este cenário foi evoluído para considerar injeção de dependência, acesso remoto e interceptadores para tornar o acesso a componentes remotos o menos impactante possível para aplicações existentes. Neste cenário, o cliente que deseja acessar o objeto remoto tem a sua requisição interceptada logo na sua saída (ressaltada com a marcação "A") para injeção de informações na requisição. Quando a requisição é recebida pelo servidor de aplicação, esta é interceptada novamente (marcação "B") para retirar da requisição as informações injetadas na saída, realizar um processamento e passar a requisição para o objeto remoto sem as informações injetadas em "A". Dessa forma, a injeção e a recuperação de informações não impactam quem envia a requisição e quem recebe a requisição. O cliente não precisa saber como ocorre a interceptação e o objeto remoto não precisa ser alterado para considerar novas informações injetadas na requisição. O *framework* Spring é responsável por tratar interceptação, injeção e extração de informações de segurança. É importante enfatizar que o framework é responsável também por fornecer ao objeto remoto uma conexão com o banco de dados configurada com as informações trafegadas na requisição.

Para implementação do cenário apresentado na Figura 1, a aplicação faz uso das funcionalidades de Injeção de Dependência e *Remote Access* fornecidas pelo framework Spring. Enquanto as funcionalidades de Injeção de Dependência são usadas da forma costumeira, servindo para auxiliar na diminuição do acoplamento entre módulos da aplicação, as funcionalidades do *Remote* auxiliam na localização e invocação de componentes de negócio disponibilizados no módulo servidor da aplicação como serviços Spring e invocados através de HTTP Invokers (conceito apresentado na Seção 2.2).

A implementação da aplicação foi separada em três módulos: Cliente, Servidor e Compartilhado (Figura 11). O primeiro módulo corresponde à parte da aplicação implantada na máquina utilizada pelo usuário, sendo responsável por fazer toda a interface com este. Este módulo faz acesso a componentes disponibilizados no módulo Servidor (segunda parte da aplicação), sendo estes responsáveis, por exemplo, por realizar o acesso ao banco de dados. A terceira parte da aplicação, o Módulo Compartilhado, foi concebida com o intuito de possibilitar a disponibilização unificada de componentes utilizados tanto pela camada Cliente quanto pela camada Servidor, como, por exemplo, classes de modelo, tais quais as que descrevem um Poço ou um Usuário, sendo portanto utilizada tanto pelo módulo cliente quanto pelo módulo servidor. Para que possa ser acessada por múltiplas aplicações o módulo compartilhado se encontra disponibilizado no servidor de aplicação

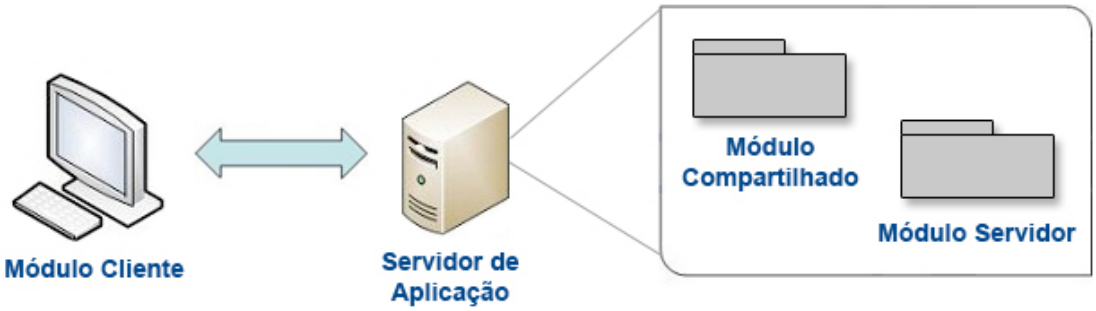

Figura 11 - Arquitetura modular de aplicação real

3.2 Análise do uso de Remote na arquitetura

Para que a camada Cliente seja capaz de acessar componentes da camada Servidor, componentes são disponibilizados como serviços Spring através do uso das funcionalidades *Remote* do framework, mais especificamente através das implementações de HTTP Invoker disponibilizadas, como apresentado na Seção 2.2

Originalmente, as classes `HTTPInvokerServiceExporter` e `HTTPInvokerProxyFactoryBean` (que são, respectivamente, as implementações padrão para uso do HTTP Invoker no lado Servidor e no lado Cliente de aplicações que utilizam Spring) não forneciam todas as funcionalidades necessárias ao funcionamento da aplicação. O principal problema encontrado foi a necessidade de não apenas invocar componentes remotos, mas sim passar informações de acesso a dados para os componentes sem ter que refazer as aplicações clientes e os componentes já disponibilizados no servidor. A dispersão dos registros em múltiplas tabelas, que por sua vez se encontram em múltiplos bancos de dados, exige que a aplicação considere mais de um DataSource[6] para conexão. Quando um usuário utiliza o sistema para acessar estes dados é necessário então que seja informado o DataSource que deverá ser utilizado para o acesso do usuário. A

[6] Configuração para a conexão com um banco de dados em um servidor.

existência de múltiplos DataSources deve permanecer transparente para o usuário. Portanto, fez-se necessário que esta informação fosse passada para o lado Servidor da aplicação de forma invisível.

Para informar ao Servidor o DataSource que deve ser utilizado, foram implementadas pela equipe de desenvolvedores da aplicação modificações às classes `HTTPInvokerServiceExporter` e `HTTPInvokerProxyFactoryBean`. As modificações foram possíveis com o uso do padrão de projeto *Composition*[7] e com o uso de herança. O esquema das classes criadas é apresentado na Figura 12.

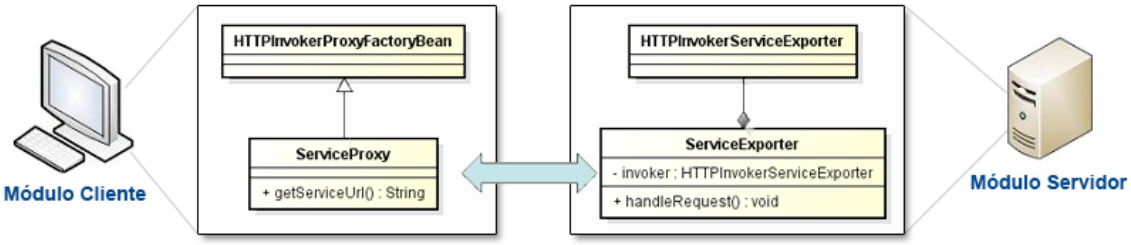

Figura 12 - Classes ServiceProxy e ServiceExporter

Inicialmente, no lado cliente, quando um *bean* do servidor é invocado, o `ServiceProxy` intercepta esta invocação e adiciona aos dados do "request" um novo parâmetro que carrega o código do DataSource que deverá ser utilizado. No lado servidor, o `ServiceExporter` intercepta a requisição e retira do "request" o parâmetro que especifica o DataSource. Uma vez que se tem conhecimento do DataSource a ser utilizado, o "request" é encaminhado para que a invocação do *bean* possa ser realizada.

A classe `ServiceProxy` visa substituir a classe `HttpInvokerProxyFactoryBean` (através de herança) no lado cliente da aplicação, cuja função é permitir o acesso a um *bean* disponibilizado no servidor de aplicação. A classe `ServiceProxy` sobrescreve o método `getServiceUrl` e, na URL de requisição do serviço sendo requisitado, adiciona o parâmetro informando o código do DataSource a ser utilizado. A implementação do método `getServiceUrl()` no `ServiceProxy` é apresentada na Figura 13.

```
@Override
public String getServiceUrl() {
    StringBuilder serviceUrl = new StringBuilder(super.getServiceUrl());
    //Recuperar o DataSourceId em uso pela aplicação
    DataSourceId dataSourceId = beanFactory.getBean(DataSourceId.class);
    //Se houver um DataSourceId sendo utilizado anexa o código ao request
    if (dataSourceId != null) {
        serviceUrl.append("?" + DataSourceId.PARAMETRO_DATA_SOURCE);
        serviceUrl.append("=" + dataSourceId.getCode());
    } else {
        throw new IllegalStateException(
            "Não foi possível encontrar o bean dataSourceId.");
    }
    return serviceUrl.toString();
}
```

Figura 13 - Método getServiceUrl() da classe ServiceProxy

[7] Padrão de projeto onde uma classe possui como atributo privado uma segunda classe e delega a execução de parte de seus métodos a esta segunda classe.

A classe `ServiceExporter` foi concebida com o intuito de substituir (através do padrão de projeto `Composition`) a classe `HttpInvokerServiceExporter`, padrão do Spring, utilizada para disponibilizar um *bean* no servidor de aplicação. Todos os métodos da `HttpInvokerServiceExporter` foram implementados na `ServiceExporter`. Em alguns casos, na classe `ServiceExporter`, apenas ocorre a invocação do método da classe `HttpInvokerServiceExporter`. Em outros casos, ocorre um pré-processamento dos parâmetros antes que o método da classe `HttpInvokerServiceExporter` seja invocado. O método ajustado com este intuito foi o `handleRequest`, objetivando recuperar no request o identificador do DataSource a ser utilizado. A implementação do método handleRequest() é apresentada na Figura 14.

```
@Override
public void handleRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    LOG.info("Executando servico: " + serviceBeanName + ".");
    String codigoDataSource = request.getParameter(DataSourceId.PARAMETRO_DATA_SOURCE);

    DataSourceId dataSourceId = beanFactory.getBean(DataSourceId.class);
    dataSourceId.setCode(Integer.parseInt(codigoDataSource));

    invoker.setService(beanFactory.getBean(serviceBeanName));
    invoker.afterPropertiesSet();
    invoker.handleRequest(request, response);
}
```

Figura 14 - Método handleRequest() da classe ServiceExporter

4 Proposta de solução para propagação de identidade para cenário 3

A solução proposta para o cenário 3 para o problema de propagação de identidade em uma aplicação cliente-servidor com o uso da tecnologia *Spring Remote* teve como base as alterações implementadas pelos desenvolvedores da aplicação às classes `HTTPInvokerServiceExporter` e `HTTPInvokerProxyFactoryBean`. Além disso, foi acrescentado na solução o uso de *beans* com escopo de requisição (*request-scoped beans* [8,9]), o uso de objeto dedicado à armazenar o contexto da aplicação Spring (Spring *ApplicationContext*[10]) e de um objeto responsável por prover sessões com o banco de dados recuperadas através do framework de mapeamento objeto-relacional Hibernate (SessionFactory[11]) . A solução considera a mesma arquitetura de três módulos apresentada na Figura 11: módulo cliente, módulo servidor e módulo compartilhado (para armazenar artefatos em comum aos dois primeiros módulos).

