Ipiranga Produto de Petróleo

PDSI – Padrão de Desenv. de Software Ipiranga

Framework JAVA

Versão 1.5.0

Histórico de Revisões

DATA	VERSÃO	DESCRIÇÃO	AUTOR
05/01/2014	1.0	Criação do Documento	PrimeUp / Ipiranga
09/01/2014	1.0.1	Revisão do documento	Adolpho / Igor
24/03/2014	1.2.0	Conceitos de WebServices	Adolpho Alves
05/09/2014	1.3.0	Tutorias de integração com SAA	Adolpho Alves
14/08/2017	1.4.0	Inclusão do SPA	PrimeUp/Ipiranga
11/01/2018	1.5.0	Testes com Mockito	PrimeUp/Ipiranga

Índice

FrameWork Java

1 Introdução

Este documento visa padronizar o desenvolvimento de aplicações Java na Ipiranga Produtos de Petróleo. As informações apresentadas aqui representam um padrão que deve ser seguido. Esse procedimento visa facilitar a comunicação entre os desenvolvedores da Ipiranga e empresas terceirizadas, facilitando futuras manutenções nos sistemas.

O padrão apresenta uma visão geral da arquitetura e abrange camadas, tecnologias, padrões de projeto e padrões arquiteturais. As seções e subseções do documento de arquitetura de software apresentam diferentes visões da arquitetura, com suas camadas e componentes, seus objetivos e restrições, organização dos componentes em pacotes e os padrões de projetos e arquiteturais utilizados.

As definições contidas nesse documento, por sua vez, não influenciam o modelo arquitetural lógico ou físico das bases de dados, tais quais: definições de entidades e tabelas, atributos, restrições, procedures, triggers ou visões.

2 Visão Lógica da Arquitetura

A visualização lógica da arquitetura é composta por quatro pacotes principais: (i) visão; (ii) serviço; (iii) domínio; e (iv) dados. Cada um dos principais pacotes citados representa uma camada da arquitetura proposta. A Figura 1 apresenta a divisão lógica da arquitetura em camadas.

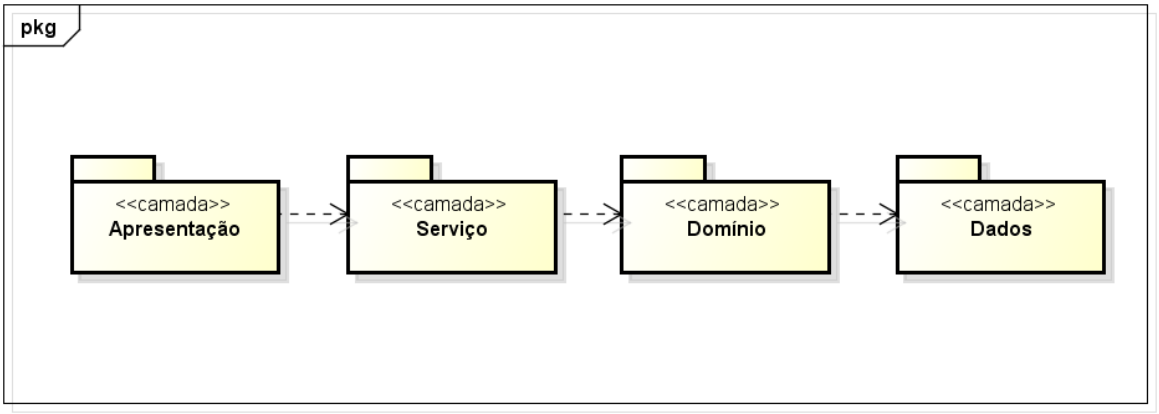


Figura 1 - Visão Lógica da Arquitetura

2.1 Definição das Camadas

Com intuito de organizar, facilitar o desenvolvimento e a manutenção dos sistemas, adotou-se como base, conforme proposto em (Evans, 2003), a divisão dos mesmos em quatro camadas: (i) Visão; (ii) Serviço; (iii) Domínio; e (iv) Dados. A seguir serão descritas as responsabilidades de cada camada presente na arquitetura.

i. Visão – Também conhecida como Apresentação, essa camada é responsável por exibir informações para o usuário e interpretar os comandos do usuário. O usuário externo pode ser um sistema de computador ou um usuário humano.

ii. Serviço - Possui a responsabilidade de coordenar as atividades da aplicação. É uma fina camada e não contem lógica de negócio. Não armazena estado dos objetos de negócio, mas pode armazenar o estado das diversas tarefas em progresso da aplicação.

iii. Domínio - Possui a responsabilidade de executar a lógica do negócio e manipulação dos dados das fontes de dados. O estado dos objetos de negócio é armazenado aqui. Persistência dos objetos de negócio e possivelmente seus estados são delegados para a camada de dados.

iv. Dados - Possui a responsabilidade de oferecer suporte para todas as outras camadas. Provê comunicação entre camadas, implementa persistência para objetos de negócio, contem bibliotecas de suporte para a aplicação, etc.

2.2 Detalhamento das Camadas

Este tópico tem o objetivo de fazer um aprofundamento das características de cada camada. Neste aprofundamento, são apresentados exemplos de código, fundamentação teórica, regras e recomendações para o desenvolvimento das camadas.

2.2.1 Camada de Domínio

Fowler, em (Fowler, 2002), define três padrões para a camada de domínio. Dentro destes padrões, a abordagem mais robusta é o uso do padrão *Domain Model*. Um *Domain Model* é essencialmente um modelo de domínio que mapeia toda a lógica de negócio do sistema através de classes que representam entidades da aplicação. Define e gerencia todos os possíveis estados dos objetos de domínio.

Este padrão, assim como qualquer outro, possui vantagens e desvantagens que serão descritas a seguir:

Vantagens:

- É um modelo capaz de trabalhar com lógicas de negócio muito complexas.
- Mapeia a lógica de negócio através de objetos interconectados, centralizando o comportamento da aplicação no domínio.
- Prove fácil manutenção e rápida alteração da lógica de negócio. Evita duplicação de regras de negócio.
- Permite carregar e trabalhar com os objetos (e regras de negócio) em memória.

Desvantagens:

- Pode ser necessária a criação de muitas classes se a aplicação for muito complexa.
- Pode ser mais difícil fazer o mapeamento com o banco de dados.
- Pode ser difícil identificar e mapear as responsabilidades exclusivas de cada entidade.

Levando em consideração que os sistemas têm um domínio e lógicas de negócios complexas, deve ser utilizado o padrão *Domain Model*. Tal padrão proporcionará uma maior organização da lógica de negócio dos sistemas.

Para fins de organização e uma melhor estruturação do projeto, um pacote específico para a camada de domínio deverá ser criado. Todos os componentes que pertencem á camada de domínio devem ficar centralizados neste mesmo pacote. Sugere-se que tal pacote tenha o nome domínio (dominio), desta forma se torna claro para o desenvolvedor o papel dos componentes e explicita a divisão em camadas.

Os componentes do domínio não devem se falar com componentes da camada de visão, da camada de dados e da camada de serviços. Além de melhorar a organização, tal restrição diminui a dependência entre camadas evitando uma referência cíclica já que, por exemplo, a camada de serviços conhece e utiliza os componentes da camada de domínio. Dessa forma, os componentes da camada de domínio serão manipulados através da camada de serviços e persistidos através da camada de dados.

Para guiar o desenvolvimento da camada de domínio, os sistemas da Ipiranga devem ser desenvolvidos usando a abordagem *Domain Driven Development* (DDD). Essa é uma abordagem solida orientada ao domínio, que propõe o desenvolvimento de um software harmonioso com o domínio. Esse modelo também orienta que a construção do software, tanto o código quanto o sistema em si, devem ser um reflexo de seu domínio.

Segundo Eric Evans (Evans, 2003), DDD é uma abordagem para desenvolver softwares complexos baseando-se integralmente no domínio e criando uma conexão direta e mutua entre o modelo de domínio e a implementação do sistema.

DDD combina design e práticas de desenvolvimento e demonstra como, o design e o desenvolvimento, podem trabalhar em conjunto para elaboração de uma solução ótima. Um bom design acelera o desenvolvimento, enquanto que o feedback obtido durante do desenvolvimento, irá melhorar o design. Além disso, ressalta a importância da comunicação e do modelo em um projeto.

2.2.1.1 Modelo DDD

DDD propõe o uso de uma arquitetura em camadas para isolar o domínio dos demais componentes envolvidos para o correto funcionamento de um sistema. O desenvolvimento do design em cada camada deve ser coeso e depender somente das suas camadas inferiores, seguindo padrões arquiteturais para prover baixo acoplamento para as camadas superiores. Todo código relacionado com o modelo de domínio deve estar concentrado na camada de domínio, isolada das demais. Esta camada deve ser responsável por mapear todas as regras de negócio.

As camadas podem se comunicar entre si de diversas formas, algumas vezes através da definição de interfaces contidas nas camadas inferiores, que são manipuladas em camadas superiores, ou através de call-back recebidos pelas camadas inferiores.

A Figura 2 exemplifica os componentes utilizando a abordagem DDD. Os principais componentes são apresentados nas seções a seguir. Estes conceitos devem ser levados em consideração ao codificar um novo software de modo que ele seja compatível com o domínio e confeccionado seguindo boas técnicas de design, aumentando sua manutenibilidade e diminuindo seu acoplamento entre componentes.

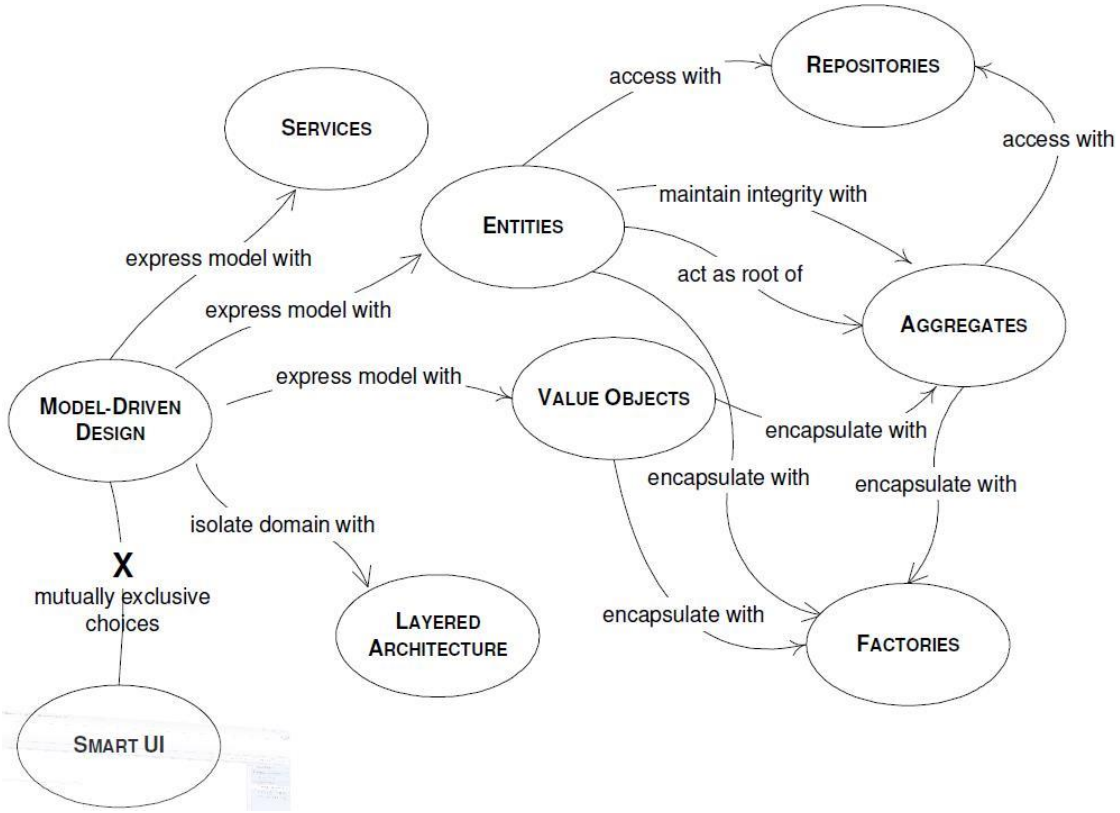


Figura 2 - Principais componentes do DDD

2.2.1.1.1 Entidades (Entities)

Uma entidade é um objeto do domínio que possui uma identidade única. Por exemplo, uma pessoa física que é identificada unicamente pelo seu CPF. Cada entidade deve possuir operações que produzam um resultado único para cada um dos seus objetos. A entidade deve ser simples e sua construção deve estar focada na continuidade do ciclo de vida e na sua identidade. Não se deve fazer com que toda classe seja uma entidade, somente os objetos que tenham uma identidade única devem ser considerados como entidades. Objetos que são utilizados apenas para descrever aspectos do domínio e que não possuem identidade devem ser *Value Objects*.

```
01 public class ZonaVenda {
02     /**
03      * Código da Zona de Venda.
04      */
05     private String cdZonven;
06
07     /**
08      * Gerencia de Venda responsavel a essa instância de Zona de Venda
09      */
10     private GerenciaVenda gerenciaVenda;
11
12     /**
13      * Nome da Zona de Venda.
14      */
15     private String nmZonven;
16
17     /**
18      * Código da coordenadoria de venda.
19      */
20     private String cdCoordVnd;
21
22     /**
23      * Código da empresa do funcionário que é responsável pela zona de
venda.
24      */
25     private String cdEmprFunc;
26
27    ...
28 }
```

Código 1 - Exemplo de Entidade

2.2.1.1.2 Value Objects

Um *Value Object* deve ser imutável e não deve possuir identidade. São criados pelo seu construtor e não são modificados durante seu tempo de vida. Quando se deseja um novo valor para o objeto, simplesmente cria-se um novo através do construtor. Objetos imutáveis são compartilhados eficientemente (melhor desempenho) e mantem a integridade dos dados. Eles devem ser pequenos, simples e representar um papel conceitual, formado a partir da união de atributos relacionados semanticamente entre si. Estes objetos podem conter outros *Value Objects* e até mesmo entidades. Porém, o uso de entidades dentro de *Value Object*s deve ser moderado e cauteloso já que, diferentemente de um *Value Object*, uma entidade pode ser alterada.

```
01 public class AuditoriaVo {
02     /**
03      * Data de Inclusão
04      */
05     private Date dtIncl;
06
07     /**
08      * Data de Alteração
09      */
10     private Date dtAlter;
11
12     /**
13      * Data de Inativação
14      */
15     private Date dtInat;
16
17     /**
18      * Código do Usuário
19      */
20     private String cdUsuario;
21   ...
22 }
```

Código 2 - Exemplo de Value Object

2.2.1.1.3 Serviços de Domínio (Domain Services)

Ao elaborar o modelo de domínio, são identificados substantivos que correspondem a entidades ou *Value Objects*, verbos identificam algumas das ações executadas pelas entidades e propriedades caracterizam o estado de cada objeto. Porém, alguns conceitos parecem que não pertencem a um objeto específico, representam um importante comportamento do domínio e não podem ser incorporados diretamente a uma entidade ou um *Value Object*. Normalmente estes conceitos englobam muitos objetos e até mesmo diferentes classes. Tais conceitos devem ser mapeados como serviços. Serviços são comuns em frameworks técnicos, porém podem ser utilizados, na camada de domínio. Um serviço normalmente se torna um ponto de conexão entre objetos e por este motivo, comportamentos que pertencem a um serviço, não devem ser incorporados a um objeto do domínio. Existem três características principais de um serviço:

1. A operação realizada por um serviço representa um conceito do domínio que não pertence naturalmente a uma entidade ou *Value Object*.
2. A operação realizada se refere a outros objetos no domínio.
3. A operação não possui armazenamento de estado.