A arquitetura para prover a propagação de identidade foi projetada com o uso das tecnologias JavaEE (versão 5), JavaSE (versão 6), servidor de aplicação Jboss (versão 4.2.0), *framework* de mapeamento objeto-relacional Hibernate (versão 3) e o *framework* Spring para possibilitar Injeção de dependências e a disponibilização

[8] Um dado armazenado em um *request-scoped bean* é mantido durante toda a requisição

[9] O *request-scoped bean* foi utilizado para armazenar as informações do usuário para propagação de identidade.

[10] *ApplicationContext* é o objeto responsável por armazenar o contexto do Spring, provendo à aplicação toda a configuração realizada no *container* do *framework*.

[11] *SessionFactory* é uma classe cujo objetivo é prover sessões com o banco de dados. A inserção deste objeto possibilita manipular a sessão antes de devolvê-la para a aplicação, tarefa que não se faz possível quando a aplicação requisita a sessão diretamente à *session factory* do Hibernate.

remota de *JavaBeans* (versão 3.0.5). A IDE de programação Netbeans (versão 7) foi utilizada para criar os projetos. O Benchmark TPC-H foi utilizado para simular uma aplicação real de apoio à decisão. Para armazenar os dados utilizou-se o SGBD Oracle (versão 10g) com auxílio do *framework* FARBAC [Azevedo *et. al.*, 2010] para aplicação em tempo real de regras de autorização de acesso.

A Figura 15 apresenta o diagrama de classes para o protótipo proposto. Inicialmente foram criados dois projetos: um para representar o módulo Servidor (Módulo Servidor) e outro para representar o módulo Cliente (Módulo Cliente). Um terceiro projeto foi criado para comportar as interfaces que são utilizadas por ambos os projetos (cliente e servidor) além de classes que podem ser utilizadas por mais de um cliente ou mais de um servidor (as classes ServiceProxy, ServiceExporter e ApplicationContextProvider), este projeto representa o módulo Compartilhado.

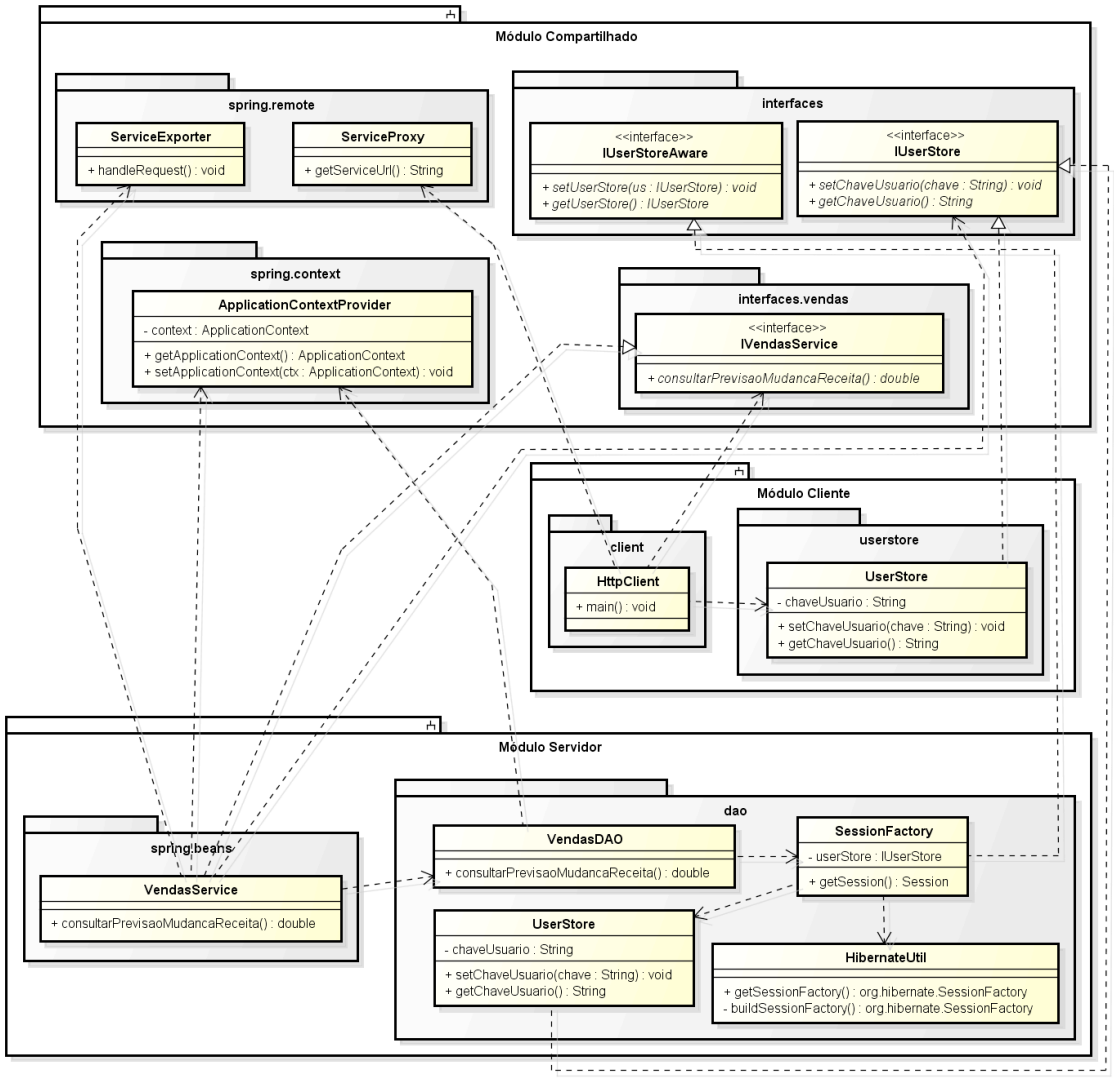

Figura 15 - Diagrama de Classes da Solução Proposta

A seguir são explicados em maiores detalhes os projetos criados para representar cada um dos três módulos e seus respectivos objetos.

4.1 Projeto Compartilhado

O projeto **compartilhado** reune todas as interfaces que são utilizadas pelos projetos cliente e servidor, além de algumas classes que foram isoladas para que pudessem ser reutilizadas em outros projetos (como o `ServiceProxy` e o `ServiceExporter`). O projeto criado é do tipo Aplicativo Java, já que o objetivo é

simplesmente gerar uma biblioteca capaz de armazenar classes e interfaces que serão disponibilizadas.

Inicialmente o projeto deve importar em seu claspath o *framework* Spring e a biblioteca servlet-api.jar[12]. Quatro pacotes foram criados (Figura 16), um para comportar as interfaces que são disponibilizadas pela biblioteca e podem ser utilizadas em múltiplas circunstâncias (pacote br.uniriotec.np2tec.propid.shared.interfaces), um para prover a interface de negócio da classe VendasService (a classe VendasService se encontra no projeto Servidor e será explicado detalhadamente na seção 4.2.2 ; o pacote que armazena a interface para esta classe é o br.uniriotec.np2tec.propid.shared.interfaces.vendas), um para armazenar as classes `ServiceProxy` e `ServiceExporter` que tem como função prover a funcionalidade *Remote* do Spring (pacote br.uniriotec.np2tec.propid.shared.spring.remote) e um terceiro para comportar classes para a manipulação do *ApplicationContext* do Spring (pacote br.uniriotec.np2tec.propid.shared.spring.context). A Figura 16 mostra a estrutura criada para o projeto, e as subseções a seguir detalham cada um dos pacotes criados.

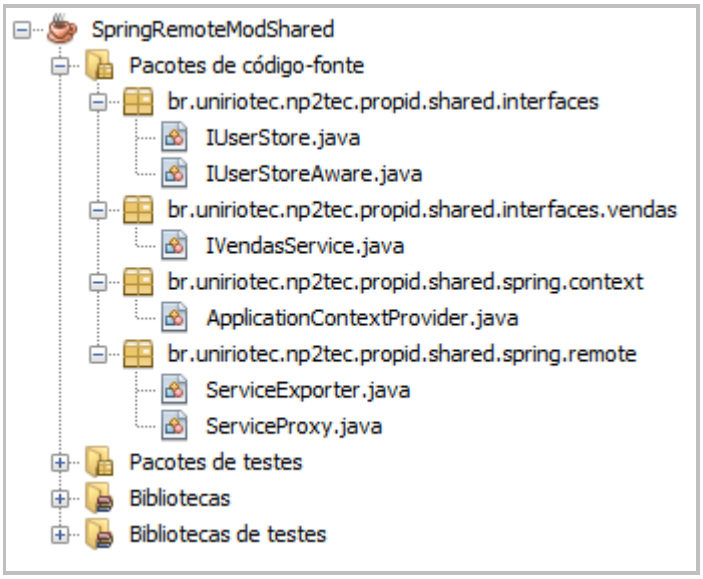

Figura 16 - Estrutura do projeto para o módulo compartilhado

4.1.1 Pacote *interfaces*

O pacote de *br.uniriotec.np2tec.propid.shared.interfaces* possui as interfaces:

- **IUserStore**, responsável por definir as operações obrigatórias de um objeto que armazena dados de um usuário; e
- **IUserStoreAware**, que define métodos necessários a um objeto onde se deseja injetar um objeto que implemente a interface IUserStore;

A Interface IUserStore (Figura 17) define os métodos *setChaveUsuario()* e *getChaveUsuario()*, enquanto a interface IUserStoreAware (Figura 18) define os métodos *setUserStore()* e *getUserStore()*.

[12] A biblioteca servlet-api.jar é fornecida junto ao servidor Tomcat (versão 6+) e pode ser encontrada no diretório *lib* do servidor.

```
public interface IUserStore {
    void setChaveUsuario(String chaveUsuario);
    String getChaveUsuario();
}
```

Figura 17 - Interface IUserStore

```
public interface IUserStoreAware {
    void setUserStore(IUserStore userStore);
    public IUserStore getUserStore();
}
```

Figura 18 - Interface IUserStoreAware

4.1.2 Pacote *interface.vendas*

O pacote *br.uniriotec.np2tec.propid.shared.interface.vendas* provê um único artefato, a interface IVendasService (Figura 19), cujo objetivo é servir de interface de negócio para a classe VendasService, pertencente ao projeto Servidor e acessada remotamente pelo projeto Cliente. Esta interface diz respeito ao negócio sendo tratado pela aplicação e foi criada devido ao cenário estabelecido para testes.

A interface IVendasService define um único método, o *consultarPrevisaoMudancaReceita()*.

```
public interface IVendasService extends Serializable{
    double consultarPrevisaoMudancaReceita() throws SQLException;
}
```

Figura 19 - Interface IvendasService

4.1.3 Pacote spring.*remote*

O pacote *br.uniriotec.np2tec.propid.shared.spring.remote* armazena as classes responsáveis por adaptar a funcionalidade das classes `HTTPInvokerServiceExporter` (adaptada pela classe ServiceExporter) e `HTTPInvokerProxyFactoryBean` (adaptada pela classe ServiceProxy), que são padrão do Spring e possibilitam o acesso a beans remotos.

A classe ServiceExporter faz uso da classe `HTTPInvokerServiceExporter` através do padrão *Composite* , e é implementada de forma semelhante à classe ServiceExporter descrita pela seção 3.2 . Para que a classe ServiceExporter possa ser utilizada no lugar da `HttpInvokerServiceProxy` é necessário o uso de algumas interfaces utilizadas pela classe original, forçando a nova a implementar determinados métodos necessários ao container Spring. Entre as interfaces utilizadas a HttpRequestHandler é de grande importância, pois implica na sobreescrita do método *handleRequest* (Figura 20), responsável por manipular uma requisição feita ao *bean* exposto remotamente. O uso da classe ServiceExporter permite realizar alterações ao método *handleRequest* original. Nesta nova versão, caso esteja presente no request um parâmetro com nome igual ao definido na variável estática PARAMETRO (neste caso, um parâmetro com o nome *chave*, definido na quarta linha da Figura 20), o método irá executar um processamento específico antes de encaminhar o request ao método *handleRequest* da classe `HttpInvokerServiceExporter`. Este processamento é responsável por recuperar no contexto do Spring um objeto que implemente a interface IUserStore e armazenar nele o valor recebido no parâmetro.

```
private HttpInvokerServiceExporter invoker = new HttpInvokerServiceExporter();
private BeanFactory beanFactory;
private ApplicationContext context;
private static final String PARAMETRO = "chave";

@Override
public void handleRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    if(request.getParameter(PARAMETRO) != null){
        String valorParametro = request.getParameter(PARAMETRO);

        context = ApplicationContextProvider.getApplicationContext();
        IUserStore userStore = (IUserStore) context.getBean("beanUserStore");
        //Injeção da chave na UserStore
        userStore.setChaveUsuario(valorParametro);
    }

    invoker.handleRequest(request, response);
}
```

Figura 20 - Método *handleRequest* da classe ServiceExporter

O valor armazenado poderá então ser utilizado por outros objetos, dependendo de sua configuração. Mais a frente, veremos que o projeto servidor utilizará um objeto UserStore singleton com escopo de request, possibilitando o acesso ao valor armazenado por qualquer classe.

A classe ServiceProxy (Figura 21) estende a classe `HTTPInvokerProxyFactoryBean`, sendo também semelhante a sua classe homônima descrita na seção 3.2 . Nesta classe, o método *getServiceUrl* é sobrescrito para que, caso um objeto que implemente a interface IUserStore tenha sido injetado no momento da criação da classe ServiceProxy, um parâmetro seja adicionado ao request contendo o valor armazenado neste objeto injetado. Este parâmetro poderá ser posteriormente interceptado pelo uso da classe ServiceExporter, como ilustrado no método `handleRequest` da Figura 20.

```
@Override
public String getServiceUrl() {
        StringBuilder serviceUrl = new StringBuilder(super.getServiceUrl());
        if (userStore != null) {
            int pos = serviceUrl.indexOf("?");
            if(pos == -1){
                serviceUrl.append("?");
            }else{
                serviceUrl.append("&");
            }
                serviceUrl.append("chave");
                serviceUrl.append("=");
                serviceUrl.append(userStore.getChaveUsuario());
        }

        return serviceUrl.toString();
}
```

Figura 21 - Método *getServiceUrl* da classe ServiceProxy

4.1.4 Pacote *spring.context*

O Pacote *br.uniriotec.np2tec.propid.shared.spring.context* possui uma única classe, a ApplicationContextProvider (Figura 22). Esta classe se encontra implementada no projeto compartilhado para permitir que ela seja utilizada por diferentes projetos. A

classe ApplicationContextProvider é uma implementação comum de um objeto que será responsável por armazenar a referência para o contexto do Spring. Desta forma, quando se deseja acessar o contexto em qualquer ponto da aplicação, não se faz necessário iniciá-lo novamente. Isto isola o processo de inicialização do framework em um único ponto da aplicação. Além disso, é garantido que o mesmo contexto será utilizado por toda a aplicação, preservando, por exemplo, classes inicializadas com escopo Singleton.

```
public class ApplicationContextProvider implements ApplicationContextAware {
    private static ApplicationContext contexto = null;

    public static ApplicationContext getApplicationContext() {
        return contexto;
    }

    public void setApplicationContext(ApplicationContext contexto)
            throws BeansException {
        // Assign the ApplicationContext into a static method
        ApplicationContextProvider.contexto = contexto;
    }
}
```

Figura 22 - Classe *ApplicationContextProvider*

Posteriormente, será visto no projeto Servidor que o uso de um único contexto por toda a aplicação se faz necessário para possibilitar a recuperação de um bean singleton que tenha em si um dado armazenado, por exemplo, um bean que implemente a interface IUserStore.

4.2 Projeto Servidor

O projeto Servidor consiste em um projeto Java Web (criado na IDE Netbeans como uma Aplicação Web da categoria Java Web) cujo objetivo é implementar uma determinada funcionalidade através de um *bean* exposto através das funcionalidades de *Remote Access* do *framework* Spring. O *bean* VendasService realiza esta função, implementando o método *consultaPrevisaoMudancaReceita*, cujo objetivo é verificar qual teria sido a variação na receita da empresa caso um determinado desconto não tivesse sido fornecido aos clientes daquela empresa.

O projeto faz uso dos *frameworks* Hibernate, C3P0[13] e Spring no lado da aplicação e FARBAC no SGBD Oracle, além dos padrões de projeto Injeção de Dependência, HibernateUtil, Singleton, *Factory* e *Data Access Objetct* (DAO). O benchmark TPC-H foi utilizado com a intenção de estabelecer um ambiente que simulasse uma aplicação de apoio a decisão. A estrutura do projeto servidor é apresentada na Figura 23.

[13] C3P0 é uma biblioteca que possibilita e facilita o controle de um pool de conexões com o banco de dados. O projeto se encontra disponível em http://sourceforge.net/projects/c3p0/.

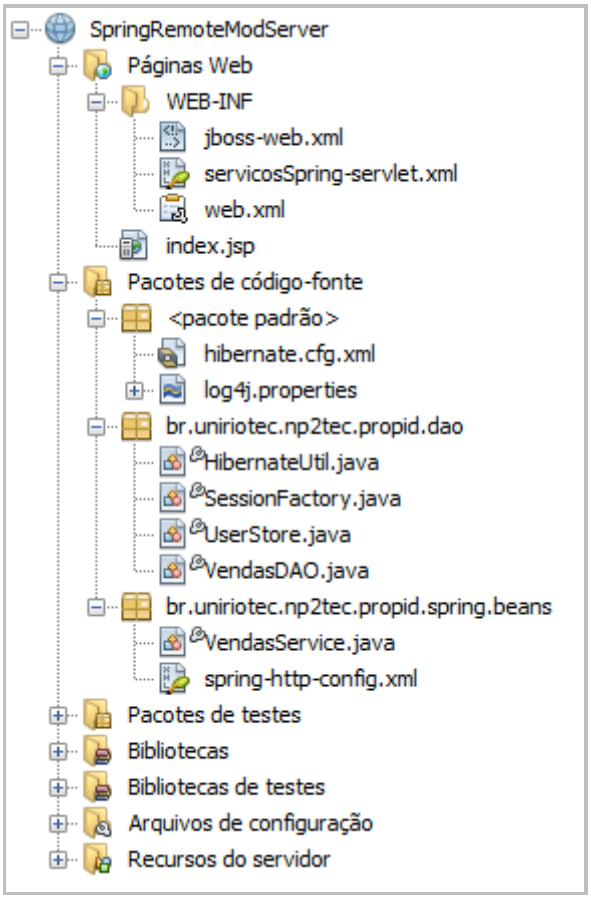

Figura 23 - Estrutura do projeto para o módulo servidor

Para disponibilizar o *bean* VendasService para acesso remoto, é necessário que o arquivo XML de configuração da aplicação faça uso da classe `HTTPInvokerServiceExporter` fornecida nativamente pelo *framework*, Este projeto, entretanto, fará uso da classe ServiceExporter provida pelo pacote `br.uniriotec.np2tec.propid.shared.spring.remote`. A exposição deste *bean* para acesso externo é explicada mais detalhadamente na seção 4.2.1 .