Quando um processo importante não pode ser mapeado naturalmente para uma entidade ou *Value Object*, deve-se criar um serviço que o englobe. O nome do serviço e da operação deve fazer parte do modelo e da linguagem ubíqua e não deve possuir estado (*stateless*). É fácil se confundir entre serviços que pertencem à camada de domínio e aqueles pertencentes à camada de infraestrutura. Ambos, aplicação e serviços de domínio são normalmente construídos no topo de entidades do domínio e provêm funcionalidades diretamente relacionadas com os mesmos, sendo a principal diferença o modo de interação com estas entidades. Aplicação utiliza as informações das entidades para fins de apresentação enquanto que os serviços de domínio são utilizados para encapsular a chamada de regras e de mecanismos de persistência e também, pela execução de operações sobre um conjunto de objetos.

```
01 public class ServicosDeCondicaoComercialProposta {
02
03     private ITodasAsCondicoesComerciaisPropostas
todasAsCondicoesComerciaisPropostas;
04     /**
05      * RN08: Não será permitido mais de uma condição comercial
proposta
06      * aprovada, para um mesmo produto, ponto de venda e unidade
operacional
07      *
08      * @throws ExcecaoDoDominio
09      */
10     public void modificarStatusAprovacaoPara(
11             CondicaoComercialProposta condicaoComercialProposta, char
aprovado)
12             throws ExcecaoDoDominio
13     {
14         if (aprovado == 'N')
15         {
16             condicaoComercialProposta.mudarStatusAprovacao('N');
17         } else
18         {
19             verificaSePodeAprovarStatus(condicaoComercialProposta);
20             condicaoComercialProposta.mudarStatusAprovacao('S');
21         }
22     }
23
24 }
```

Código 3 - Exemplo de Domain Service

2.2.1.1.4 Agregadores (Aggregators)

Um agregador é um grupo de objetos associados, que são considerados como um único elemento em consideração a alteração de dados. O agregador é demarcado por uma fronteira que separa os objetos internos dos externos. Cada agregador possui uma raiz que é uma entidade e é a única entidade acessível externamente. Ela pode conter referências para qualquer um dos objetos agregados e os agregados podem conter referencias uns com os outros, mas o objeto externo só pode conter referencia para o objeto raiz.

Com o uso de *Value Objets*, é possível passar referências temporárias dos objetos internos para os externos. Se objetos que pertencem a um agregador são armazenados em um banco de dados, somente a raiz deveria ser acessível através de queries, os demais deveriam ser obtidos através de associações transversais. Os objetos dentro de um agregador podem conter referências para raízes de outros agregadores.

```java
public class GerenciaComercial {
    /**
     * Código da gerência comercial.
     */
    private String cdGc;

    /**
     * Descrição da gerência comercial.
     */
    private String nmGc;

    /**
     * Conjunto de Gerencias de Venda agrupadas
     */
    private Collection<GerenciaVenda> gerenciasVenda;

    /**
     * Campos utilizados para auditoria
     */
    private AuditoriaVo auditGerenciaComercial;
    ...
}
```

Código 4 - Exemplo de Agregador

2.2.1.1.5 Fábricas (Factories)

Outro padrão utilizado na abordagem DDD é o uso de fábricas. Fábricas encapsulam o conhecimento necessário para criação de objetos, e são especialmente úteis para criar agregadores. Quando uma raiz de um agregador é criada, todos os objetos contidos pelo agregador também são criados. É importante que este processo seja atômico.

Um método da fábrica deve ser um método de objeto que contém e encapsula o conhecimento necessário para criar outro objeto. Quando alguma condição de construção é modificada, somente a fábrica responsável por sua criação deverá ser atualizada. Porém, algumas vezes fábricas não são necessárias para criação de objetos.

Não é recomendável a criação de uma fábrica quando:

1. O construtor não é muito complicado.
2. A criação do objeto não envolve a criação de outros e todos os atributos necessários à sua construção são passados por parâmetro.
3. O cliente está diretamente relacionado com a construção dos objetos e quer escolher qual estratégia de criação utilizar.

```
01 /**
02  * Fábrica para criação de entidades da classe Pais {@link Pais}
03  */
04 public class NovoPais {
05 
06     /**
07      * Criação de nova instância de País
08      *
09      * @param cdPais
10      * @param nmPais
11      * @param cdPaisAreaTel
12      * @param dtIncl
13      * @param cdUsuario
14      * @return Nova instância de país
15      * @throws ExcecaoDoDominio
16      * @see {@link Pais}
17      */
18     public static Pais criarNovoPaisAPartirDe(int cdPais, String
nmPais, short cdPaisAreaTel, Date dtIncl, String cdUsuario) throws
ExcecaoDoDominio {
19         if (verificarPaisValido(nmPais)) {
20             return new Pais(cdPais, nmPais, cdPaisAreaTel, dtIncl,
cdUsuario);
21 
22         } else {
23             throw new ExcecaoDoDominio("pais.inexistente");
24         }
25     }
26     ...
27 }
```

Código 5 - Exemplo de Fábrica

2.2.1.1.6 Repositórios (Repositories)

Quando se tem o acesso direto ao banco, é possível obter objetos que são internos a um agregador, sem a necessidade de navegação através da raiz do agregador, violando o seu encapsulamento.

Para contornar este problema, a abordagem DDD utiliza o padrão de repositórios. Seu objetivo é encapsular toda a lógica necessária para obtenção das referências dos objetos. Como consequência, o domínio abstrai a necessidade de se preocupar em armazenar os objetos e suas referências, bastando somente acessar a infraestrutura de persistência, representada pelo repositório.

Repositórios devem ser criados apenas para as raízes dos agregadores que precisam de acesso direto. Desta forma, o cliente fica focado no modelo delegando todo o acesso e armazenamento de dados ao repositório.

A principal diferença entre repositório e fábrica é que a fábrica é responsável pela criação de novos objetos enquanto que o repositório fica responsável por armazenar e obter objetos criados previamente. Apesar dos repositórios estarem localizados na camada de dados, este é um conceito relevante ao domínio e a abordagem DDD.

```
01 /**
02  * Representa o repositório de Pontos de Venda.
03  * @see {@link PontoVenda}
04  */
05 public interface ITodosOsPontosVenda {
06     /**
07      * Obtém um Ponto de Venda a partir do CdPtov passado
08      * @param cdPtov
09      * @return A instância de ponto de venda.
10      * @see {@link PontoVenda}
11      */
12     public PontoVenda obterPontoDeVendaAPartirDeCdPtov(int cdPtov);
13 }
```

Código 6 - Exemplo de Interface de Repositório

2.2.2 Camada de Dados

Evans, em (Evans, 2004), define a camada de dados como uma camada de infraestrutura que suporta a aplicação, na qual sua responsabilidade é mapear e manipular dados para suas respectivas fontes, e fornecer para o restante da aplicação objetos mapeados de domínio.

Nesta abordagem, a aplicação não possui preocupação com a origem e destino dos dados e utiliza-se da camada de dados para obter e persistir dados. Estas fontes de dados podem ser mais comumente bancos de dados relacionais ou web services expostos em um barramento, e não há necessidade de conhecer esta complexidade.

> Em contrapartida, a camada de dados NÃO deve possuir lógica de negócio ou de aplicação.

Em (Fowler, 2002) são definidos padrões arquiteturais para fontes de dados, sendo o padrão *Data Mapper* e *Repository* bem interessantes quando trata-se de mapeamento entre o domínio da aplicação e um banco de dados.

O padrão Data Mapper é funciona como uma camada que separa objetos em memória de objetos persistidos e sua responsabilidade é isolar ambos, diminuindo o acoplamento com código SQL, por exemplo. Atualmente, o próprio mapeamento objeto-relacional da JPA e Hibernate, através do Entity Manager, fornecem um mapeador.

Para centralizar as ações sobre dados e aproveitar-se do padrão de mapeamento, deve ser utilizado o padrão *Repository*, que media a comunicação entre domínio e o mapeamento, oferecendo também suporte à construção de consultas complexas que não conseguem ser realizadas automaticamente pelo mapeador. Como muitas das fontes de dados se repetem em várias aplicações e existem operações repetitíveis, é possível se aproveitar de um repositório genérico, e reutilizar operações de consulta, listagem ou inserção.

O Repositório Genérico é uma classe que reaproveita operações básicas entre as classes da camada de dados. Esse repositório representa um contrato genérico que permite a reutilização de código e facilita o desenvolvimento e manutenção do sistema. O Código 7 apresenta um exemplo desse tipo de repositório.

Este tipo de abordagem pode expor métodos indesejáveis para determinados objetos de domínio. Nestes casos, é possível sobrescrever o método lançando uma exceção ou simplesmente não disponibiliza-los através das interfaces.

Como pode ser visto no **Código 8**, o Repositório Genérico deve ser estendido pela subclasse que queira implementar os métodos oferecidos pela superclasse. A subclasse deverá criar um construtor passando o tipo de classe que ela é responsável. Esse tipo de abordagem reduz consideravelmente o número de linhas de código da subclasse. Esta por sua vez somente deve implementar métodos específicos para a manipulação de dados da classe a qual é responsável.

```java
/**
 * Repositório Genérico o qual contém as operações de CRUD
 */
@Repository
public abstract class OracleRepositorioGenericoDados<T, TipoId>
implements IRepositorioGenericoDados<T, TipoId> {

    @PersistenceContext
    private EntityManager gerenciadorDeEntidade;

    /**
     * Classe da Entidade que será manipulada
     */
    private Class<T> classeDaEntidade;

    /**
     * Construtor genérico do repositório onde será criado um
     * gerenciador de entidade
     */
    public OracleRepositorioGenericoDados(Class<T> classe) {
        this.classeDaEntidade = classe;
    }

    public void salvar(T entidade) {
        this.gerenciadorDeEntidade.persist(entidade);
    }

    public void excluir(T entidade) {
        this.gerenciadorDeEntidade.remove(entidade);
    }

    public void alterar(T entidade) {
        this.gerenciadorDeEntidade.merge(entidade);
    }

    public EntityManager getGerenciadorDeEntidade() {
        return gerenciadorDeEntidade;
    }

    public T obterPorId(TipoId id) {
        return this.gerenciadorDeEntidade.find( this.classeDaEntidade,
id);
    }
    ...
}
```

Código 7 - Repositório Genérico

```
01 /**
02  * Representa o Repositório de Endereços.
03  */
04 @Repository
05 public class OracleEnderecoDados extends
06         OracleRepositorioGenericoDados<Endereco, EnderecoId>
07         implements IEnderecoDados {
08
09     public OracleEnderecoDados() {
10         super(Endereco.class);
11     }
12 }
```

Código 8 - Subclasse de um Repositório Genérico

2.2.2.1 Configuração da Infraestrutura de Dados

A configuração da infraestrutura para acesso aos dados é importante para permitir o pleno funcionamento da camada e para indicar estratégias e formas de implementação do acesso às fontes de dados. Na arquitetura dos sistemas da Ipiranga a configuração é centrada em dois arquivos de configuração principais.

O arquivo persistence.xml representa a unidade de persistência do sistema. Neste arquivo são configuradas as propriedades de conexão com as fontes de dados e propriedades de frameworks relacionados, como o Hibernate. O Código 9 apresenta um exemplo de configuração do arquivo e abaixo é apresentado um detalhamento das configurações deste arquivo.

A *Linha 02* define a unidade de persistência. Este campo é utilizado para configurar as informações referentes ao acesso à fonte de dados, incluindo informações sobre o provedor da Java Persistence API (JPA) e sobre o banco de dados. Neste item deve ser declarado o nome da unidade de persistência do projeto.

A *Linha 05* configura o dialeto utilizado pelo Hibernate para gerar SQL para o banco de dados. O Hibernate fornece uma série de dialetos para a conexão com diversos bancos de dados.

A *Linha 06* define a propriedade para geração automática do banco de dados a partir das classes mapeadas. Opções válidas:

- validate: valida o schema, não realiza mudanças na base.
- update: atualiza o schema.
- create: cria o schema, destrói dados prévios.
- create-drop: apaga o schema ao final da sessão.

<<<ATENÇÃO>>>, esta propriedade NÃO pode ser utilizada em ambiente de produção.

A *Linha 07* define a propriedade que permite a exibição dos SQLs gerados pelo Hibernate para o acesso à base de dados.

<<<ATENÇÃO>>>, esta propriedade NÃO pode ser utilizada em ambiente de produção.

A *Linha 08* define a propriedade que permite a formatação do SQL gerado pela propriedade "hibernate.show_sql". Essa propriedade formata o SQL para ficar mais legível para a leitura, facilitando o desenvolvimento.

<<<ATENÇÃO>>>, esta propriedade NÃO pode ser utilizada em ambiente de produção.