4.2.1 Diretório WEB-INF

O diretório WEB-INF armazena três arquivos de configuração utilizados pelo servidor web JBoss durante a implantação do projeto. O arquivo *jboss-web.xml* (Figura 24) especifica o caminho que o servidor deverá considerar como caminho de acesso à aplicação Web.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <jboss-web>
3    <context-root>/SpringRemoteModServer</context-root>
4  </jboss-web>
```

Figura 24 - Arquivo de configuração *jboss-web.xml*

O arquivo *web.xml* é comum a muitos servidores web e especifica redirecionamentos que devem ser realizados quando um determinado caminho da aplicação é acessado, como por exemplo para a página que deve ser exibida como "*homepage*" da aplicação ou para qual *servlet* uma determinada requisição deve ser encaminhada. Como a aplicação em questão faz uso do *framework* Spring, o arquivo de configuração *web.xml* é também utilizado para definir a inicialização do container do Spring, carregando o arquivo de configuração que especificar os *beans*. A Figura

25 mostra os principais parâmetros configurados no arquivo *web.xml* do projeto. Os principais parâmetros são detalhados a seguir.

```
<servlet>
    <servlet-name>servicosSpring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>servicosSpring</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>

<!-- CONFIGURAÇAO DO SPRING (chamada do arquivo de config) -->
<listener>
  <listener-class>
     org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/classes/br/uniriotec/np2tec/propid/spring/beans/spring-http-config.xml
    </param-value>
</context-param>
<!-- FIM DA CONFIGURAÇAO DO SPRING -->

<session-config>
    <session-timeout>30</session-timeout>
</session-config>
<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

Figura 25 - Arquivo de configuração *web.xml*

Três trechos específicos do arquivo fazem referência à configurações do Spring. A *tag* `<listener>` define que a classe ContextLoaderListener do Spring será utilizada como Listener da aplicação e a *tag* `<context-param>` indica a localização do arquivo que deverá ser utilizado para carregar o contexto do spring durante a inicialização da aplicação. Há ainda a *tag* `<servlet>` que mapeia o nome "servicosSpring" para a classe DispatcherServlet, também do *framework* Spring. O uso da classe DispatcherServlet fará com que um segundo arquivo de configuração seja utilizado para configurar o servlet em questão. O Spring irá procurar por um arquivo xml no diretório WEB-INF com o mesmo nome definido em `<servlet-name>` com o sufixo "-servlet". Neste caso se faz necessária a existência do arquivo *servicosSpring-servlet.xml*, cuja parte principal é mostrada pela Figura 26.

```
<bean id="httpVendasService" class="br.uniriotec.np2tec.propid.shared.spring.remote.ServiceExporter">
   <property name="service" ref="vendasService" />
   <property name="serviceInterface"
     value="br.uniriotec.np2tec.propid.shared.interfaces.IVendasService"/>
</bean>

<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
   <property name="mappings">
      <props>
         <prop key="/vendasService">httpVendasService</prop>
      </props>
   </property>
</bean>
```

Figura 26 - Arquivo de configuração*servicosSpring-servlet.xml*

A Figura 26 mostra a configuração de dois *beans* Spring configurados para o servlet *servicosSpring*. O *bean* referenciado pelo ID *urlMapping* estabelece que quando um request é recebido na url "vendasService" deve-se encaminhá-lo para o *bean* cujo ID é "httpVendasService".

O *bean* "httpVendasService" por sua vez serve para expor um segundo *bean* através da classe ServiceExporter, obtida através do uso do módulo compartilhado. O uso da classe ServiceExporter implica na definição de duas propriedades, uma para definir o *bean* que será exposto e outra para definir a interface de negócio do *bean* exposto. O *bean* exposto é o de ID "vendasService", citado posteriormente, na seção 4.2.2 ao falarmos sobre o pacote *spring.beans*, e como interface de negócio é utilizada a interface IVendasService presente no módulo compartilhado.

4.2.2 Pacote *spring.beans*

O pacote `br.uniriotec.np2tec.propid.spring.beans` comporta a classe vendasService e o arquivo de configuração *spring-http-config.xml*.

Como apresentado na seção 4.2.1 , ao implantar o projeto no servidor JBoss, alguns arquivos de configuração são utilizados, entre eles o *web.xml* que inicializará o contexto do Spring através do arquivo de configuração *spring-http-config.xml* (Figura 27).

```
<bean id="beanUserStore" class="br.uniriotec.np2tec.propid.dao.UserStore"
      scope="request">
   <aop:scoped-proxy proxy-target-class="false"/>
</bean>

<bean id="beanSessionFactory" class="br.uniriotec.np2tec.propid.dao.SessionFactory"
      scope="request">
   <property name="userStore" ref="beanUserStore" />
</bean>

<bean id="vendasService" class="br.uniriotec.np2tec.propid.spring.beans.VendasService" >
</bean>

<bean id="applicationContextProvider"
      class="br.uniriotec.np2tec.propid.shared.spring.context.ApplicationContextProvider"/>
```

Figura 27 - Arquivo de configuração *spring-http-config.xml*

Ao inicializar o contexto, quatro *beans* deverão ser pré-configurados, um deles implementa a classe ApplicationContextProvider, e armazenará o contexto a fim de fornecê-lo a outros pontos da aplicação quando necessário. O *bean* de ID "vendasService" implementa a classe VendasService, presente no mesmo pacote do arquivo de configuração, e será exposto através da funcionalidade *Remote Access* para acesso remoto por outras aplicações com o uso da classe ServiceExporter. O terceiro *bean*, "beanSessionFactory", faz o papel de ponte entre a aplicação e o Hibernate, abstraindo o uso do framework para a obtenção de sessões com o banco de dados e adicionando processamento ao processo de obtenção de uma sessão. Este *bean* recebe através de injeção o *bean* que armazena os dados do usuário da aplicação *beanUserStore*. Um quarto *bean* implementa a classe UserStore, do pacote *dao*, que será explicada posteriormente neste trabalho.

É interessante adiantar a explicação de que a classe UserStore implementa a interface IUserStore fornecida pelo módulo compartilhado e que é utilizada para armazenar a chave do usuário informada pela aplicação cliente durante o acesso ao *bean* remoto. O *bean* que implementa a classe UserStore tem seu escopo definido como *request*, de forma que sua vida útil se iniciará com o recebimento da requisição por parte da aplicação e encerrará ao fim da mesma. *Beans* com escopo de request podem ser enxergados exclusivamente pela *Thread* criada para realizar o processamento da requisição, garantindo o isolamento da chave de usuário recebida. O funcionamento dos outros tipos de escopo dos *beans* Spring foram detalhados na seção 2.3 .

A classe VendasService implementa a interface, IVendasService que faz o papel de interface de negócio, possibilitando expor um *bean* desta classe através da funcionalidade *Remote Access*. A implementação da classe pode ser vista na Figura 28.

```
public class VendasService implements IVendasService{
    private VendasDAO daoVendas = new VendasDAO();

    public double consultarPrevisaoMudancaReceita() throws SQLException {
        try{
            return daoVendas.consultarPrevisaoMudanca();
        }catch(Exception e){
            throw new SQLException(e.getMessage());
        }
    }

}
```

Figura 28 - Implementação da classe VendasService

A classe possui um método principal, o *consultarPrevisaoMudancaReceita()*. Este método tem como objetivo retornar a variação na receita de uma determinada empresa caso não fosse aplicado um determinado desconto a todos os clientes daquela empresa. Esta consulta foi implementada no método *consultarPrevisaoMudanca* da classe VendasDAO. A consulta é criada sobre a estrutura do *benchmark* TPC-H, que simula uma aplicação de apoio à decisão. Posteriormente, será visto que a consulta é realizada sobre uma sessão com o banco de dados onde o usuário da aplicação cliente é conhecido.

4.2.3 Pacote *dao*

O pacote `br.uniriotec.np2tec.propid.dao` foi criado para comportar as classes responsáveis por armazenar dados durante a execução da aplicação (caso da classe UserStore) ou para possibilitar o acesso à bases externas de dados.

A classe VendasDAO (Figura 29) atende ao padrão de projeto DAO (*Data Access Object*) fornecendo um objeto que se responsabilize por encapsular a complexidade do acesso à base de dados. No caso, a aplicação servidor faz uso do *framework* Hibernate para acessar o banco de dados. Desta forma, a classe disponibiliza somente um método (*consultarPrevisaoMudanca()*) que recupera através da SessionFactory uma sessão com o banco de dados e realiza a consulta que retorna o valor da receita da empresa fictícia para o caso de um determinado desconto não ter sido dado aos clientes daquela empresa.

```
public class VendasDAO {

    private SessionFactory sessionFactory = null;

    public double consultarPrevisaoMudanca() throws Exception{
        ApplicationContext ctx = ApplicationContextProvider.getApplicationContext();
        sessionFactory = (SessionFactory) ctx.getBean("beanSessionFactory");

        String sql = "select sum(l_extendedprice * l_discount) as revenue "+
                     "from tpch.lineitem "+
                     "where l_shipdate >= date '1994-01-01' "+
                      "and l_shipdate < date '1994-01-01' + interval '1' year "+
                      "and l_discount between 0.06 - 0.01 and 0.06 + 0.01 "+
                      "and l_quantity < 24";

        Session s = sessionFactory.openSession();
        Connection con = s.connection();
        PreparedStatement ps = con.prepareStatement(sql);
        ResultSet rs = ps.executeQuery();
        s.close();

        double resultado = 0;
        while(rs.next()){
            resultado = rs.getDouble("revenue");
        }

        return resultado;
    }
}
```

Figura 29 - Implementação da classe VendasDAO

Para executar a consulta a classe VendasDAO necessita de uma sessão com o banco de dados, tal sessão é fornecida pela classe SessionFactory, sendo ambas as classes disponibilizadas pelo mesmo pacote. A classe SessionFactory implementa a interface IUserStoreAware e atua como intermediária no acesso de algumas das funcionalidades disponibilizadas pela fábrica de sessões no *framework* Hibernate, como, por exemplo, o método "openSession()" (Figura 30). O método tem como objetivo recuperar uma sessão com o banco de dados e fornecê-la à aplicação para que esta possa realizar consultas, invocar *procedures*, executar operações CRUD, etc. Uma sessão com o banco é aberta através do Hibernate (chamada ao método openSession da fábrica de sessões do Hibernate) e antes de devolver a sessão à aplicação, o método executa a chamada à procedure do SGBD Oracle *dbms_application_info.set_client_info()*, responsável por ajustar no contexto da sessão com o banco qual usuário está realizando acesso aos dados. Através desta identificação, o *framework* FARBAC pode aplicar regras de autorização de acesso aos dados. A SessionFactory tem conhecimento deste usuário devido à injeção do *bean* userStore, que armazena os dados do usuário e é inicializado pela classe ServiceExporter.

```
public Session openSession(){
    Session s = HibernateUtil.getSessionFactory().openSession();

    if(userStore != null){
        //preparar o contexto da sessao no banco com a chave do usuario
        Connection con = s.connection();
        try{
            CallableStatement st = con.prepareCall("{call dbms_application_info.set_client_info(?)}");
            st.setString(1, userStore.getChaveUsuario());
            st.executeUpdate();
        }catch(SQLException e){
            return null;
        }
    }

    return s;
}
```

Figura 30 - Método openSession() da classe SessionFactory

O *bean* userStore instancia a classe UserStore e é definido no contexto do Spring com escopo de *request*, portanto um dado armazenado nele no início da requisição pode ser acessado durante toda a requisição.

A classe UserStore (Figura 31) é uma classe POJO que implementa a interface IUserStore, disponibilizada pelo módulo compartilhado. O único objetivo desta classe no projeto é possibilitar o armazenamento da chave do usuário em um objeto capaz de manter seu estado por toda a vida útil da requisição. O ciclo de vida do *bean* é gerenciado pelo *framework* Spring e ocorre devido ao parâmetro *scope* cujo valor é ajustado como *request*.

```
18  public class UserStore implements IUserStore{
19      private String chaveUsuario;
20
21      public String getChaveUsuario() {
22          return chaveUsuario;
23      }
24
25      public void setChaveUsuario(String chaveUsuario) {
26          this.chaveUsuario = chaveUsuario;
27      }
28  }
```

Figura 31 - Implementação da classe UserStore

A classe HibernateUtil, utilizada pela classe SessionFactory, é a implementação de um padrão de projeto proposto pelos desenvolvedores do *framework* Hibernate que tem por objetivo centralizar em um único ponto da aplicação a inicialização do *framework*. Esta classe possui métodos estáticos, constituindo na prática um objeto que se comporta como um Singleton.

4.2.4 Pacote *Padrão*

O pacote padrão do projeto é criado de forma automática pela IDE Netbeans quando um arquivo é adicionado à raiz do projeto. Este pacote guarda dois arquivos de configuração do *framework* Hibernate.

O arquivo *hibernate.cfg.xml* (Figura 32) comporta a configuração a ser utilizada pelo *framework* para acessar o banco de dados. Neste arquivo são descritos, por exemplo, o usuário e a senha a serem utilizados para acesso ao banco de dados e o *driver* e URL de conexão com o banco de dados. O arquivo comporta ainda a configuração da biblioteca C3P0, utilizada para controlar o pool de conexões do banco de dados.

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
                "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
                "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

        <session-factory>

                <!-- Database connection settings -->
                <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
                <property name="connection.url">jdbc:oracle:thin:@10.0.0.121:1521:ORCL</property>
                <property name="connection.username">tpch</property>
                <property name="connection.password">*****</property>
                <!-- SQL dialect -->
                <property name="dialect">org.hibernate.dialect.Oracle10gDialect</property>

                <!-- Enable Hibernate's automatic session context management -->
                <property name="current_session_context_class">thread</property>

                <!-- Disable the second-level cache -->
                <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

                <!-- Echo all executed SQL to stdout -->
                <property name="show_sql">true</property>

                <property name="hibernate.connection.provider.class">
                        org.hibernate.connection.C3P0ConnectionProvider
                </property>
                <property name="hibernate.c3p0.min_size">1</property>
                <property name="hibernate.c3p0.max_size">2</property>
                <property name="hibernate.c3p0.timeout">180</property>
                <property name="hibernate.c3p0.idle_test_period">100</property>

        </session-factory>

</hibernate-configuration>
```

Figura 32 - Arquivo de configuração *Hibernate.cfg.xml*

O arquivo *log4j.properties* define configurações necessárias à biblioteca Apache LOG4J[14], utilizada pelo *framework* Hibernate.

4.3 Projeto Cliente

Para testar o módulo Servidor implementado foi criado um projeto cliente, cujo objetivo é consumir o *bean* exposto através da tecnologia Spring *Remote*. O módulo faz uso do *framework* Spring e do módulo compartilhado, sendo implementado como um projeto Java *desktop*.

Não há autenticação no projeto que representa o módulo cliente. É considerado que o escopo do trabalho se limita ao processo de propagação da identidade do usuário, sendo a autenticação uma responsabilidade externa. Entretanto, considera-se que o cenário onde será aplicada a solução proposta conta com uma fase de autenticação e espera-se que o módulo autenticador possa ser adaptado para que uma vez que o acesso tenha sido permitido, a chave do usuário do sistema seja armazenada em algum objeto que permaneça ativo durante a sessão. Para tal, um objeto que implemente a interface IUserStore (provida pelo módulo compartilhado) deve ser criado. Este objeto será acessado posteriormente a fim de recuperar a chave do usuário e injetá-la na requisição feita ao módulo servidor.

O projeto respeita uma estrutura simples (Figura 33), composta por dois pacotes, um para armazenar a classe que executa o projeto e o arquivo de configuração do Spring (`br.uniriotec.np2tec.propid.spring.cliente`) e outro para comportar o bean que armazena a chave do usuário (`br.uniriotec.np2tec.propid.spring.userstore`).

[14] A biblioteca Apache LOG4j PE um projeto apache com objetivo de prover serviços de log para aplicações Java. - http://logging.apache.org/log4j/

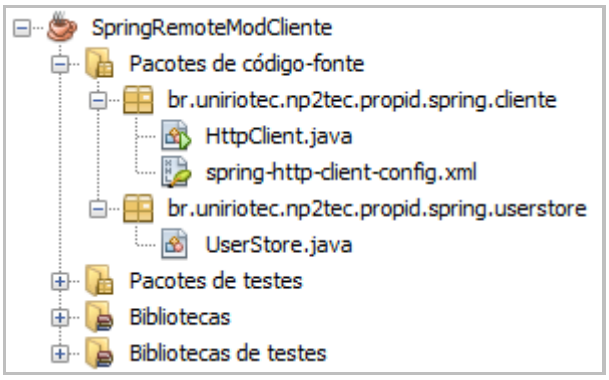

Figura 33 - Estrutura do projeto para o módulo Cliente

4.3.1 Pacote *userstore*

O pacote `br.uniriotec.np2tec.propid.spring.userstore` comporta uma única classe, a UserStore. Esta classe é um POJO que implementa a interface IUserStore fornecida pelo módulo compartilhado, assim como a classe homônima presente no pacote *dao* do módulo servidor, descrita na seção 4.2.3 .

4.3.2 Pacote *cliente*

Neste pacote, se encontram a classe que executa o projeto e o arquivo de configuração do contexto do Spring.

O arquivo *spring-http-client-config.xml* é responsável por carregar toda a configuração de *beans* que serão utilizados pela aplicação e que devem ser injetados pelo container do Spring (Figura 34). Somente dois *beans* são configurados. O *bean* com ID igual *beanUserStore* implementa a classe UserStore e será instanciado pelo container como um *Singleton* (já que ao omitir o atributo *scope* o bean automaticamente é configurado como *Singleton*), servindo para armazenar a chave do usuário. O segundo *bean* recebe o ID *vendasProxyService* e servirá para recuperar a referência de um *bean* exposto através da tecnologia *Remote* do Spring. Para isso, o bean deve ser da classe ServiceProxy, fornecida pelo módulo compartilhado. Três propriedades deverão ser injetadas no *bean*:

- *serviceUrl*: representa a URL do serviço que será referenciado pelo *bean*.
- *serviceInterface*: recebe como valor a interface de negócio do *bean* referenciado pela propriedade *serviceUrl*. Neste caso, uma interface fornecida pelo módulo compartilhado.
- *userStore*: deve ser um objeto que implementa a interface IUserStore e possui armazenada em si uma chave de usuário. Neste caso será injetado o *bean beanUserStore*.