A *Linha 09* a propriedade que configura o Hibernate para a leitura automática das entidades mapeadas. Passando a configuração "class, hbm" o Hibernate procura por classes anotadas e por arquivos hbm de mapeamento das entidades do sistema. Sem esta configuração, todas as entidades mapeadas devem ser declaradas neste arquivo.

```
01 <persistence ... >
02     <persistence-unit name="blueprintPU" ...>
03         ...
04         <properties>
05             <property name="hibernate.dialect"
value="org.hibernate.dialect.Oracle10gDialect" />
06             <property name="hibernate.hbm2ddl.auto" value="update" />
07             <property name="hibernate.show_sql" value="true" />
08             <property name="hibernate.format_sql" value="true" />
09             <property name="hibernate.archive.autodetection"
value="class, hbm" />
10             ...
11         </properties>
12     </persistence-unit>
13 </persistence>
```

Código 9 - Configuração do Arquivo persistence.xml

O arquivo infraestrutura.xml é responsável pelas configurações de infraestrutura do Spring. Neste arquivo estão configurações de estratégias de implementação formas de criação de objetos e injeção de dependência de classes de infraestrutura da aplicação. O Código 10 apresenta um exemplo de configuração do arquivo e abaixo é apresentado um detalhamento das configurações deste arquivo.

Na *Linha 02* feita à configuração do *bean* que representa a fábrica do *entity manager*. A fábrica recebe como parâmetro uma unidade de persistência (*Linha 03*), o *bean* responsável pelo datasource (*Linha 04*) e a implementação do JPA (*Linhas 05 a 07*).

A *Linha 09* define o *bean* que representa o gerenciador de entidades (*Entity Manager*). O gerenciador de entidades fornece funcionalidades para o gerenciamento de entidades persistentes. O gerenciador de entidades está associado a um contexto de persistência de dados. Esse contexto contém um conjunto de entidades identificadas unicamente. Esse *bean* será criado a partir da fábrica definida na *Linha 02* e é passado como parâmetro na *Linha 10*.

Na *Linha 12* é definido o *bean* que permite que o *entity manager* seja injetado a partir da anotação *@PersistenceContext* (vide Código 7).

```
01 <beans ...>
02     <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
03         p:dataSource-ref="DSblueprint"
04         p:persistenceUnitName="blueprintPU">
05         <property name="jpaVendorAdapter">
06             <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
07         </property>
08     </bean>
09     <bean id="entityManager"
class="org.springframework.orm.jpa.support.SharedEntityManagerBean">
10         <property name="entityManagerFactory"
ref="entityManagerFactory"/>
11     </bean>
12     <bean
class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostPr
ocessor"/>
13     ...
14 </beans>
```

Código 10 - Configuração da Infraestrutura da Camada de Dados

2.2.2.1.1 Configuração do Datasource

O datasource oferece um meio padrão e eficiente para acesso a fontes de dados. O datasource pode oferecer diferentes formas de conexão para o banco de dados, cada um com suas particularidades. As formas de conexão são os seguintes:

- Básica – Produz uma conexão por requisição ao datasource
- Pool de Conexões – Retorna uma conexão proveniente de um pool de conexões.
- Transação Distribuída – Produz uma conexão que pode utilizada para transações distribuídas e na sua maioria das vezes participa de um pool de conexões.

Para a utilização de um datasource no sistema, é necessária a realização de configuração em alguns arquivos. Como pode ser visto no Código 11, é necessário configurar o arquivo persistence.xml. As *Linhas 03* apresentam a configuração do datasource para a unidade de persistência. Ambas realizam a mesma configuração, a diferença é o tipo de gerenciador de transação utilizada pela aplicação. Caso esteja sendo usada a JTA deve-se utilizar a *tag* <*jta-data-source*> caso contrário deve-se utilizar a *tag* <*non-jta-data-source*>.

```
01 <persistence ... >
02     <persistence-unit name="blueprintPU" transaction-type="JTA">
03         <jta-data-source>java:comp/env/jdbc/DSblueprint</jta-data-
source>
04         ...
05     </persistence-unit>
06 </persistence>
```

```
01 <persistence ... >
02     <persistence-unit name="blueprintPU" transaction-
type="RESOURCE_LOCAL">
03         <non-jta-data-source>java:comp/env/jdbc/DSblueprint</non-jta-
data-source>
04         ...
05     </persistence-unit>
06 </persistence>
```

Código 11 - Configuração do Datasource no persistence.xml

Também é necessária a configuração do datasource para Spring, para que o framework possa injetar corretamente as dependências nos *beans* passando a referência do datasource. Essa configuração deve ser realizada no arquivo infraestrutura.xml. A *Linha 07* configura o *lookup* do endereço do datasource via JNDI, essa linha também informa o tipo de datasource esperado. Com essa configuração, é passada uma referência para a fábrica do *Entity Manager* na *Linha 03*. Assim, a fábrica poderá criar o *Entity Manager* passando a referência para o datasource.

```
01 <beans ...>
02     <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
03         p:dataSource-ref="DSblueprint"
04         p:persistenceUnitName="blueprintPU">
05         ...
06     </bean>
07     <jee:jndi-lookup jndi-name="jdbc/DSblueprint" id="DSblueprint"
expected-type="javax.sql.DataSource"/>
08     ...
09 </beans>
```

Código 12 - Configuração do Datasource no infraestrutura.xml

2.2.2.2 Mapeamento Objeto-Relacional

A especificação JPA (Java Persistence API) oferece suporte aos conceitos arquiteturais de mapeamento objeto-relacional, definidos em Fowler (2002), permitindo abstrair o modelo relacional comuns em bancos de dados como o Oracle. O Hibernate, framework de persistência líder de mercado, implementa esta especificação.

Além de realizar o mapeamento, é possível melhorar a performance da aplicação a partir de configurações de Cache, Pool de conexões. Isso pode ser configurado através do arquivo "*persistence.xml*".

Estas configurações de *cache* habilitam que entidades, coleções de entidades ou consultas possam ser armazenados para acesso rápido, otimizando consultas ou evitando ir ao banco de dados. Isso pode ser realizado tanto pelo mapeamento da JPA, para *cache* de entidades e coleções, quanto pelo *Entity Manager*, para *cache* de consultas. As configurações sobre conexão e um *pool* das mesmas podem ser realizadas (de preferência) em uma fonte de dados localizada no próprio servidor de aplicação.

2.2.2.3 Consumo de dados via Web Services

Em muitos casos, uma entidade ou um determinado conjunto de dados podem não ser acessíveis em uma base de dados conhecida à aplicação, e a persistência destes está abstraída em um web service distribuído. Nestes casos podemos extrair as classes necessárias para consumo dos dados a partir de um WSDL (Web Services Description Language) utilizando a API JAX-WS.

Utilizando o RAD, é possível importar o serviço através de seu WSDL, utilizando New >> Web Service Client.

Com abordagem de serviços, o conceito de consumo e persistência dos dados funciona de forma semelhante ao repositório, e a camada de serviço só precisará conhecer a interface de acesso da camada de dados. Por definição a interface de acesso, a implementação desta interface e os arquivos criados ao gerar o cliente web servisse devem estar presentes na camada de dados de acordo com a regra de pacotes estabelecida nesse documento.

```
01 public interface ISaaDados {
02
03    public long obtemUsuarioAutenticado(Usuario usuario) throws
DadosException;
04    public List<Boolean> obtemAutorizacaoUsuario(Usuario usuario) throws
DadosException;
05    public boolean usuarioPertencePerfil(Usuario usuario, int
codigoPerfil) throws DadosException;
06
07}
```

Código 13 - Interface de Acesso aos Serviços do SAA

O novo padrão Java também define que o ENDPOINT utilizado para apontamento do ambiente de consumo do serviço (desenvolvimento, homologação ou produção) seja obtido diretamente do servidor de aplicação Websphere, que será o responsável por prover o servidor correto. Está informação está disponível através da JNDI de nome url/Barramento.

2.2.2.4 Utilização da base de dados

Além da utilização e mapeamento do banco de dados através da JPA, é possível utilizar alguns recursos do próprio SGBD para obter melhoria de performance ou centralização de alguma regra de negócio, por exemplo.

2.2.2.4.1 Stored Procedures

A JPA permite a utilização de *querys* nativas. Isto permite que possam ser utillizadas *functions* ou *stored procedures* conforme exemplo abaixo:

```
01 StringBuilder consulta = new StringBuilder("CALL ");
02
consulta.append("obterCondicoesComerciaisComMesmoProdutoPontoVendaEUnidade
Operacional ");
03 consulta.append("(:cdProduto, :cdUnidadeOperacional, :cdPtov)");
04 try {
05          Query procedure = repositorioDeCondicoesComerciais
06
.getGerenciadorDeEntidade().createNativeQuery(consulta.toString());
07          procedure.setParameter("cdProduto", cdProduto);
08          procedure.setParameter("cdUnidadeOperacional",
cdUnidadeOperacional);
09          procedure.setParameter("cdPtov", cdPtov);
10
11          procedure.executeUpdate();
12
13 } catch (Exception e) {
14          throw new ExcecaoDeDados();
15 }
```

Código 14 - Exemplo de Chamada de Procedure

Também é possível obter como resultado da execução da procedure em um parâmetro. Para isso, seria necessário incluir na chamada *createNativeQuery* um ValueObject como retorno, e que a execução da procedure fosse realizada através da chamada *getResultList*.

2.2.2.4.2 Regras tratadas em banco

Outra maneira de repassar responsabilidades ao banco de dados é realizar processamento de dados ou de regras de negócio internamente pelo SGBD. As regras de negócio podem estar em procedures, mas em alguns casos, podem ser realizadas consultas muito complexas, que exigem processamentos com alta performance, e isto pode onerar a aplicação desnecessariamente.

Isto pode ser observado em um somatório financeiro que inclui busca em diversas taxas em diferentes tabelas. Neste caso um ValueObject poderia ser instanciado a partir deste somatório, que pode ser calculado pelo próprio SGBD.

```
01 public Collection<ConsultaProdutosVo>
gerarRelatorioDeComprasPorClientesDa(
02          GerenciaComercial gerenciaComercial) {
03      EntityManager entityManager = repositorioDeGerenciasComerciais
04              .getGerenciadorDeEntidade();
05      StringBuilder consulta = new StringBuilder(
06              "SELECT NEW
ipp.aci.pdsiblueprint.dominio.vo.ConsultaProdutosVo(pv.cdPtov,
rc.produto.cdprod , SUM(rc.valorCompra))");
07      consulta.append(" FROM PontoVenda pv JOIN pv.registroCompra rc");
08      consulta.append(" WHERE pv.gerenciaVenda.gerenciaComercial =
:gerenciaComercial");
09      consulta.append(" GROUP BY pv.cdPtov, rc.produto.cdProd");
10
11      TypedQuery<ConsultaProdutosVo> q = entityManager.createQuery(
12              consulta.toString(), ConsultaProdutosVo.class);
13      q.setParameter("gerenciaComercial", gerenciaComercial);
14
15      Collection<ConsultaProdutosVo> resultado = q.getResultList();
16      return resultado;
17  }
```

Código 15 - Exemplo de Tratamento de Regras no Banco

2.2.2.4.3 Cache de dados

Como descrito na seção de mapeamento objeto-relacional, uma das maneiras de aumentar performance da aplicação e evitar consumo desnecessário do banco de dados, é realizar o cache de instâncias de entidades e coleções ou *queries*.

Para isso, é necessário utilizar as anotações da JPA para informar ao gerenciador que deseja-se fazer cache da entidade ou da coleção.

```
01 @Cacheable
02 @Cache(usage=CacheConcurrencyStrategy.READ_WRITE)
03 public class Pais implements java.io.Serializable
04 { ... }
```

Código 16 - Exemplo de Cache na Entidade

É possível fazer configurações específicas sobre tamanho do cache, política de expiração, localização física (memória, arquivos indexados, cluster) e concorrência. Para isso deve-se utilizar o arquivo de configuração do provedor de cache (*EhCache*), o que permite fazer uma configuração padrão e para cada utilização do cache.

Também é possível realizar uma invalidação programática do cache através do *EntityManager*, e isto pode ser realizado, por exemplo, sempre que for buscar dados financeiros em uma determinada entidade.

Em outros casos, é possível realizar caches de consultas executadas com frequência. Isto fará com que as chaves das entidades obtidas na consulta sejam armazenadas, para que as próximas consultas sejam realizadas através de chaves primárias e tenham melhor desempenho.

Com o Hibernate, basta informar na consulta que deseja realizar o cache:

```
01 queryJPA.setHint("org.hibernate.cacheable", true);
```

Código 17 - Exemplo de Cache na Query

Quando NÃO usar cache:

- Dados com alto volume de alteração
- Dados legados compartilhados
- Dados financeiros

2.2.3 Camada de Serviço

Fowler em, (Fowler, 2002), define quatro padrões para a organização da lógica de domínio. (Evans, 2003) sugere e emprega uma camada auxiliar de serviços para coordenação e monitoramento das atividades executadas na aplicação, organizando o acesso aos dados. Esta camada segue o padrão Service Layer definido por (Fowler, 2002). Service Layer é, essencialmente, uma centralização das operações executadas na aplicação em uma única camada, encapsulado interações complexas que envolvem transações entre múltiplos recursos e coordenando diversas respostas a uma ação determinada. Ao contrário da camada de domínio, que se responsabiliza pela lógica de negócio, esta camada se responsabiliza pela lógica da aplicação.

Este padrão, assim como qualquer outro, possui vantagens e desvantagens que serão descritas a seguir:

Vantagens:

- Permite a separação da lógica da aplicação da lógica de negócio.
- Permite que os objetos de domínio sejam reutilizados em diferentes aplicações.
- Diminui a dependência de aspectos e bibliotecas específicas da aplicação.

Desvantagens:

- Pode gerar uma complexidade desnecessária em aplicações simples.
- Não recomendado para aplicações com uma interface simples cujas respostas não envolvam múltiplas entidades ou recursos.

Levando em consideração que os sistemas poderão ser integrados entre si, um mecanismo central para a lógica da aplicação se faz necessário. Para este fim, recomenda-se o uso do padrão Service Layer. Tal padrão proporcionará uma maior organização da lógica da aplicação, bem como um mecanismo robusto de auditoria e monitoramento.

A camada de serviços conhece classes das demais camadas, com exceção da camada de apresentação. A camada de apresentação é a única que conhece a de serviços, chamando as operações necessárias para o tratamento das diversas requisições. A divisão de camadas visa uma melhor modularização e divisão de responsabilidades dos diversos componentes presentes no projeto. A camada de serviços não deve possuir lógica de persistência de dados, ou de negócio devendo delegar tal responsabilidade para a camada de dados e domínio, respectivamente. Ao fazer uso da camada de dados e de domínio, os serviços estarão independentes da tecnologia especifica de persistência de dados e dos detalhes de implementação das regras do negócio.