```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:p="http://www.springframework.org/schema/p"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util-2.5.xsd">

   <bean id="beanUserStore" class="br.uniriotec.np2tec.propid.spring.userstore.UserStore" />

   <bean id="vendasProxyService" class="br.uniriotec.np2tec.propid.shared.spring.remote.ServiceProxy">
      <property name="serviceUrl"
                value="http://localhost:8081/SpringRemoteModServer/vendasService" />
      <property name="serviceInterface"
                value="br.uniriotec.np2tec.propid.shared.interfaces.IVendasService" />
      <property name="userStore" ref="beanUserStore" />
   </bean>

</beans>
```

Figura 34 - Arquivo de configuração *spring-http-client-config.xml*

A classe HttpClient (Figura 35) possui um único método responsável por executar o projeto, no caso da linguagem Java, um método *main*. Primeiramente, o método *main* inicializa o contexto do Spring com base no arquivo *spring-http-client-config.xml*. Dado que o projeto não conta com um módulo de autenticação, é feita o armazenamento direto da chave de um usuário no *bean* que implementa a classe UserStore fornecido pelo contexto do Spring (o *bean* de ID *beanUserStore*).

A seguir um objeto que implementa a interface IVendasService é instanciado através da requisição do *bean vendasProxyService* ao contexto do Spring. Como o *bean vendasProxyService* faz uso da classe ServiceProxy, um proxy para acesso remoto ao *bean* é criado, e dada a implementação da classe (que pode ser vista com maiores detalhes na seção 4.1.2), quando uma requisição for feita ao *bean* remoto, uma chave de usuário é injetada na requisição remota.

```java
public class HttpClient{

    public static void main(final String[] arguments){
       String url = "br/uniriotec/np2tec/propid/spring/cliente/spring-http-client-config.xml";
       ApplicationContext context = new ClassPathXmlApplicationContext(url);

       IUserStore userStore = (IUserStore) context.getBean("beanUserStore");
       userStore.setChaveUsuario("Y5QL");

       IVendasService vendasService = (IVendasService) context.getBean("vendasProxyService");
        try {
            System.out.println(vendasService.consultarPrevisaoMudancaReceita());
        } catch (SQLException ex) {
            System.out.println("EXCEÇÃO: " + ex.getMessage());
        }
   }

}
```

Figura 35 - Implementação da classe de execução do projeto Cliente

O método *consultarPrevisaoMudanca()* é invocado sem que qualquer parâmetro seja passado, entretanto, como o *bean beanUserStore* possui um valor armazenado, a classe ServiceProxy o utilizará e anexará dinamicamente à requisição um novo parâmetro contendo a chave do usuário. Do lado servidor o parâmetro será recuperado para que o banco de dados tenha ciência de quem é o usuário realizando a consulta.

5 Testes Realizados

Para avaliar a arquitetura proposta, um protótipo foi criado através da tecnologia Java. Foi utilizado o esquema e os dados do *benchmark* TPC-H. O TPC-H é uma especificação de *benchmark* com uma ampla relevância na indústria, que simula o cenário de uma aplicação de apoio à decisão (TPC COUNCIL, 2008). Esse cenário possui um contexto realista que representa as atividades de uma indústria de atacado que precisa gerenciar as suas vendas ou distribuir um produto em todo o mundo (por exemplo, aluguel de veículos, distribuição de alimentos, etc.). O *benchmark* consiste da descrição de um esquema de banco de dados e de um conjunto de consultas *ad-hoc* OLAP (Online Analytical Processing) orientadas ao negócio. A consulta utilizada no teste foi a de previsão de mudança na receita, a qual quantifica o aumento da receita resultante da eliminação de algum desconto percentual em determinado ano (Figura 36).

```
select sum(l_extendedprice * l_discount) as revenue
from lineitem
where l_shipdate >= date '1994-01-01'
  and l_shipdate < date '1994-01-01' + interval '1' year
  and l_discount between 0.06 - 0.01 and 0.06 + 0.01
  and l_quantity < 24;
```

Figura 36 - Previsão de mudança na receita

Foi definida a seguinte regra de autorização que foi cadastrada no modelo do FARBAC: "O gerente de vendas do norte da América e da Ásia deve ter acesso somente aos pedidos provenientes de clientes das nações do hemisfério Norte das regiões Ásia e América". A implementação desta regra necessitou de algumas adaptações no modelo do TPC-H e no FARBAC, tais como: a chave do usuário na tabela *CUSTORMER* do TPCH referenciando a chave do usuário do FARBAC; e, inclusão de campo para indicar o hemisfério da nação.

Para implementar essa regra de autorização foi definida uma hierarquia de perfis, com um perfil pai chamado *Gerente de Vendas* e um perfil filho chamado *Gerente de Vendas Norte América e Ásia*. Considerando a consulta, as informações a serem protegidas por essa política referem-se às tabelas *ORDERS* e *LINEITEM*, tabelas de pedidos e de itens de pedido, respectivamente. As informações foram cadastradas no FARBAC e a função de autorização retorna os predicados referentes à Figura 37 e à Figura 38 para cada uma das tabelas. O predicado correspondente à tabela ORDERS (Figura 37) verifica se o cliente do pedido corresponde a um cliente que está na região América ou Ásia e no hemisfério norte. Já o predicado para a tabela LINEITEM faz acesso à tabela ORDERS. Ao fazer o acesso a esta tabela, a política correspondente é executada e, consequentemente, são retornados apenas os itens das regiões América e Ásia e do hemisfério norte.

```
(O_CUSTKEY in (
select C_CUSTKEY
from TPCH.CUSTOMER
inner join TPCH.NATION on N_NATIONKEY = C_NATIONKEY
inner join TPCH.REGION on R_REGIONKEY = N_REGIONKEY
where R_NAME IN ('AMERICA' , 'ASIA') AND N_HEMISPHERE IN
('NORTH')))
```

Figura 37 – Predicado para a tabela ORDERS

```
(L_ORDERKEY in (
select O_ORDERKEY
from TPCH.ORDERS))
```

Figura 38 – Predicado para a tabela LINEITEM

Foram criados os usuários: *Bob* como gerente de vendas norte da América e da Ásia; e *Alice*, que representando a presidente da empresa e ela recebeu o privilégio *EXEMPT ACCESS POLICY*, que faz com que todas as políticas sejam ignoradas.

Como *Bob* é o gerente de vendas norte da América e da Ásia, ele tem acesso apenas aos itens de pedido do norte dessas regiões, enquanto *Alice*, que é a presidente, tem acesso a todos os itens de pedido. Para *Alice*, a mudança na receita foi de R$ *88.894.386,60*. Já para *Bob* esta é de apenas R$ *21.445.915,50*, afinal a sua visão é somente dos itens de pedido da América e da Ásia.

Em seguida, a Aplicação Cliente foi executada múltiplas vezes (seis instâncias foram inicializadas simultaneamente) para invocar a consulta apresentada na Figura 36, com os usuários *Bob*, *Alice* e outros 4 usuários sem perfil atribuído. O módulo servidor foi editado para que seu processamento fosse retardado (foi utilizado o método *Thread.sleep()*) e as seis requisições fossem executadas simultaneamente, possibilitando testar a concorrência entre múltiplas requisições ao servidor.

```
/*************************************************************************
 ** Trecho de código usado no teste. Desencessário para a arquitetura.   **
 *************************************************************************/
    //Gerar ID randomico para a Thread
    int threadID = generateID();
    //Printar o ID da thread e a chave do usuário armazenado
    printStatusConsole(threadID, s, "PreProc");
    //Parada para testar concorrencia na sessão
    Thread.sleep(15000);
    //Printar o ID da thread e a chave do usuário armazenado
    printStatusConsole(threadID, s, "PosProc");
/*************************************************************************
 **                          Fim do código de teste                      **
 *************************************************************************/
```

Figura 39 - Código verificar isolamento entre as requisições

A Figura 39 mostra o trecho de código inserido no método *consultarPrevisaoMudancaReceita()* (descrito originalmente na Figura 29) para verificar o isolamento entre múltiplas requisições recebidas simultaneamente pelo servidor de aplicação. Logo após a obtenção de uma sessão previamente identificada por meio do método *openSession()* da classe SessionFactory, o método imprime no console do servidor o ID da thread (valor gerado de forma randômica simplesmente para efeitos de teste pelo método *generateID()*), o usuário que se encontra armazenado no *beanUserStore* (*bean* com escopo de requisição responsável por armazenar os dados do usuário) e o usuário que está identificado no atributo CLIENT_INFO da sessão com o banco de dados. Estes dois últimos dados descritos são recuperados através dos métodos *retrieveUserStore()* e *retrieveClienteInfo()*, respectivamente. A escrita destes dados no console do servidor é realizada pelo método *printStatusConsole()*, descrito na Figura 40.

```
private boolean printStatusConsole(int threadID, Session s, String tipo){
    System.out.println(tipo+" -> Thread ID: " + threadID +
        " / UserKey: " + retrieveUserStore().getChaveUsuario() +
        " / Session_Client_Info: " + retrieveClientInfo(s));
    return true;
}
```

Figura 40 - Método *printStatusConsole()* para exibir no console do servidor os dados

da requisição

Logo após, o método sofre uma paralisação de 15 segundos, tempo que permitiria que um dos outros 5 processos alterasse as informações de usuários no *beanUserStore* e no atributo CLIENT_INFO, caso houvesse falha no isolamento dos dados da requisição ou cruzamento de informações de qualquer tipo. Uma vez terminado o processamento do comando que paralisou a requisição, os dados do usuário (*beanUserStore* e CLIENT_INFO) são impressos mais uma vez no console do servidor de aplicação. Em nenhum caso, o usuário de uma instância empregado em uma requisição foi sobrescrito pelo usuário de outra requisição, como é possível observar no resumo do LOG do servidor de aplicação (Figura 41), evidenciando que mesmo após a instanciação dos 6 processos os dados do usuário em cada processo permaneceram inalterados.