É interessante pensar em construir um componente que represente um determinado contexto do negócio. Além disso, cada componente presente nesta camada deve conter o prefixo “Servicos”. Logo, se existir um componente que represente um endereço ele deve possuir o nome “ServicosDoEndereco”. Esta padronização por nome facilitará a manutenção do código.

> Serviços podem acessar componentes de camadas inferiores, no entanto, um serviço não pode acessar outro serviço da mesma camada.

2.2.3.1 Spring Framework

Esta camada pode se beneficiar de uma mecanismo de injeção de dependências e inversão de controle, diminuindo o acoplamento entre classes e camadas. Isto se torna fundamental na camada de serviços, uma vez que ela interage simultaneamente com a camada de dados e de domínio, além de conter serviços transversais à aplicação.

O Spring é um framework líder de mercado que possui este mecanismo implementado. Sua implementação através de anotações facilita o desenvolvimento e permite utilização dos serviços sem o conhecimento da criação dos mesmos.

```
01 //Anotação para controle do Spring
02 @Service
03 public class ServicosDePais {
04     //Injeção de Dependência pelo Spring
05     @Autowired
06     private IPaisDados todosOsPaises;
07
08     public void excluirPaisAPartirDoCodigo(int cdPais)
09             throws PaisNaoPodeSerExcluidoException {
10         Pais paisASerExcluido =
todosOsPaises.obterPaisAPartirDeCdPais(cdPais);
11         if (paisASerExcluido.verificarPossibilidadeDeExclusao()) {
12             todosOsPaises.removerPais(paisASerExcluido);
13         } else {
14             throw new
PaisNaoPodeSerExcluidoException("pais.naoPodeSerExcluido");
15         }
16
17     }
18 }
```

Código 18 - Exemplo de Anotações do Spring

Por possuir uma suíte extensa de funcionalidades, além do mecanismo de injeção de dependência, o Spring tem suporte para diversas questões arquiteturais e de implementação, como controle de transações, configuração de fontes de dados, integração com log e serviço de e-mail, entre outros.

2.2.3.2 Controle de Transação

O Spring Framework fornece um modelo de apoio à manipulação de transações. Este modelo é facilmente integrável com diversas tecnologias e diferentes APIs de apoio a transação. Neste modelo, existem duas formas para a implementação do gerenciamento de transações:

- Gerenciamento de Transação Declarativa

- Gerenciamento de Transação Programática

O Gerenciamento de Transação Declarativa pode ser utilizado através de configurações em XML e via anotações no código Java. No Gerenciamento de Transação Programática o Spring oferece uma API para facilitar a manipulação de transações. Para o a arquitetura dos sistemas da Ipiranga deverá ser utilizado o Gerenciamento de Transação Declarativa através de anotações. Este modelo é menos intrusivo no código fonte do que o modelo programático e é mais fácil de manter do que o modelo declarativo via XML.

2.2.3.2.1 Configuração do Controle de Transação

Como dito anteriormente, o modelo fornecido pelo Spring para controle de transações suporta diversas APIs. Entre as APIs mais utilizadas é possível citar:

- *DataSourceTransactionManager* – Classe para gerenciamento de transações associada a *DataSources*.
- *HibernateTransactionManager* – Classe para gerenciamento de transações fornecido pelo Hibernate.
- *JpaTransactionManager* – Classe para gerenciamento de transações associada a JPA. Essa interface também aceita transações associadas com *DataSources*.
- *JtaTransactionManager* – Classe para gerenciamento de transações associada a JTA. Suporta múltiplos recursos transacionais na mesma transação. Geralmente é associado a um servidor JEE coordenador de transação.

No contexto desta arquitetura serão utilizadas duas classes, a do *JpaTransactionManager* e a do *JtaTransactionManager*. Dependendo da necessidade da aplicação uma ou outra é preferível.

É recomendada a utilização do *JpaTransactionManager* quando a aplicação implementa um único JPA *EntityManager*. Esse mecanismo vincula o JPA *EntityManager* a partir de uma fábrica especificada à uma thread. Essa interface pode ser utilizada também com *DataSources* e com conexões JDBC diretamente.

Como pode ser visto no Código 19, para configurar o gerenciador de transação via JPA é preciso passar a fábrica responsável pela construção do EntityManager para o *bean* do gerenciador. Além disso, é preciso informar que o controle de transação é orientado a anotações. O Código 20 apresenta a configuração necessária no arquivo *persistence.xml*. Neste arquivo é configurado o tipo de transação como RESOURCE_LOCAL.

```
01 <tx:annotation-driven />
02
03 <bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
04     <property name="entityManagerFactory" ref="entityManagerFactory"/>
05 </bean>
```

Código 19 - Configuração do Controle de Transação em JPA para o Spring

```
01 <persistence ... >
02    <persistence-unit name="blueprintPU" transaction-
type="RESOURCE_LOCAL">
03          ...
04    </persistence-unit>
05 </persistence>
```

Código 20 - Configuração do Controle de Transação em JPA no persistence.xml

Para acessar múltiplos recursos transacionais em uma mesma transação é recomendada a utilização do *JtaTransactionManager*. Esse gerenciador de transação pode controlar transações distribuídas e para controlar transações de recursos do servidor de aplicação.

Como pode ser visto no Código 21, para configurar o gerenciador de transação via JTA basta declarar a *tag* <tx:*jta-transaction-manager*>. Esta *tag* é equivalente a criação de um *bean* com a definição da classe que implementa o gerenciamento de transação JTA. Além disso, essa *tag* automaticamente detecta a utilização de servidores de aplicação que necessitam de configurações específicas, sendo o caso do Websphere. Assim como na configuração em JPA é preciso informar que o controle de transação é orientado a anotações. O Código 22 apresenta a configuração necessária no arquivo *persistence.xml*. Neste arquivo, é configurado o tipo de transação como JTA e é informada a plataforma de controle de transação para o Hibernate.

```
01 <tx:annotation-driven />
02 <tx:jta-transaction-manager />
```

Código 21 - Configuração do Controle de Transação em JTA no Spring

```
01 <persistence ... >
02     <persistence-unit name="blueprintPU" transaction-type="JTA">
03         ...
04         <properties>
05             ...
06             <property name="hibernate.transaction.jta.platform"
value="org.hibernate.service.jta.platform.internal.WebSphereExtendedJtaPla
tform" />
07         </properties>
08     </persistence-unit>
09 </persistence>
```

Código 22 - Configuração do Controle de Transação em JTA no persistence.xml

2.2.3.2.2 Declaração da Transação

Como dito anteriormente, a criação e utilização de transações no código dos sistemas da Ipiranga deve ser realizada através da transação declarativa com anotações. O Código 23 apresenta um exemplo de utilização da transação declarativa via anotação.

Como pode ser visto no Código 23 a anotação *@Transactional* deve ser utilizada antes da declaração dos métodos. Com a finalidade de otimização do controle de transações é possível passar o parâmetro "*readOnly = true*" para que o gerenciador de transações otimize a transação para somente leitura do banco de dados. Por padrão, o gerenciador de transações assume que a transação é de leitura/gravação do banco de dados, sendo assim, não é necessário declarar o campo como "*readOnly = false*".

```
01 @Service
02 public class ServicosDePontoVenda {
03     ...
04     @Transactional(readOnly = true)
05     public PontoVenda obterPontoVenda(int cdPtov) {
06         ...
07     }
08     @Transactional(readOnly = true)
09     public ZonaVenda obterZonaVenda(String cdZonven) {
10         ...
11     }
12     @Transactional(readOnly = true)
13     public List<ZonaVenda> obterZonasVendaDaGerencia(String cdGv) {
14         List<ZonaVenda> zonasVendaAssociadas = todasAsZonasVenda
15                 .obterZonasDeVendaDaGerencia(cdGv);
16         return zonasVendaAssociadas;
17     }
18     @Transactional
19     public void incluirNovaZonaVendaAPartirDe(String cdZonven, String
nmZonven,
20             String cdGv, String cdCoordVnd, String cdEmprFunc,
21             Integer noMatrFunc, String cdUsuario) {
22         GerenciaVenda gerenciaVenda =
todasAsGerenciasVenda.obterPorId(cdGv);
23         ZonaVenda zonaVenda = new ZonaVenda(cdZonven, gerenciaVenda,
nmZonven,
24                 null, null, null, cdCoordVnd, cdEmprFunc, noMatrFunc,
null,
25                 cdUsuario);
26         todasAsZonasVenda.salvar(zonaVenda);
27     }
28 }
```

Código 23 - Exemplo de Declaração de Transação

Também é possível anotar a classe ao invés dos métodos para o controle de transações. O Código 24 exemplifica a utilização da anotação na classe. As anotações nos métodos são preferenciais às anotações na classe. Desse modo é possível anotar o método com uma recorrente e sobrescrever a anotação em alguns métodos. No exemplo, a *Linha 08* sobrescreve a anotação da *Linha 02*.

```
01 @Service
02 @Transactional(readOnly = true)
03 public class ServicosDeRegistroCompra {
04     ...
05     public RegistroCompra obterRegistroCompra(Long numero_reg_cpra) {
06         ...
07     }
08     @Transactional
09     public List<Long> incluirRegistrosCompraAPartirDeArquivo(
10             String caminhoArquivoUpload) throws ParseException,
11             ExcecaoDeServico {
12         ...
13     }
14     public Collection<RegistroCompra>
obterRegistrosCompraAssociadosAProduto(
15             int cdProd) {
16         ...
17     }
18     public Collection<RegistroCompra>
obterRegistrosCompraAssociadosAUnidadeOperacional(
19             String cdUnidOper) {
20         ...
21     }
22     public Collection<RegistroCompra> obterRegistrosCompraAssociadosA(
23             int cdPtov, int cdProd, String cdUnidOper){
24         ...
25     }
26     public Collection<RegistroCompra> obterTodosRegistrosCompra(){
27         ...
28     }
29 }
```

Código 24 - Exemplo de Declaração de Transação na Classe

A anotação *@Transactional* é um metadado que especifica que uma classe ou método deve ter uma semântica transacional. O fluxo padrão de execução de uma transação é iniciado quando um método é chamado, nesse instante é iniciada uma nova transação. As operações são realizadas dentro do método e ao final é realizado o *commit* automático. Caso seja levantada uma exceção do tipo *RuntimeException* o gerenciador de transação realiza o *rollback*. As configurações padrões da anotação são as seguintes:

- Nível de propagação *PROPAGATION_REQUIRED*.
- Nível de isolamento *ISOLATION_DEFAULT*.
- A transação é de leitura/escrita.
- O tempo para expirar a transação segue o padrão do sistema, caso não exista padrão a transação não é limitada com tempo para expirar.
- Qualquer *RuntimeException* dispara automaticamente o *rollback*.

Essas configurações podem ser alteradas dependendo da necessidade. A anotação aceita as seguintes propriedades opcionais:

Tabela 1 - Propriedades da anotação

Propriedade	Descrição

value	Especifica o gerenciador de transação para ser utilizado.
propagation	Especifica o nível de propagação. Suporta os seguintes valores: • MANDATORY – Suporta a transação corrente. Lança exceção caso nenhuma exista. • NESTED – Executa em uma transação aninhada se já existir alguma. Somente funciona em alguns gerenciadores transacionais específicos. • NOT_SUPPORTED – Não executa a transação e suspende a transação caso exista. • REQUIRED - Suporta a transação corrente. Cria uma nova transação caso nenhuma exista. • REQUIRES_NEW – Cria uma nova transação. Suspende a transação corrente caso exista. • SUPPORTS – Suporta a transação corrente e executa de forma não transacional caso nenhuma exista.
isolation	Especifica o nível de isolamento. • DEFAULT – Usa o nível de isolamento padrão do banco de dados utilizado. • READ_COMMITTED – Indica que a leitura de dados “sujos” são evitados. • READ_UNCOMMITTED – Indica que a leitura de dados “sujos”, não repetidos e “fantasmas” podem ocorrer. • REPEATABLE_READ - Indica que a leitura de dados “sujos” e não repetidos são evitados. Entretanto, dados “fantasmas” podem ocorrer. • SERIALIZABLE - Indica que a leitura de dados “sujos”, não repetidos e “fantasmas” são evitados.
readOnly	Especifica se a transação é somente leitura ou leitura/escrita.
timeout	Especifica o tempo para expirar a transação.
rollbackFor	Especifica uma lista de classes de exceções que causam *rollback*.
rollbackForClassname	Especifica uma lista de nomes de classes de exceções que causam *rollback*.
noRollbackFor	Especifica uma lista de classes de exceções que NÃO causam *rollback*.
noRollbackForClassname	Especifica uma lista de nomes de classes de exceções que NÃO causam *rollback*.

2.2.3.3 Log

O Spring possui integração com o framework Log4j, para geração de log de aplicação. Algumas boas práticas de auditoria de aplicação podem ser empregadas com o uso do Log:

- Utilizar com consciência os níveis de log, de acordo com a situação

  Exemplos: DEBUG para testes, INFO para dados de auditoria, ERROR para rastreamento de bugs;
- Utilizar um local em separado para os arquivos de log, com tamanho máximo para que não fiquem ilegíveis;
- Padrão de escrita conciso e objetivo que permita identificar o local do erro, a causa do mesmo, o momento em que aconteceu, e principalmente, os parâmetros envolvidos;

O Log pode ser combinado com o paradigma de orientação à aspectos, permitindo utilizar o log de forma transparente em locais comuns da aplicação, sem que haja necessidade de escrita do desenvolvedor. O Spring AOP possui esta capacidade, através da implementação do *AspectJ*.