```
15:35:28,850 PreProc -> Thread ID: 909305931 / UserKey: W2R9 / Session_Client_Info: W2R9
15:35:29,630 PreProc -> Thread ID: 981275526 / UserKey: D6G4 / Session_Client_Info: D6G4
15:35:30,753 PreProc -> Thread ID: 382661120 / UserKey: P9U0 / Session_Client_Info: P9U0
15:35:31,548 PreProc -> Thread ID: 533394192 / UserKey: J5Y8 / Session_Client_Info: J5Y8
15:35:32,796 PreProc -> Thread ID: 209032620 / UserKey: Y2R7 / Session_Client_Info: Y2R7
15:35:33,842 PreProc -> Thread ID: 973668320 / UserKey: Y2T0 / Session_Client_Info: Y2T0
                        /** Processamento omitido */
15:35:43,888 PosProc -> Thread ID: 909305931 / UserKey: W2R9 / Session_Client_Info: W2R9
15:35:44,668 PosProc -> Thread ID: 981275526 / UserKey: D6G4 / Session_Client_Info: D6G4
15:35:45,791 PosProc -> Thread ID: 382661120 / UserKey: P9U0 / Session_Client_Info: P9U0
15:35:46,587 PosProc -> Thread ID: 533394192 / UserKey: J5Y8 / Session_Client_Info: J5Y8
15:35:47,882 PosProc -> Thread ID: 209032620 / UserKey: Y2R7 / Session_Client_Info: Y2R7
15:35:48,880 PosProc -> Thread ID: 973668320 / UserKey: Y2T0 / Session_Client_Info: Y2T0
```

Figura 41 - Saída do processamento das requisições no Console do Servidor

Os resultados obtidos no teste de concorrência podem ser considerados para situações onde o método que realiza o acesso ao banco de dados faz uso de transações, sejam elas declaradas pelo próprio Spring (através da anotação *@Transactional*), utilizadas através do *framework* Hibernate (por meio dos métodos *beginTransaction()* e *commit()*) ou criadas diretamente via JDBC. Os motivos para considerar os resultados extensíveis a cenários que requerem transação são explicados adiante.

A Figura 42 ilustra um cenário típico onde o uso de transações se faz necessário. O método *inserirPedido* recebe como parâmetro um objeto do tipo *Pedido*, composto por uma lista de objetos *ItemPedido*, além de outras informações como: o atendente que registrou o pedido, data e hora de sua criação, loja na qual foi realizado, etc. O método requisita ao contexto da aplicação o objeto *SessionFactory* para poder recuperar uma sessão com o banco de dados e armazenar os objetos. O método tem o objetivo de salvar o pedido e depois, em uma tabela à parte do banco, relacionar quais itens estavam atrelados àquele pedido. Este tipo de operação requer que no advento de uma falha, toda e qualquer alteração no estado do banco de dados realizado pelo método seja desfeito, por exemplo, caso ocorra falha na inserção de um item do pedido, a inserção do pedido em si deve ser desfeita.

```
@Transactional
public boolean inserirPedido(Pedido p) throws Exception{
    ApplicationContext ctx = ApplicationContextProvider.getApplicationContext();
    sessionFactory = (SessionFactory) ctx.getBean("beanSessionFactory");
    Session s = sessionFactory.openSession();

    Transaction tx = null;

    try{
        //Inserir Pedido
        tx = s.beginTransaction();
        s.save(p);
        tx.commit();

        tx = s.beginTransaction();
        //Inserir Itens do pedido
        for(ItemPedido ip : p.getItensPedido()){
            s.save(ip);
        }
        tx.commit();
    }catch(RuntimeException ex){
        throw ex;
    }

    //Fechar Sessão
    s.close();


    return true;
}
```

Figura 42 - Exemplo de Uso da Anotação *@Transactional*

Neste caso em especial, devido ao uso do *framework* Hibernate, a inserção de um objeto deve ser feita por meio do método *save* em uma transação criada pelo próprio Hibernate. Para fins de demonstração separamos a inserção do objeto *Pedido* da inserção dos objetos *ItemPedido* (é possível delegar ao Hibernate a tarefa de salvar um objeto complexo sem ter de destrinchá-lo em várias inserções), além de criar uma transação Hibernate para cada objeto sendo salvo no banco. Apesar de cada objeto ser salvo isoladamente em um acesso ao banco, a anotação *@Transactional*, inserida na linha 1, informa ao *framework* Spring tudo que for executado no método deve obedecer ao padrão de transação, ou seja, no advento de múltiplas operações com o banco de dados o *framework* deverá garantir que caso uma destas operações ser mal-sucedida, aquelas que ocorreram de forma correta devem ser desfeitas.

O suporte a transações é de fato implementado por classes auxiliares como a `DataSourceTransactionManager`, a `JtaTransactionManager` ou a `HibernateTransactionManager`, todas providas pelo *framework*. Esta classe auxiliar deve ser disponibilizada por meio da configuração de um bean no arquivo de configuração utilizado para inicializar o *framework* (o mesmo arquivo onde são configurados *beans* como o *beanUserStore* e o *beanSessionFactory*). A Figura 43 ilustra a configuração do *bean "txManager"* para auxiliar no suporte transacional e adiciona a configuração *tx:advice*, informando ao *framework* qual *bean* deverá ser utilizado quando o suporte a transações for necessário.

```
<bean id="txManager"
      class="org.springframework.transaction.jta.JtaTransactionManager"/>

<tx:advice id="txAdvice" transaction-manager="txManager">
```

Figura 43 - Configuração de um *bean* para auxiliar no suporte transacional

Como apresentando anteriormente, os resultados obtidos no teste de concorrência realizado são extensíveis a um cenário onde se faz uso de transações. A razão para considerar que tais resultados são aplicáveis se encontra na consideração de que independente do método transacional escolhido ou mesmo do não uso de transações uma única sessão com o banco de dados será criada quando o método se inicia (neste caso, na linha 5 do método *inserirPedido*) e encerrada ao fim deste método. Mesmo que esta sessão seja fechada, tanto o *beanSessionFactory* quanto o

beanUserStore são configurados com escopo de requisição (como visto anteriormente), garantindo que caso outra sessão precise ser criada, o usuário a ser ajustado no contexto desta sessão pela classe SessionFactory será o mesmo utilizado na criação da primeira sessão.

6 Conclusões

Este trabalho apresentou uma proposta de solução para propagação de identidade em três camadas, mais especificamente para o cenário "cliente acessa servidor de aplicação que acessa o banco de dados", descritos por Azevedo *et al.* [2009] e Leão *et al.* [2011b], empregando o *framework* Spring. O *framework* Hibernate é utilizado para acessar o banco de dados e propagar a identidade do usuário até o banco de dados.

Uma premissa para esta solução é que o Spring esteja sendo utilizado para disponibilizar o acesso remoto a beans executando no servidor de aplicação. Sendo esta premissa atendida, a solução proposta leva a um impacto mínimo nas aplicações clientes e nos beans implementados. Isto ocorre pelo uso das classes HttpInvokerServiceExporter e HttpInvokerProxyFactoryBean, ambas nativas do *framework* Spring e responsáveis por possibilitar, respectivamente, a exposição de *beans* para acesso remoto e o consumo de *beans* expostos desta forma. Estas classes foram evoluídas para permitir a propagação de identidade.

Na aplicação cliente, é necessário configurar um bean para armazenamento das informações do usuário, o qual deve ser instanciado assim que o usuário logar no sistema. A partir daí, todas as invocações de *beans* remotos serão interceptadas pelo ServiceProxy o qual irá anexar as informações do usuário à URL de requisição. Já no servidor, um bean de escopo de requisição deve ser configurado para armazenar as informações do usuário. Toda requisição de bean remoto é interceptada pelo ServiceExporter, o qual retira da URL de invocação as informações do usuário e instancia o bean que irá armazenar as informações do usuário. A toda abertura de conexão, o Hibernate obtém as informações do usuário do bean e as ajusta na conexão, propagando os dados até o SGBD.

No SGBD, o *framework* FARBAC [Azevedo *et al.*, 2010] é utilizado para executar as regras de autorização de acesso aos dados. Aplicando as regras de controle de acesso e fazendo com que o usuário receba apenas os dados que ele tem acesso.

Foi desenvolvido um protótipo para avaliação experimental da proposta, considerando o *benchmark* TPC-H. Os resultados comprovaram a possibilidade de implementar alterações não intrusivas que possibilitam a propagação da identidade do usuário desde o módulo cliente, passando pelo módulo servidor até o banco de dados sem que seja necessário alterar assinatura de métodos ou refatorar a aplicação para que esta armazene o estado da conexão entre módulos cliente e servidor. Além disso, foi demonstrado via testes experimentais que a execução de uma requisição ocorre de forma isolada e que transações são executadas normalmente, considerando o usuário logado na aplicação cliente na execução das operações de acesso ao banco de dados.

A funcionalidade *Remote Access* do *framework* Spring pode ainda ser utilizado para expor Web Services, desta forma propõe-se como trabalho futuro a pesquisa sobre alterações similares as da proposta deste trabalho que possibilitem a propagação da identidade atendendo ao quarto cenário descrito por Azevedo *et al.* [2009] e Leão *et. al.* [2011a].

Referências Bibliográficas

AZEVEDO, L.; DUARTE, D.; PUNTAR, S.; ROMEIRO, C.; BAIAO, F.; CAPPELLI, C. **Avaliação de Ferramentas para Gestão e Execução de Regras de Autorização.** Relatórios Técnicos do DIA/UNIRIO (RelaTe-DIA), RT-0027/2009, 2009. Disponível (também) em: http://seer.unirio.br/index.php/monografiasppgi

AZEVEDO, L. G., PUNTAR, S., THIAGO, R., BAIÃO, F., CAPPELLI, C. **A Flexible Framework for Applying Data Access Authorization Business Rules.** In: 12th International Conference on Enterprise Information Systems, pp. 275-280, 2010.

BAUER, C., KING, G., **Hibernate in Action**, Manning 2005.

BURKE, B. e MONSON-HAEFEL, R. **Enterprise JavaBeans, 3.0.** O'Reilly, 2006, 760p. Bibliografia: ISBN-10: 0-596-00978-X.

JOHNSON, R., HOELLER, J., DONALD, K. et al., **Spring Framework Reference Documentation (3.0)**, 2010. Disponível em http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/pdf/spring-framework-reference.pdf. Acessado em Maio de 2011.

LEÃO, F., PUNTAR, S., AZEVEDO, L. G., CAPPELLI, C., BAIÃO, F. **Propagação de Identidade em Aplicações em Três Camadas**. Relatórios Técnicos do DIA/UNIRIO (RelaTe-DIA), RT-0007/2011, 2011a. Disponível em <http://www.seer.unirio.br/index.php/monografiasppgi>. Acessado em Agosto de 2012.

LEÃO, F., PUNTAR, S., AZEVEDO, L. G., CAPPELLI, C., BAIÃO, F., **Controle de Acesso a Dados em Sistemas de Informação através de Mecanismos de Propagação de Identidade e Execução de Regras de Autorização.** In: VII Simpósio Brasileiro de Sistemas de Informação (SBSI 2011), Salvador, Bahia, 2011b.

PRASANNA, D.R. **Dependency Injection: Design Patterns Using Spring and Guice.** Manning, Greenwich, CT, 2009.

WALLS, C. BREIDENBACH, R. **Spring in Action.** Manning, 2007.

Apêndice 1 Acesso a um EJB 3.0 com Spring

É importante observar que o uso do Spring não impede o uso do EJB, na verdade, o framework é capaz de até mesmo facilitar o acesso e a implementação de *Enterprise Java Beans*. O uso do Spring no acesso a EJBs permite, por exemplo, que a troca entre EJBs locais, remotos ou variantes POJO sejam feitas sem que seja necessário alterar o código cliente, dado que sua configuração ocorre externa ao código.

Assim como em um cliente que acessa diretamente a JNDI[15] onde se encontra publicado um EJB, o uso do Spring requer configuração prévia do acesso a JNDI responsável pelo EJB. Inicialmente considere um EJB *Stateless*, implementado com EJB 3, chamado `EJBOlaMundo` (Figura 44). O EJB possui um único método "`dizerOi()`" que retorna uma String. A classe implementa a interface `EJBOlaMundoRemote`, cuja única responsabilidade é expor o método `dizerOi()`.

```
1  package br.uniriotec.propid.spring.olamundo;
2  import javax.ejb.Stateless;
3
4  @Stateless
5  public class EJBOlaMundo implements EJBOlaMundoRemote {
       @Override
7      public String dizerOi() {
8          return "Olá Mundo";
9      }
10 }
```

Figura 44 - Implementação do EJBOlaMundo

No lado cliente, considere uma aplicação que faz uso do *framework* Spring e cujo objetivo é recuperar um *bean* que implemente um método. Para a aplicação, não importa a localização deste *bean*, seja ele local ou remoto (seja o servidor de aplicação JBoss, GlassFish, WebSphere etc.), Spring ou EJB.

A Figura 45 mostra o trecho de configuração que deve ser feito em um arquivo de configuração do Spring para que o framework consiga acesso à JNDI de um servidor, por exemplo, JBoss e mais especificamente a um EJB disponibilizado por ele.

[15] Java Naming and Directory Interface, API Java para serviços de diretórios que permite a descoberta e acesso a objetos remotos.

```
<!-- CONFIGURAÇÃO JNDI -->
<bean id="jndiConfig" class="org.springframework.jndi.JndiTemplate">
  <property name="environment">
    <props>
      <prop key="java.naming.factory.initial">org.jnp.interfaces.NamingContextFactory</prop>
      <prop key="java.naming.provider.url">jnp://servidor:1099</prop>
      <prop key="java.naming.factory.url.pkgs">org.jboss.naming:org.jnp.interfaces</prop>
    </props>
  </property>
</bean>

<bean id="ejb" class="org.springframework.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean"
    lazy-init="true">
  <property name="jndiName" value="EJBOlaMundo/remote" />
  <property name="businessInterface" value="br.uniriotec.propid.spring.olamundo.EJBOlaMundoRemote" />
  <property name="jndiTemplate">
    <ref local="jndiConfig" />
  </property>
</bean>
```

Figura 45 - Configuração do SpringXMLConfig.xml para acesso a um EJB remoto

Como mostrado na Figura 45, é necessário que o *bean* responsável por referenciar o EJB, neste caso identificado pelo atributo `id="ejb"`, implemente a classe `SimpleRemoteStatelessSessionProxyFactoryBean`, do pacote `org.springframework.ejb.access`. O *bean* deve possuir uma propriedade que informe o nome do EJB na JNDI, neste caso `EJBOlaMundo` (o "/remote" é uma especificidade do servidor de aplicação JBoss), sendo uma propriedade que informa a interface de negócio do EJB, incluindo seu pacote. Uma terceira propriedade é necessária para informar os dados da JNDI onde se encontra o EJB. Estes dados podem ser isolados em um *bean* da classe `JndiTemplate`, pertencente ao pacote `org.springframework.jndi`. Na Figura 45, o JndiTemplate é recuperado através do *bean* com id="jndiConfig".

A Figura 46 mostra a implementação de uma classe "Main" que faz uso do *bean* "ejb", invocando seu método "`dizerOi()`".

```
1  package br.uniriotec.propid.spring.cliente;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  public class Main {
7      public static void main(String[] args) {
8          ApplicationContext context =
9                  new ClassPathXmlApplicationContext("SpringXMLConfig.xml");
10         EJBDIRemote ejb = (EJBDIRemote) context.getBean("ejb");
11         System.out.println(ejb.dizerOi());
12     }
13
14 }
```

Figura 46 - Programa cliente invocando o EJB

Primeiramente, o programa cria um contexto a partir do arquivo de configuração do Spring (SpringXMLConfig.xml), utilizando seu método "`getBean()`" para recuperar um *bean* descrito no arquivo a partir do seu "id", para então utilizar um de seus métodos (no caso o único método disponível, "`dizerOi()`"). Como mostrado, é transparente para a aplicação cliente o local do *bean* "ejb", inclusive qual a tecnologia do servidor de aplicação que o disponibiliza e se ele é implementado através da tecnologia EJB 3 ou Spring.

Apêndice 2 Criação de Enterprise Java Beans com Spring

Sendo um container "leve", o Spring é frequentemente considerado como um substituto ao EJB. A combinação do container do framework as suas funcionalidades transacionais podem ser usadas em substituição a EJBs e seus respectivos containers [Johnson *et. al.*, 2010].

A implementação de um EJB através do Spring se dá através da atribuição de uma classe pai específica à classe que se deseja expor como um EJB. Em outras palavras, a classe destinada a se tornar um EJB deverá herdar uma de três classes abstratas possíveis. A classe a ser implementada dependerá do tipo de EJB que se deseja criar (*Stateless*, *Stateful* ou *MessageDriven*). A herança cuidará da implementação de todas as interfaces responsáveis pelo ciclo de vida de um EJB no container Spring e acarretará a implementação de no mínimo um método, responsável por auxiliar na instanciação do EJB (o método "`onEjbCreate()`").

As três interfaces disponíveis são:

- **AbstractStatelessSessionBean** – Utilizada ao implementar um EJB Stateless[16].
- **AbstractStatefulSessionBean** – Utilizada ao implementar um EJB Stateful[17].
- **AbstractMessageDrivenBean/AbstractJmsMessageDrivenBean** – Utilizada ao implementar um Message Driven bean.

É recomendável ainda que o uso de uma destas classes abstratas venha acompanhado da prática em isolar a lógica do negócio em um *bean* POJO a ser utilizado pelo EJB. Quando o EJB for criado, ele deverá instanciar este POJO e implementar métodos que façam o trabalho de repassar a invocação ao método correspondente no POJO. Esta é considerada uma boa prática de programação que tende a auxiliar na realização de testes da aplicação.

É importante ainda observar que o EJB deverá implementar a mesma interface que o POJO citado, garantindo que todos os métodos sejam implementados e expondo aos consumidores a sua interface de negócio.

O Spring carregará automaticamente uma BeanFactory baseada em um documento XML, o arquivo `ejb-jar.xml`. Esta Bean Factory se encarregará de expor o EJB criado através da herança de uma das três classes citadas. Para informar ao Spring a localização do arquivo XML, a variável de ambiente `ejb/BeanFactoryPath` deverá ser ajustada no descritor de implantação padrão do EJB (EJB *deployment descriptor*), o arquivo `ejb-jar.xml`.

Maiores detalhes sobre a implementação de *Enterprise Java Beans* através do framework Spring podem ser vistas na seção 20.3 da documentação oficial do Spring provida pela Comunidade Spring através do site SpringSource[18] [Johnson et al., 2010].

---

[16] A nomenclatura *Stateless* define um módulo que não armazena o estado da aplicação, ou seja, o ciclo de vida de uma requisição inicia e termina junto ao método invocado no servidor.

[17] A nomenclatura *Stateful* define um módulo capaz de armazenar o estado de uma aplicação. Um exemplo comum é o "carrinho de compras", onde durante todo o tempo determinadas informações como produtos selecionados, se encontrem armazenadas.

[18] http://www.springsource.org/documentation