O código abaixo mostra um exemplo de interceptador de log, utilizando os conceitos da orientação à aspectos para realizar um ponto de corte em todas as chamadas de métodos da camada de serviços, incluindo uma escrita no início do método, no fim do método, e em caso de erro.:

```
01 @Aspect
02 public class LogInterceptador {
03     /**
04      * Através de expressão regular, o aspecto realiza o ponto
05      * de corte em todas as execuções de métodos da camada de serviço.
06      */
07     @Around("execution(* ipp.aci.*.servicos.*.ServicosDe*.*(..))")
08     public void criarLogServico(ProceedingJoinPoint joinPoint) throws
Throwable{
09         //Utilização do Log4j
10         Log.debug("Inicializando método " +
joinPoint.getSignature().getName());
11         try{
12             joinPoint.proceed();
13         } catch (Exception e) {
14             Log.error("Falha ao executar o método " +
joinPoint.getSignature().getName() + ".", e);
15             throw e;
16         }
17         Log.debug("Finalizando método com sucesso.");
18     }
19 }
```

Código 25 - Exemplo de um Apecto para Log

Para o funcionamento desta estratégia de log, é necessária a realização da configuração da infraestrutura de log. A configuração pode ser realizada de duas formas. A primeira é através do arquivo web.xml. Esta configuração é a melhor maneira de configurar o log, pois ela é iniciada juntamente com a aplicação. Esta configuração pode ser vista no Código 26. No exemplo, a *linha 06* define um nome no contexto do arquivo e a *linha 06* especifica a localização do arquivo de configuração do log. As *linhas de 08 a 10* especificam a classe responsável por gerenciar o log da aplicação.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <web-app ...>
03     ...
04     <context-param>
05         <param-name>log4jConfigLocation</param-name>
06         <param-value>/WEB-INF/classes/log4j.xml</param-value>
07     </context-param>
08     <listener>
09         <listener-
class>org.springframework.web.util.Log4jConfigListener</listener-class>
10     </listener>
11     ...
12 </web-app>
```

Código 26 - Configuração do Log no web.xml

O segundo tipo de configuração é concentrado no arquivo infraestrutura.xml. Este tipo foi utilizado para a configuração do log para os testes da aplicação. Como a aplicação não é iniciada durante a execução dos testes, o arquivo web.xml não é iniciado acarretando no não funcionamento do log. Desta forma, o log é configurado através do arquivo de configuração do Spring. O Código 27 apresenta um exemplo de configuração do log. Como pode ser visto no exemplo, na *Linha 03* é declarado que o log será realizado através de Aspectos fornecidos pelo Spring. A *Linha 04* declara o *bean*/aspecto responsável por fazer o log da aplicação. O *bean*/aspecto declarado nesta linha é o apresentado no Código 25 em mais detalhes. A *Linha 05* configura a utilização do framework Log4J para fornecer a infraestrutura para o log. Esta linha configura o inicializador do log4j do Spring, permitindo incluir as configurações realizadas no *classpath* do projeto. A *Linha 11* indica o caminho para o arquivo de configuração do log.

```
01 <beans ...>
02     ...
03     <aop:aspectj-autoproxy/>
04     <bean id="logAspecto"
class="ipp.aci.pdsiblueprint.servicos.LogInterceptador" />
05     <bean id="log4jInitializer"
class="org.springframework.beans.factory.config.MethodInvokingFactoryBean"
>
06         <property name="staticMethod">
07
<value>org.springframework.util.Log4jConfigurer.initLogging</value>
08         </property>
09         <property name="arguments">
10             <list>
11                 <value>classpath:META-INF/log4j.xml</value>
12             </list>
13         </property>
14     </bean>
15 </beans>
```

Código 27 - Configuração do Log para testes

As configurações realizadas no arquivo log4j.xml podem ser vistas no Código 28. Como dito anteriormente, o Log4j fornece a infraestrutura para o log, como gravação de arquivo, formatação do texto do log, nível de depuração, entre outros. Na *Linha 02* deste arquivo é configurado o *FileAppender*. Esse tipo de *FileAppender* é responsável por fazer o backup de log quando ele atinge um certo tamanho. Na *Linha 03* é configurado o caminho para o arquivo de log. Na *Linha 04* é configurado o tamanho máximo do arquivo de log. Nas *Linhas 05, 06 e 07* é feita a configuração da formatação da mensagem do log. Na *Linha 10* é declarado que os logs dentro do root irão fazer o log em nível de Debug. Na *Linha 11* é incluído o *FileAppender* configurado anteriormente para realizar o log no nível do root.

```
01 <log4j:configuration>
02     <appender name="fileAppender"
class="org.apache.log4j.RollingFileAppender">
03         <param name="File"
value="/opt/WebSphere/logs/AppWeb/LogPDSI.log" />
04         <param name="MaxFileSize" value="5MB" />
05         <layout class="org.apache.log4j.PatternLayout">
06             <param name="ConversionPattern" value="%d{HH:mm:ss,SSS} %5p
[%-20c{1}] %m%n" />
07         </layout>
08     </appender>
09     <root>
10         <level value="DEBUG" />
11         <appender-ref ref="fileAppender" />
12     </root>
13 </log4j:configuration>
```

Código 28 - Configurações do Log4j

O diretório de LOG deve ser indicado como **/opt/WebSphere/logs/AppWeb/** que é o diretório mapeado nos servidores para pesquisa e controle de arquivos de log.

2.2.3.4 E-mail

O Spring oferece serviço de envio de e-mail completo, com opções para envio de texto ou html, para um ou vários destinatários. Uma das funcionalidades também permite que seja configurado um HTML padrão de envio, descrito nos próprios arquivos de configuração do Spring.

Abaixo, um exemplo de função para enviar e-mail através do Spring.

```
01 @Service
02 public class ServicosDeEmail {
03     @Autowired
04     private JavaMailSender mailSender;
05     @Autowired
06     private SimpleMailMessage emailPadrao;
07
08     public void enviarEmailTexto(String remetente, String[]
destinatarios, String assunto, String corpoDoEmail) throws
ExcecaoDeServico {
09         try {
10             emailPadrao.setFrom(remetente);
11             emailPadrao.setTo(destinatarios);
12             emailPadrao.setSubject(assunto);
13             emailPadrao.setText(corpoDoEmail);
14             mailSender.send(emailPadrao);
15         } catch (MailException e) {
16             throw new ExcecaoDeServico("email.naoFoiPossivelEnviar");
17         }
18     }
19 }
```

Código 29 - Exemplo de Função para Envio de E-mail

Para o funcionamento desta estratégia de envio de e-mail, é necessária a realização da configuração da infraestrutura de e-mail. Essa configuração é concentrada no arquivo infraestrutura.xml. O Código 30 apresenta um exemplo da configuração realizada no arquivo. Como pode ser visto no exemplo, a *Linha 03* configura o *bean* responsável pelo envio de e-mail. A Linha 04 configura o host do servidor de envio de e-mail. A *Linha 05* configura a porta para o envio de e-mail. A Linha 06 configura o usuário e a *Linha 07* configura a senha do usuário. Na *Linha 10* é configurado o protocolo de comunicação do e-mail. Na *Linha 11* é configurada a autenticação do protocolo SMTP. A *Linha 12* habilita a proteção TLS da conexão. Na *Linha 13* é configurado o debug do envio de e-mail. Da *Linha 10 a 29* é configurado um exemplo de e-mail padrão para o sistema. Este e-mail é composto de um HTML que é definido nesta configuração.

```
<beans ...>
    ...
    <bean id="mailSender"
class="org.springframework.mail.javamail.JavaMailSenderImpl">
        <property name="host" value="smtp.gmail.com"/>
        <property name="port" value="25"/>
        <property name="username" value="testeipirangapdsi@gmail.com"/>
        <property name="password" value="ipiranga123"/>
        <property name="javaMailProperties">
            <props>
                <prop key="mail.transport.protocol">smtp</prop>
                <prop key="mail.smtp.auth">true</prop>
                <prop key="mail.smtp.starttls.enable">true</prop>
                <prop key="mail.debug">true</prop>
            </props>
        </property>
    </bean>
    <bean id="emailPadrao"
class="org.springframework.mail.SimpleMailMessage">
        <property name="text">
            <value>
                <![CDATA[
                    <html>
                    <body>
                        <h1>Hello %s</h1>
                    </body>
                    </html>
                ]]>
            </value>
        </property>
    </bean>
</beans>
```

Código 30 - Configuração da Infraestrutura de Envio de E-mail

2.2.4 Camada de Apresentação

2.2.4.1 Visão Geral

2.2.4.1.1 Descrição

A camada de apresentação representa a interface entre o sistema e o usuário. Nesta camada são exibidas as páginas, ou telas, onde o usuário fornece entradas e recebe os resultados, ou saídas, do sistema.

A interação entre o usuário e o sistema é gerenciada nesta camada, fornecendo os resultados de acordo com as requisições. Nela, é encapsulada a comunicação com a camada de serviços que contém as lógicas do negócio e o acesso a dados persistentes, possibilitando a melhor separação de responsabilidades através de menor acoplamento e, consequentemente, garantindo maiores manutenibilidade e qualidade do projeto.

Serviços (web services) também podem ser representados nesta camada, sendo expostos através de controladores que, respeitando os padrões, delegam as chamadas para as camadas que contém a implementação dos serviços.

Lógicas de negócio ou de acesso a dados não devem ser implementados na camada de apresentação.

Esta camada deve conter apenas lógica de interface e de interação com o usuário e chamadas à camada de serviços.

Controladores da camada de "Apresentação" tem que usar Serviços da camada de "Servicos" para acessar os dados do domínio ou os serviços do domínio.

2.2.4.1.2 Padrões e Frameworks

O framework utilizado para a camada de apresentação é o Spring MVC, que implementa o padrão MVC – *Model View Controller*, através do qual as responsabilidades são distribuídas, sendo elas:

- Representação de conceitos de negócio e passagem de dados (*Model*)
- Exibição de telas ou páginas (*View*)
- Gerenciamento das interações com o usuário (*Controller*).

O papel das telas ou páginas (*View*) é realizado pelas páginas HTML e JSP, incluindo fragmentos JSP do framework Tiles para a montagem de páginas. O framework Tiles, responsável por renderizar as visões, permite a montagem de páginas a partir de fragmentos JSP. Esta montagem de páginas realizada pelo Tiles é uma implementação do padrão *Composite*.

O papel de modelo (*Model*) é realizado pelas classes VO, bem como pelas entidades persistentes e agregadores e objetos *Model*, do Spring. Estas classes, representantes do papel de *Model*, são trafegadas entre as páginas e os controladores.

O papel do gerenciamento das interações com o usuário é realizado pelas classes controladoras do Spring, anotadas com *@Controller*.

O Spring MVC também implementa o padrão *FrontController*, que permite a separação das responsabilidades de tratamento da requisição (*controller*) e exibição da resposta (*view*).

Como a tecnologia de visões (*views*) é JSP, aplica-se, aqui, o padrão *Template View*, onde *tags* são inseridas em conteúdos JSP e HTML, para serem substituídas por conteúdo dinâmico em tempo de execução.

Maiores Informações Sobre os Padrões e Frameworks

Tabela 2 - Referências para tecnologias

Padrão / Framework	Descrição
Spring MVC	http://spring.io/
Tiles	https://tiles.apache.org/
MVC - Model View Controller	http://www.martinfowler.com/eaaCatalog/modelViewController.html
Composite	http://www.oodesign.com/composite-pattern.html
VO - Value Object	http://martinfowler.com/bliki/ValueObject.html

DDD – Domain Driven Design	http://www.domainlanguage.com/ddd/ http://martinfowler.com/bliki/tags/domain%20driven%20design.html http://refcardz.dzone.com/refcardz/getting-started-domain-driven
Front Controller	http://en.wikipedia.org/wiki/Front_Controller_pattern
Template View	http://www.martinfowler.com/eaaCatalog/templateView.html

2.2.4.2 Arquitetura

O Spring MVC é um framework orientado a requisições e trabalha com uma Servlet central que despacha a requisição para o controlador adequado, através do padrão FrontController. Além disso, essa Servlet oferece acesso aos demais serviços do Spring, como injeção de dependências.

2.2.4.2.1 Fluxo de Execução

O diagrama abaixo ilustra o modelo do fluxo no framework, onde a aplicação deve fornecer seus controladores e views.

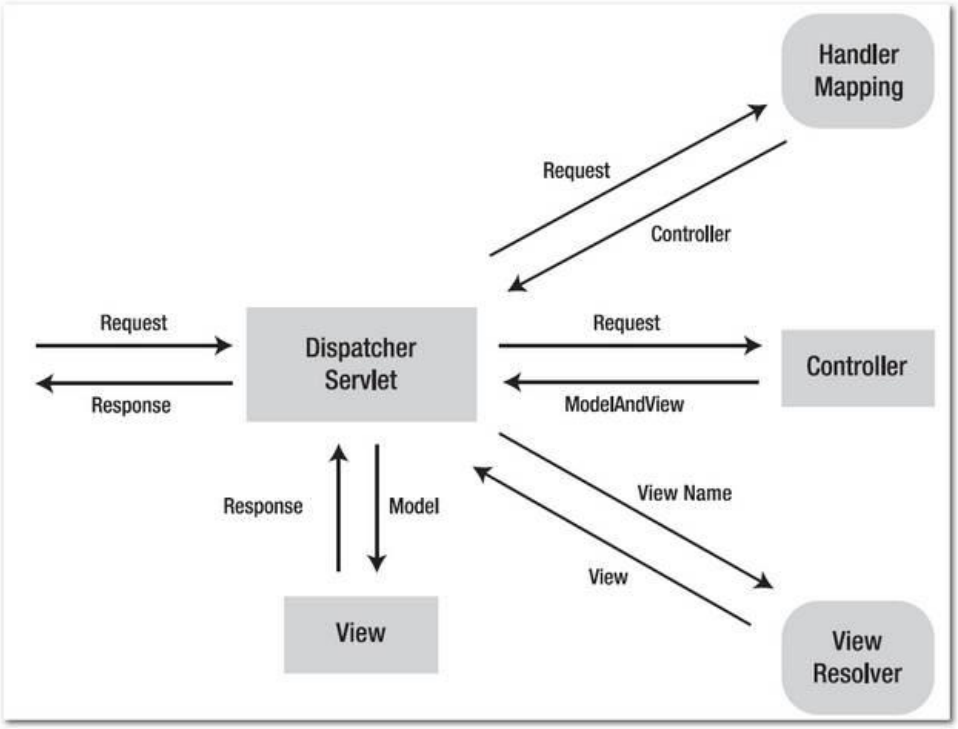


Figura 3 – Fluxo de execução da camada de apresentação

1. O **Dispatcher Servlet** recebe a requisição e delega para o **Handler Mapping**.
2. O **Handler Mapping** determina qual *controller* deve ser utilizado e o retorna.
3. O **Dispatcher Servlet** redireciona a requisição para o *controller* determinado.
4. O **Controller** realiza o processamento e retorna uma resposta.
5. O **Dispatcher Servlet** chama o **View Resolver** para determinar a *view* a ser renderizada.
6. O **View Resolver** determina qual *view* deve ser utilizada e a retorna.
7. O **Dispatcher Servlet passa** o modelo para popular a *view* retornada.
8. A **View** é renderizada para o usuário.

2.2.4.2.2 Controladores

Controladores disponibilizam acesso pelas views à lógica da aplicação. Eles interpretam as entradas do usuário e povoam objetos para serem apresentados nas views.

As requisições dos usuários são mapeadas para os controladores de acordo com a URL da requisição e seus parâmetros. Mais informações sobre este mapeamento na seção *Handler Mapping*.

2.2.4.2.2.1 Características Gerais

- Controladores são definidos através da anotação *@Controller*.
- Não é necessário, mas possível, estender ou utilizar outras classes.
- O framework reconhece as classes controladoras, procura por métodos anotados e reconhece os mapeamentos das requisições (*RequestMappings*).
- Ainda assim, controladores podem ser definidos explicitamente no contexto da aplicação web.

2.2.4.2.2.2 Mapeamento

O mapeamento das requisições para os controladores será feito de modo automático, onde a URL da requisição informará o controlador adequado e o seu respectivo método a ser invocado. Mais informações sobre o mapeamento de requisições para os controladores na seção *Handler Mapping*.

2.2.4.2.2.3 Anotações

No Spring, a configuração dos controladores é feita através de anotações. Diversas anotações são disponibilizadas para se definir os controladores e os respectivos comportamentos desejados. Entre elas, podemos encontrar as seguintes anotações, utilizadas para o tratamento das requisições dos usuários:

Annotation Types Summary

Annotation Type	Description
ControllerAdvice	Indicates the annotated class assists a "Controller".
CookieValue	Annotation which indicates that a method parameter should be bound to an HTTP cookie.
ExceptionHandler	Annotation for handling exceptions in specific handler classes and/or handler methods.
InitBinder	Annotation that identifies methods which initialize the `WebDataBinder` which will be used for populating command and
Mapping	Meta annotation that indicates a web mapping annotation.
MatrixVariable	Annotation which indicates that a method parameter should be bound to a name-value pair within a path segment.
ModelAttribute	Annotation that binds a method parameter or method return value to a named model attribute, exposed to a web view.
PathVariable	Annotation which indicates that a method parameter should be bound to a URI template variable.
RequestBody	Annotation indicating a method parameter should be bound to the body of the web request.
RequestHeader	Annotation which indicates that a method parameter should be bound to a web request header.
RequestMapping	Annotation for mapping web requests onto specific handler classes and/or handler methods.
RequestParam	Annotation which indicates that a method parameter should be bound to a web request parameter.
RequestPart	Annotation that can be used to associate the part of a "multipart/form-data" request with a method argument.
ResponseBody	Annotation that indicates a method return value should be bound to the web response body.
ResponseStatus	Marks a method or exception class with the status code and reason that should be returned.
RestController	A convenience annotation that is itself annotated with `@Controller` and `@ResponseBody`.
SessionAttributes	Annotation that indicates the session attributes that a specific handler uses.

Figura 4 – Anotações Spring para os controladores

Maiores Informações:

http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/package-summary.html

2.2.4.2.3 Serviços

No contexto do Spring, serviços REST são controladores com a característica de atender a requisições através do retorno de dados sem intermédio de uma interface de usuário. Tipicamente, serão invocados por aplicações ou interfaces (e.g Android) externas. A troca de dados, tanto os parâmetros como os dados de retorno, se dá através de um padrão específico, como XML ou JSON.

Abaixo, o exemplo de um controlador Spring que é um serviço REST:

```
01 @Controller
02 @RequestMapping("/rest/members")
03 public class MemberRestController
04 {
05     @Autowired
06     private MemberDao memberDao;
07
08     @RequestMapping(method=RequestMethod.GET, produces="application/json")
09     public @ResponseBody List<Member> listAllMembers()
10     {
11         return memberDao.findAllOrderedByName();
12     }
13
14     @RequestMapping(value="/{id}", method=RequestMethod.GET,
produces="application/json")
15     public @ResponseBody Member lookupMemberById(@PathVariable("id") Long id)
16     {
17         return memberDao.findById(id);
18     }
19 }
```

Código 31 – Controlador REST com Spring

A *Linha 01* mostra o uso da anotação Controller para definir a classe como um controlador do Spring.

A *Linha 02* define o URL do serviço através da anotação RequestMapping.

A *Linha 08* define o tipo de requisição e o formato de retorno através da anotação RequestMapping.

A *Linha 09* define que o retorno do método deve ser incluído no corpo da página.

A *Linhas 14 e 15* definem um parâmetro da requisição através do uso de *place holders* na anotação *RequestMapping* e do uso da anotação *PathVariable*.

Maiores informações:

http://en.wikipedia.org/wiki/Representational_state_transfer

2.2.4.2.4 Visões

As visões representam as páginas do sistema e são disponibilizadas para o usuário de acordo com a sua requisição. Através do Spring MVC, as visões são montadas pelo framework Tiles e povoadas e gerenciadas pelos controladores para serem disponibilizadas ao usuário.

2.2.4.2.4.1 Nomenclatura

Os nomes das *views* são definidos pela funcionalidade que cada *view* disponibiliza, juntamente com a subpasta onde se encontra.

2.2.4.2.4.2 Organização de Pastas

As *views* estão organizadas em subpastas da pasta *views* onde cada subpasta representa o conceito de negócio aos quais as *views* se aplicam e o nome da subpasta é definido pelo respectivo controlador.

2.2.4.2.4.3 Resolução das Views

As *views* são resolvidas através do retorno dos métodos dos controladores. Os métodos chamados devem retornar uma *String* que será utilizada para a resolução da *view*, de acordo com a configuração do *viewResolver* no arquivo de configuração **contextoWeb.xml** (veja seção View Resolver).

No retorno, os métodos chamados pelos controladores em uma requisição do usuário devem retornar o nome da *view* a ser exibida, incluindo a subpasta da pasta WEB-INF/views na qual a *view* se encontra.

O exemplo abaixo ilustra o retorno direto do nome de uma *view* a partir de um método de um controlador, para a resolução da *view*:

```
01 @RequestMapping
02 public String exibir(Model model) {
03     RegistroCompra registroCompra = new RegistroCompra();
04     model.addAttribute("registroCompra", registroCompra);
05 
06     Collection<UnidadeOperacional> unidades = servicosDeUnidadeOperacional
07                     .obterTodasUnidadesOperacionais();
08     model.addAttribute("unidades", getUnidadesMap(unidades));
09 
10     model.addAttribute("busca", false);
11 
12     return "/registroCompra/exibir";
13 }
```

Código 32 – Retorno direto de nome de view para resolução

A *Linha 12* mostra o retorno direto do nome de uma *view*, incluindo a subpasta em que se encontra.

Para detalhes da configuração, consulte a seção Tiles.

2.2.4.2.4.4 Tag Libraries

Tag libraries, ou *Taglibs*, fornecem um recurso de reutilização de código HTML para renderização de componentes nas *views*. Serão utilizadas as *taglibs* do Spring Framework, *taglibs* padrão JSP (JSTL) e a *taglib* do Tiles, listadas a seguir:

Tabela 3 - Tags utilizadas

Tecnologia	Taglib	Descrição
JSP	Core	Suporte a variáveis, controle de fluxo, gestão de URLS e outras funções.
JSP	Internationalization	Suporte a formatação de dados localizados (locais, números, datas).
Spring	Spring	Exibição de mensagens internacionalizáveis e de erros.
Spring	Form	Geração de formulários com diversos tipos de campos de entrada.
Tiles	Tiles Tags	Tags para montagem de layouts.
Pdsi	Tags Pdsi	Tags criadas com o foco no desenvolvimento de aplicações Ipiranga

Para serem utilizadas, as *taglibs* devem ser importadas nas *views*, de acordo com o exemplo a seguir:

```
01 <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
02 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
03 <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt"
04 <%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>
05 <%@ taglib uri="/WEB-INF/tld/pdsi-tag.tld" prefix="pdsi" %>
```

Código 33 – Exemplo de importação de Taglibs

Abaixo, exemplos de utilização de *tags* das *taglibs*:

```
01 <!-- Formulario com informacoes para serem enviadas para o controlador. *Deve
ser utilizada a tag do spring  -->
02 <form:form modelAttribute="zonaVenda" cssClass="apps-form" action="buscar">
03     <fieldset>
04           <!--   Linha do primeiro campo do formulario  -->
05           <div class="row inline-label margin-bottom-10px">
06                 <!-- Texto sobre o campo. *Deve ser utilizada a tag fmt -->
07                 <div class="span2">
08                       <label for="cdZonven">
09                             <fmt:message key="zonaVenda.form.cdZonven"/>
10                       </label>
11                 </div>
12                 <!--   Campo. *Deve ser utilizada a tag do spring  -->
13                 <div class="span3 tablet-span5">
14                       <form:input path="cdZonven" />
15                 </div>
16           </div>
17           <!--   Linha do segundo campo do formulario  -->
18           <div class="row inline-label margin-bottom-10px">
19                 <div class="span2">
20                       <label for="gerenciavenda">
21                             <fmt:message key="zonaVenda.form.cdGv"/>
22                       </label>
23                 </div>
24                 <div class="span3 tablet-span5">
25                       <form:select path="gerenciaVenda.cdGv"
items="${gerencias}"/>
26                 </div>
27           </div>
28     </fieldset>
29 </form:form>
```

Código 34 – Exemplo de uso de taglibs para geração de HTML e texto

A *Linha 02* mostra o uso da *tag form:form* para geração de um formulário HTML.

A *Linha 09* mostra o uso da *tag fmt:message* para gerar uma mensagem na página.

A *Linha 14* mostra o uso da *tag form:input* para definir um campo de entrada de dados na página.

A *Linha 25* mostra o uso da *tag form:select* para gerar uma *tag* HTML *select*.

```
01 <c:if test="${not empty mensagem}">
02     <div class="row">
03           <div class="span12">
04                 <p class="msg msg-success close">
05                       <span>${mensagem}</span>
06                 </p>
07           </div>
08     </div>
09 </c:if>
```

Código 35 – Exemplo de uso de taglibs para controle de fluxo

A *Linha 01* mostra o uso *da tag <c:if>* para controle de fluxo.

Maiores informações:

- http://docs.oracle.com/javaee/5/tutorial/doc/bnakc.html
- http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/view.html
- http://tiles.apache.org/2.0/framework/tiles-jsp/tagreference.html

2.2.4.2.4.5 Expression Language (EL)

Expression Language, ou EL, é um mecanismo importante que permite a comunicação da camada de visão com a lógica da aplicação; Em outras palavras, permite utilizar uma inteligência na apresentação das informações nas páginas JPS, permitindo que dados armazenados nos *beans* sejam acessados e manipulados pelas *views*. Expressões EL são utilizadas entre **${}** (dólar, abre e fecha chaves).

Abaixo, um exemplo para recuperação de valores de atributos de *beans* através da página JSP e avaliação de expressões lógicas:

```
01 <c:if test="${not empty mensagem}">
02     <div class="row">
03           <div class="span12">
04                 <p class="msg msg-success close">
05                       <span>${mensagem}</span>
06                 </p>
07           </div>
08     </div>
09 </c:if>
```

Código 36 – Exemplo de uso de Expression Language

A *Linha 01* mostra o uso de *Expression Language* para avaliação de expressões lógicas.

A *Linha 05* mostra o uso de *Expression Language* para o acesso a informações enviadas pelo controlador.

Maiores informações:

http://docs.oracle.com/javaee/6/tutorial/doc/gjddd.html

2.2.4.2.4.6 Internacionalização

Os recursos de internacionalização permitem que os rótulos e mensagens estejam organizados e que sejam exibidos de acordo com o local do usuário. Para os detalhes da configuração, veja a seção *Internacionalização*.

2.2.4.2.4.7 Passagem de Parâmetros nas Requisições

Ao receber requisições no contexto Java, o Spring realiza o *data binding* dos parâmetros passados nas requisições através da anotação *ModelAttribute*. Objetos declarados com esta anotação têm seus atributos povoados automaticamente pelo framework. Dessa forma, parâmetros são extraídos da requisição e mapeados para os atributos dos objetos anotados.

As classes utilizadas para *binding* de dados devem, obrigatoriamente, possuir métodos *getters* e *setters* para os atributos, pois tais métodos são utilizados pelo Spring para a realização do *binding*.

No exemplo abaixo, além de povoar o objeto zonaVenda no contexto Java através dos parâmetros da requisição, dados são passados para a *view* através da classe *Model*, do Spring:

```
01 @RequestMapping
02 public String buscar(@ModelAttribute("zonaVenda") ZonaVenda zonaVenda,
03            BindingResult result, Model model) {
04     if (zonaVenda != null) {
05            String cdGv = null;
06            if (zonaVenda.getGerenciaVenda() != null) {
07                   cdGv = zonaVenda.getGerenciaVenda().getCdGv();
08            }
09            Collection<ZonaVenda> zonas = servicoPontoVenda.obterZonasVendaPor(
10                         zonaVenda.getCdZonven(), cdGv);
11            model.addAttribute("resultados", zonas);
12     }
13     Collection<GerenciaVenda> gerencias = servicoGerenciaVenda
14                   .obterTodasAsGerenciasVenda();
15     model.addAttribute("gerencias", getGerenciasMap(gerencias));
16
17     return "/zonaVenda/listar";
18 }
```

Código 37 – Passagem de parâmetros nas requisições para o controlador

A *Linha 02* mostra o uso da anotação *ModelAttribute*, que povoa os atributos do objeto de acordo com os parâmetros na requisição.

A *Linha 11* adiciona um atributo ao modelo para ser utilizado pela *view*.

2.2.4.2.4.8 Passagem de Objetos para a View

Nos contexto Java, os objetos a serem disponibilizados na *view* são definidos através da classe *Model*, do Spring. No exemplo abaixo, o objeto *model* será passado para a *view* para ser apresentado para o usuário:

```
01 @RequestMapping
02 public String exibir(Model model) {
03     ZonaVenda zonaVenda = new ZonaVenda();
04     model.addAttribute("zonaVenda", zonaVenda);
05     Collection<GerenciaVenda> gerencias = servicoGerenciaVenda
                   .obterTodasAsGerenciasVenda();
06     model.addAttribute("gerencias", getGerenciasMap(gerencias));
07
08     model.addAttribute("busca", false);
09
10     ZonaVenda zonaVendaIncl = new ZonaVenda();
11     model.addAttribute("zonaVendaIncl", zonaVendaIncl);
12
13     return "/zonaVenda/listar";
14 }
```

Código 38 – Passagem de objetos para a view

As *Linhas 04, 07, 09 e 12* povoam atributos do modelo a ser passado para a *view*.

2.2.4.2.4.9 Apresentação de Objetos na View

No contexto JSP, as *views* têm acesso aos objetos definidos pelos controladores através do nome dos mesmos e de seus atributos. O mapeamento é realizado a partir dos nomes dos atributos do objeto para a propriedade "path" das *tags* HTML / JSP nas páginas, de acordo com o exemplo abaixo:

```
01 <form:form modelAttribute="zonaVenda" cssClass="apps-form" action="buscar">
02     <fieldset>
03           <!--   Linha do primeiro campo do formulario  -->
04           <div class="row inline-label margin-bottom-10px">
05           <!--   Texto sobre o campo. *Deve ser utilizada a tag fmt  -->
06                 <div class="span2">
07                       <label for="cdZonven"><fmt:message
key="zonaVenda.form.cdZonven"/></label>
08                 </div>
09                 <!--   Campo. *Deve ser utilizada a tag do spring  -->
10                 <div class="span3 tablet-span5">
11                       <form:input path="cdZonven" />
12                 </div>
13           </div>
14           <!--   Linha do segundo campo do formulario  -->
15           <div class="row inline-label margin-bottom-10px">
16                 <div class="span2">
17                       <label for="gerenciavenda"><fmt:message
key="zonaVenda.form.cdGv"/></label>
18                 </div>
19                 <div class="span3 tablet-span5">
20                       <form:select path="gerenciaVenda.cdGv"
items="${gerencias}"/>
21                 </div>
22           </div>
23     </fieldset>
24 </form:form>
```

Código 39 – Apresentação de objetos na view

A *Linhas 11 e 20* acessam os objetos povoados pelo controlador, antes de chamar a *view*.

2.2.4.2.5 Tiles

O framework Apache Tiles foi definido como responsável por resolver a *view*. A resolução através da classe *ViewResolver* do Spring utiliza a classe *TilesView* do Tiles. Os nomes resolvidos pelo Tiles são passados para o *RequestDispatcher*, para renderização da *view* resolvida. Este mecanismo permite que o Spring seja integrado a diferentes tecnologias de view, caso necessário.

Através do padrão *Composite*, o Tiles permite a definição de fragmentos reutilizáveis que são montados em *runtime* servindo páginas completas padronizadas. Os fragmentos são montados através de layouts.

Os seguintes fragmentos foram definidos:

Tabela 4 - Fragmentos do layout

Fragmento	Descrição
header	Representa o cabeçalho das páginas
menu	Representa o menu das páginas
body	Representa o conteúdo principal das páginas
scripts	Representa a área onde devem ser inseridos os scripts
title	Representa o título da aplicação

Os seguintes layouts foram definidos:

Tabela 5 - Templates do sistema

Layout	Descrição
layout-default.jsp	Layout default do sistema
layout-modal.jsp	Layout para Modais utilizados no projeto

Os layouts estão localizados na pasta layouts. Os fragmentos podem ser encontrados de diferentes formas. Os fragmentos que representam subpáginas (ex.: header) estão localizados na pasta layouts/tiles do conteúdo web do projeto. Os fragmentos que representam texto (ex.: title) estão localizados no próprio arquivo de layout. Os fragmentos que variam o conteúdo de acordo com a página estão localizados em subseções das próprias páginas (ex.: body).

Maiores informações:

https://tiles.apache.org/

Para a utilização deste framework, as páginas devem ser anotadas com taglibs indicando o que representa o trecho de código dentro de um *template* do sistema.

```
01 <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"
%>
02 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
03 <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>
04 <%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
05
06 <!--     Definição do layout a ser incluído na página -->
07 <tiles:insertTemplate template="/layouts/layout-default.jsp">
08
09     <tiles:putAttribute name="body">
10         <!--     Conteúdo da página -->
11         <div class="container">
12             ...
13         </div>
14     </tiles:putAttribute>
15
16     <tiles:putAttribute name="scripts">
17         <script type="text/javascript">
18             ...
19         </script>
20         <script type="text/javascript"
src="../assets/js/jquery.equalheights.js"></script>
21     </tiles:putAttribute>
22
23 </tiles:insertTemplate>
```

Código 40 - Exemplo de utilização do Tiles

A *Linha 04* apresenta a importação da *taglib* do Tiles na página.

A *Linha 07* mostra o uso da *tag* <tiles:insertTemplate>, que especifica o *template* a ser utilizado para a montagem do layout.

As *Linhas de 09 a 14* mostram o uso da tag <tiles:putAttribute>, que contém o trecho de código que será inserido no *template* na parte reservada ao *body*.

As *Linhas de 16 a 21* mostram o uso da tag <tiles:putAttribute>, que contém o trecho de código que será inserido no *template* na parte reservada aos *scripts*.

O código abaixo (Código 41) exemplifica o arquivo de *template* utilizado na Linha 07 do Código 40. As *tags* <tiles:insertAtribute> presentes nas *linhas 19, 20, 21 e 31* definem os fragmentos de código que podem ser inseridos no *template*. Caso seja necessário podem ser definidos valores padrões para estes fragmentos.

```
01 <%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
02 <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>
03 <html>
04     <head>
05         <title><tiles:getAsString name="title" defaultValue="PDSI
Blueprint Java"/></title>
06         <meta http-equiv="Content-Type" content="text/html;charset=ISO-
8859-1" />
07         <meta name="viewport" content="width=device-width, initial-
scale=1, maximum-scale=1" />
08         <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1"
/>
09         <link rel="stylesheet" type="text/css" href="<c:url
value='/assets/css/fonts.css'/>" />
10         ...
11         <link rel="shortcut icon" type="text/css" href="<c:url
value='/assets/icons/favicon.ico'/>" />
12         <!--[if lt IE 9]>
13         <script type="text/javascript" src="<c:url
value='/assets/js/html5shiv.js'/>"></script>
14         ...
15         <![endif]-->
16     </head>
17     <body class="overflow-y-scroll">
18         <div class="apps-content">
19             <tiles:insertAttribute name="header"
defaultValue="/layouts/tiles/header.jsp"/>
20             <tiles:insertAttribute name="menu"
defaultValue="/layouts/tiles/menu.jsp"/>
21             <tiles:insertAttribute name="body" />
22         </div>
23            <script type="text/javascript">
24             $(document).ready(function(){
25
26             });
27         </script>
28         <script type="text/javascript" src="<c:url
value='/assets/js/jquery.min.js'/>"></script>
29         ...
30         <script type="text/javascript" src="<c:url
value='/assets/js/apps.js'/>"></script>
31         <tiles:insertAttribute name="scripts" defaultValue="" />
32     </body>
33 </html>
```

Código 41 - Exemplo de template do Tiles

2.2.4.2.6 Validação

Será utilizado o Spring Bean Validator para validação de dados, através da injeção de um *Validator* no *Controller* e da execução de chamadas de validação no *Validator* injetado.

Além disso, o mecanismo de validação apresenta as seguintes características:

- Oferece suporte à validação no *request* através da anotação @Valid

- Permite utilizar a *tag form:errors* em uma JSP para exibir as mensagens de validação
- As mensagens de validação podem estar um arquivo de internacionalização
- Spring possui uma interface *Validator* utilizada para validar objetos;
- A interface *Validator* faz uso de objetos do tipo *Errors* para que possa reportar erros encontrados no objeto que será validado;
- Vantagem de implementação sem necessidade de uso de anotações no modelo.

Maiores informações:

http://docs.spring.io/spring/docs/current/spring-framework-reference/html/validation.html

2.2.4.2.6.1 Implementação de Validadores

Os validadores do Spring devem implementar a interface *Validator*.

Maiores informações:

http://docs.spring.io/spring/docs/3.2.5.RELEASE/spring-framework-reference/html/validation.html

No exemplo abaixo, na classe *RelatorioValidator*, o método *validate* verifica se o campo código da Gerência Comercial está vazio.

```
01 /**
02 * Classe de validação que implementa o Validator do Spring com o objetivo de
03 * validar os campos.
04 *
05 *
06 */
07 @Component
08 public class RelatorioValidator implements Validator {
09
10     @Override
11     public boolean supports(Class<?> classe) {
12
13           return GerenciaComercial.class.isAssignableFrom(classe);
14
15     }
16
17     @Override
18     public void validate(Object target, Errors errors) {
19
20           ValidationUtils.rejectIfEmptyOrWhitespace(errors, "cdGc",
21                       "gerenciaComercial.ValorNulo");
22
23     }
24 }
```

Código 42 – Exemplo de implementação de validador

A *Linha 08* mostra a implementação da interface *Validator*, do Spring.

A *Linha 18* mostra implementação do método *validate*.

A *Linha 20* executa a regra de validação.

No exemplo abaixo, uma instância do validador é injetada quando necessário.

```
01 @Autowired
02 private RelatorioValidator relatorioValidator;
```

Código 43 – Injeção de validadores

A *Linha 01* mostra o uso da anotação *Autowired* para a injeção do validador.

O exemplo abaixo, da classe *RelatorioController*, ilustra uma chamada ao método *validate* do validador.

```
01 @RequestMapping(params = "arquivo")
02 public String download(
03           @ModelAttribute("gerenciaComercial") GerenciaComercial
gerenciaComercial,
04           BindingResult result, Model model, HttpServletResponse response) {
05
06     relatorioValidator.validate(gerenciaComercial, result);
07
08     model.addAttribute("gerenciaComercial", gerenciaComercial);
09
10     if (!result.hasErrors()) {
11           File arquivoDownload;
12           try {
13                 filePath = servico
14
       .gerarRelatorioDeComprasPorClientesDa(gerenciaComercial
15                                             .getCdGc());
16                 arquivoDownload = new File(filePath);
17
18                 response.setContentType("application/xls");
19                 response.setContentLength(new Long(arquivoDownload.length())
20                             .intValue());
21                 response.setHeader("Content-Disposition",
22                             "attachment; filename= Relatorios de
Gastos.xls");
23
24                 FileCopyUtils.copy(new FileInputStream(arquivoDownload),
25                             response.getOutputStream());
26           } catch (IOException e) {
27                 Log.error(this.getClass().toString(), "download",
e.getMessage(), e);
28           } catch (ExcecaoDeServico e) {
29                 Log.error(this.getClass().toString(), "download",
e.getMessage(), e);
30           }
31     }
32     return "/relatorios/relatorios-gastos-clientes";
33 }
```

Código 44 – Chamada ao método validate

A *Linha 06* executa a chamada ao método *validate*.

A *Linha 10* verifica o resultado da validação. Caso o objeto *result* não contenha erros gerados pela validação, o controlador executa a lógica.

2.2.4.2.6.2 Exibição de Mensagens na View

O exemplo abaixo ilustra a utilização da tag form:errors.

```
01 <form:form modelAttribute="gerenciaComercial" cssClass="apps-form"
action="downloadEmMemoria${cdGc}">
02     <div class="row inline-label margin-left-10px margin-bottom-10px">
03           <div>
04                 <form:errors path="cdGc" cssClass="msg msg-error ico-alert-inv
close" />
05           </div>
06     </div>
07     <fieldset>
08           <div class="row inline-label margin-bottom-10px">
09                 <div class="span2">
10                       <label for="cdGc"><fmt:message
key="relatoriosGastosCliente.form.cdGc"/></label>
11                 </div>
12                 <div class="span3 tablet-span5">
13                 <form:input path="cdGc" type="text" />
14
15                 </div>
16           </div>
```

Código 45 – Exemplo de uso de tag para exibição de mensagens de erro

A *Linha 04* ilustra o uso da *tag form:errors* para exibir mensagens de erro geradas pela execução do método *validate*, no controlador.

2.2.4.2.6.3 Mensagens por Tipo de Exceção

Mensagens padronizadas para exceções específicas podem ser definidas através dos códigos das mensagens de erro.

Esse mecanismo é possível através da interface *MessageCodesResolver* que constrói os códigos das mensagens de erro a serem povoados no objetos *ObjectError* e *FieldError* de acordo com as exceções levantadas. Quando um erro é encontrado, a implementação da interface *MessageCodesResolver* criará os seguintes códigos de erro, na seguinte ordem e prioridade:

1. Código da mensagem + "." + nome do objeto + "." + nome do campo
2. Código da mensagem + "." + nome do campo
3. Código da mensagem + "." + tipo do campo
4. Código da mensagem

As código construídos representam o escopo da mensagem da seguinte forma:

- O caso 1 abrange o campo do objeto informado;
- O caso 2 abrange os campos com o nome informado, em qualquer objeto;
- O caso 3 abrange o tipo de dado em qualquer campo ou objeto;
- O caso 4 abrange qualquer situação em que a exceção seja levantada.

Por exemplo, a exceção do tipo *TypeMismatchException*, que representa o erro ocorrido quando um tipo de dado incompatível é informado (e.g. caracteres no lugar de números), pode ser utilizada para se definir mensagens específicas para este caso.

O código abaixo ilustra a utilização de mensagem de erro padrão para tipos de dados incompatíveis, contemplando situações de erro mais genéricas e situações de erro mais específicas:

```
01 # Mensagens sobre Ponto de Venda
02 ...
03  typeMismatch.pontoVenda.cdPtov  =  O  Código  do  Ponto  de  Venda  deve  ser  um
número.
04 typeMismatch.cdPtov = O Código informado deve ser um número.
05 typeMismatch.int = O valor informado deve ser um número.
06 typeMismatch = O valor informado é inválido.
```

Código 46 – Utilização de exceções para a definição de códigos de mensagens de erro

Figura 5 – Tela com exibição de mensagens de erro

Maiores informações:

- http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/validation/MessageCodesResolver.html
- http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/validation/DefaultMessageCodesResolver.html

2.2.4.2.7 Ajax

O processamento Ajax utiliza JavaScript e jQuery sendo chamados a partir de páginas JSP. O arquivo *assets/js/appsModal.js* contém funções que realizam a comunicação com o servidor, compreendendo requisições e respostas. Desta forma, este arquivo deve ser incluído nas páginas que devam utilizar Ajax, através da *tag* HTML <script>. Esta tag deve informar a localização do arquivo JS mencionado através da *tag* JSTL <c:url>, conforme o exemplo abaixo:

```
01 <script
02    type="text/javascript"
03    src="<c:url value='/assets/js/appsModal.js'/>"
04 />
```

Código 47 – Inclusão do arquivo Javascript appsModal.ja para processamento Ajax

A *Linha 01* ilustra o uso da *tag* HTML <script> para inclusão de arquivo de funções JavaScript, utilizado no processamento Ajax.

A *Linha 03* ilustra o uso da *tag* JSP <c:url> dentro da *tag* HTML para inclusão dinâmica do URL absoluto da aplicação.

As seções abaixo apresentam os principais mecanismos utilizados para a comunicação com o servidor em termos de funções Javascript e chamadas e recebimentos de retorno nas páginas JSP.

2.2.4.2.7.1 Principais Funções

Abaixo, tabela com a descrição sucinta das principais funções utilizadas nos processamentos Ajax, contidas no arquivo do arquivo appsModal.js:

Tabela 6 - Métodos de requisição ajax

Função	Descrição
onSuccessDefault	Função default de *callback* executada no caso de retorno Ajax com sucesso do servidor. Informada como parâmetro nas chamadas $.ajax.
MakeOpcoesAjax	Função auxiliar para montagem das opções das chamadas Ajax. Retorna um *array* de parâmetros, sendo eles: *url* da chamada, tipo da chamada (*get* ou *post*) e função de *callback* para o caso de sucesso.
AjaxPost	Função que realiza as chamadas Ajax passando como parâmetros a *url* que deve ser chamada, o id do formulário de onde será obtidos os dados e a função de tratamento do retorno da requisição com sucesso.
`AjaxPostData`	Função que realiza as chamadas Ajax passando como parâmetros a *url* que deve ser chamada, os dados a serem enviados para o servidor e a função de tratamento do retorno da requisição com sucesso.
AjaxPostDefault	Função que realiza as chamadas Ajax passando como parâmetros a *url* que deve ser chamada e o id do formulário de onde será obtidos os dados. Esta função utiliza como *callback* de sucesso a função default **`onSuccessDefault().`**
AjaxPostDataDefault	Função que realiza as chamadas Ajax passando como parâmetros a *url* que deve ser chamada e os dados a serem enviados para o servidor. Esta função utiliza como *callback* de sucesso a função default **`onSuccessDefault().`**
`AjaxPostDataSemLoading`	Função que realiza as chamadas Ajax passando como parâmetros a *url* que deve ser chamada, os dados a serem enviados para o servidor e a função que tratará o retorno da requisição. Utilizável para fazer requisição em background.
abrirEditar	Função que carrega os dados para edição através de chamada ao servidor e exibe o formulário com os dados preenchidos, para serem editados ou consultados pelo usuário. Recebe como parâmetro o contexto (elemento que ativou a ação), a *url* que deve ser chamada e os dados (geralmente o id do objeto), utilizados para recuperar o objeto que deseja-se editar.

2.2.4.2.7.2 Chamadas Ajax

Para a realização de uma chamada Ajax, é utilizado o método $.ajax(), do jQuery, dentro do código Javascript, como explicado nas próximas seções.

2.2.4.2.7.2.1 Parâmetros das Chamadas

Para executar uma chamada Ajax através da função $.ajax(), é necessário informar os seguintes argumentos:

Tabela 7 - Argumentos da chamada Ajax

Argumento	Descrição
url	Endereço da *action* do controlador que será chamado (e.g. salvar)
type	Método HTTP a ser utilizado na chamada (e.g. POST ou GET).
success	Função *callback* que será executada no caso de um retorno com sucesso do servidor.
error	Função *callback* que será executada no caso de um retorno com erro do servidor.
data	Dados a serem enviados ao servidor como parâmetro da chamada (e.g. dados preenchidos no formulário de entrada).

A função MakeOpcoesAjax() contém uma implementação reutilizável para montagem das opções das chamadas Ajax, sendo necessário passar como parâmetros o URL, o método HTTP da requisição (GET ou POST) e a função de *callback* de sucesso, conforme código abaixo:

```
01 MakeOpcoesAjax = function(url, submiType, callback){
02    var opcoes = {
03            url: url,
04            type: submiType,
05            success: callback,
06            error: function(xhr, textStatus, errorThrown){
07                   alert("General Exception");
08              },
09        };
10        return opcoes;
11 };
```

Código 48 – Função reutilizável para a montagem das opções das chamadas Ajax.

A *Linha 01* mostra a assinatura da função, que deve receber o URL, o método HTTP da requisição e a função de *callback*.

As *Linhas de 02 a 08* representam a montagem do *array* de opções para a chamada Ajax.

A *Linha 06* mostra a opção para a função de *callback* com erro, que será padrão, apenas exibindo uma mensagem genérica de erro.

Percebe-se que esta função não preenche o parâmetro "data" que deve conter os dados a serem enviados para o servidor. Isso se deve ao fato de que os dados podem ser obtidos de diferentes fontes, dependendo do caso. Por isso, após a chamada a essa função, é necessário preencher os dados a serem enviados ao servidor.

Abaixo, a implementação de método que executa a chamada Ajax fazendo uso da função MakeOpcoesAjax() e preenchendo o parâmetro "data" em sequência:

```
01 /* Requisição Post obtendo os dados diretamente do formulário*/
02 AjaxPostDefault = function(url, formId, idModal){
03          _idModal = idModal;
04          startHourGlass({});
05          var data = $('#'+formId).serialize();
06          var opcoes = MakeOpcoesAjax(url,'POST', onSuccessDefault);
07          opcoes.data = data;
08          $.ajax(opcoes);
09 };
```

Código 49 – Uso da função makeOpcoesAjax() e preenchimento do parâmetro "data".

A *Linha 05* recupera os dados informados no formulário através do método serialize() do objeto do formulário *$('#'+formId)*.

A *Linha 06* chama a função makeOpcoesAjax() passando o URL a ser chamado, o método HTTP da chamada, no caso, "POST" e a função reutilizável de *callback* de sucesso, *onSuccessDefault*.

A *Linha 07* preenche o parâmetro "data" que deve conter os dados a serem enviados para o servidor. Nesse caso, os dados foram obtidos diretamente do formulário preenchido pelo usuário, no código da *Linha 05*, explicado anteriormente. Uma outra forma de recuperação dos dados do usuário a serem enviados para o servidor é através de uma *querystring* concatenada diretamente no URL do servidor.

2.2.4.2.7.3 Funções de Callback

Funções de *callback* são passadas como argumento para as chamadas Ajax para serem executadas após o recebimento de resposta do servidor. Tipicamente pode haver uma função *callback* para retorno com sucesso e outra função *callback* para retorno com falha. No caso de sucesso, normalmente existe a conversão dos dados retornados que podem estar, por exemplo, nos formatos XML ou JSON. No caso de falha, normalmente são exibidas as mensagens de erro retornadas pelo servidor. Tanto no caso de sucesso como no caso de falha, o objeto que representa a requisição (*XmlHttpRequest*) possui um atributo para o status do retorno (e.g. Status 500 - Internal Server Error ou Status 200 - OK).

O código abaixo representa uma função *callback* para retorno com sucesso após chamada ao servidor:

```
01 var _idModal = null;
02 onSuccessDefault = function(retorno, xhr){
03    //Tratar o retorno da chamada
04     //Converte uma string em objeto JSON
05     var result = $.parseJSON(retorno);
06     if(result != null && result.STATUS == "ERROR"){
07       showErrorMessage(result.mensagem, _idModal);
08       stopHourGlass(); //deve vir antes para que o background
permaneça ativo
09       atualizaOpacityBackground(_idModal);
10     }else if(result != null && result.STATUS == "SUCCESS"){
11        stopHourGlass();
12          showSuccessMessage(result.mensagem);
13          $('#'+_idModal).hide();
14     }
15 };
```

Código 50 – Função callback de retorno com sucesso do servidor após chamada Ajax.

A *Linha 02* recebe os parâmetros sendo que o primeiro parâmetro, "retorno", representa os dados retornados pelo servidor (e.g. dados de um ponto de venda). O segundo parâmetro é o objeto Javascript com os demais dados da resposta (*XmlHttpRequest*).

A *Linha 05* converte o retorno que é uma *string* nos moldes de um objeto JSON, para um objeto JSON através do método do jQuery $.parseJSON.

As demais linhas realizam exibição de mensagens com o resultado da conversão e atualizam a exibição da janela.

As funções de *callback* devem ser passadas como parâmetro nas chamadas Ajax, conforme o exemplo abaixo.

```
01 MakeOpcoesAjax = function(url, submiType, callback){
02    var opcoes = {
03            url: url,
04            type: submiType,
05            success: callback,
06            error: function(xhr, textStatus, errorThrown){
07                   alert("General Exception");
08              },
09        };
10        return opcoes;
11 };
```

Código 51 – Passagem da função de callback como parâmetro.

O código acima declara um *array* de variáveis que representam a configuração da chamada Ajax.

As *Linhas 05 e 06* especificam as funções a serem executadas no caso de retorno com sucesso e erro, respectivamente, da chamada Ajax que for executada utilizando esta configuração.

2.2.4.2.7.4 Chamadas nas Páginas JSP

As chamadas Ajax realizadas nas páginas JSP são chamadas Javascript simples, sem distinção de outras chamadas. Tipicamente são implementadas no evento "*onclick*" dos componentes.

O código abaixo ilustra uma chamada Javascript na tag HTML <a>:

```
01 <a href="javascript:void(0)"
02    onclick="editar('${zonaVenda.cdZonven}', this)"
03    data-target="edit_modal"
04    title="${editTooltip}">
05 </a>
```

Código 52 – Chamada Javascript em tag HTMLde link utilizando o evento onclick.

A *Linha 02* especifica o código Javascript a ser executado quando for realizado um click no componente. Será executado o método editar() que receberá como parâmetro o valor do atributo *cdZonven* do objeto *zonaVenda* e o próprio componente que gerou a ação. O método editar chamado do javascript monta o dado a ser enviado para o servidor e então realiza a chamada ajax:

```
01 editar = function(idObjeto, context){
02    var data = {cdZonven: idObjeto};
03    abrirEditar(context, "editar", data);
04 };
```

Código 53 - Tratamento do dado antes de realizar a chamda ajax

A seguir, outro exemplo de utilização:

```
01 <a href="javascript:void(0)"
02    onclick="saveForm()">
03          <fmt:message key="sistema.botao.salvar"/>
04 </a>
```

Código 54 – Chamada Javascript em tag HTMLde link utilizando o evento onclick.

A *Linha 02* especifica o código Javascript a ser executado quando for realizado um click no componente. Será executado o método saveForm(), implementado, nesse caso, na própria página JSP.

```
01 saveForm = function(){
02    var zonaVendaSerialize = $("#formIncluir").serialize();
03     var pontosDeVenda = getValuesFromSelectedList("selected_list");
04
05     var temFoto = $("#selectedfile")[0].files[0];
06     if(temFoto == null){
07          temFoto = "temFoto=false";
08     }else{
09          temFoto = "temFoto=true";
10     }
11
12     var data = zonaVendaSerialize+"&"+pontosDeVenda+"&"+temFoto;
13     var url = "salvar";
14
15    AjaxPostDataDefault(url, data, "include_modal");
16 };
```

Código 55 - Obtém os dados necessários e só então faz a chamada Ajax.

Da *linha 2 à 10* são obtidos os valores do formulário de id "*formIncluir*". A *linha 12* concatena todos os dados obtidos, esses dados estão em formato *querystring* "variável=valor&variavel2=valor2&variavel3" e etc. A *linha 15* realiza a chamada Ajax.

2.2.4.2.7.5 Passagem e Retorno de Dados

A passagem de dados para o servidor utilizando Ajax ocorre em dois formatos:

- Utilizando o formato *querystring*: "variável=valor&variavel2=valor2&variavel3"

```
01 var data = zonaVendaSerialize+"&"+pontosDeVenda+"&"+temFoto;
```

Código 56 - formato querystring

- Utilizando objeto JSON

```
01 var data = {cdZonven: idObjeto};
```

Código 57 - Objeto JSON

O retorno acontece também de duas formas:

Retornando uma *string* em um formato que represente um JSON, para que seja possível fazer a conversão em objeto JSON no lado cliente. Para criarmos a *string* no formato JSON no servidor, utilizamos a classe JSONObject do pacote *org.json.simple.JSONObject*.

```
01 JSONObject retorno = new JSONObject();
02 if (result.hasErrors()) {
03
04    retorno.put("STATUS", "ERROR");
05    retorno.put("mensagem", "Erro ao validar objeto.");
06
07    return retorno.toJSONString();
08 }
```

Código 58 - Retorno de string no formato JSON

O string no formato JSON acima ficaria:

{“STATUS”: “ERROR”, “mensagem”: “Erro ao validar objeto.”}

Retornando uma página “.jsp” acoplando-a a página principal via javascript.

```
01 public String editar(@ModelAttribute("zonaVenda") ZonaVenda zonaVenda,
02          BindingResult result, String cdZonven, HttpServletRequest
request,
03          Model model) throws ExcecaoDeVisaoModal {
04
05    ZonaVenda zonaVendaEditar = servicoPontoVenda
06                .obterZonaVendaComGerencias(cdZonven);
07    if (zonaVendaEditar == null ) {
08          model.addAttribute("mensagemErro", "A zona de venda não possui
ponto de venda associada.");
09          return "/excecao/erroInesperadoModal";
10    }
11
12    model.addAttribute("zonaVenda", zonaVendaEditar);
13    inicializarValoresEdicao(model,cdZonven);
14    model.addAttribute("pontosVenda", zonaVendaEditar.getPontosVenda());
15
16    return "/zonaVendaModal/editar";
17 }
```

Código 59 - Retorno da página JSP

```
01 abrirEditar = function(context, url, data){
02    $("#edit_modal #js-corpo").empty();
03    stack = 20000;
04    startHourGlass({});
05    $('#blockUI').css({zIndex: ++stack}).show();
06    $.get(url,data, function(result){
07          $("#edit_modal #js-corpo").append(result);
08          $modal = $(context).attr('data-target');
09          $('#' + $modal).css({zIndex: ++stack}).fadeIn('fast');
10          stopHourGlassMin();
11    });
12 };
```

Código 60 - Script para carregamento da página retornada

A *linha 07* acopla a página retornada do servidor à página principal.

2.2.4.2.7.6 Upload de Foto

Para a realização do upload de foto é utilizado os métodos abaixo:

Tabela 8 - Métodos para upload de foto

Método	Descrição
uploadForm	Obtém alguns dados necessários antes de chamar a requisição Ajax.
doProgressUpload	Exibe e manipula a barra de loading, exibida enquanto é feita o upload da foto.
doUpload	Método que faz a requisição Ajax.

A particularidade do método *doUpload* é a utilização do objeto *FormData* implementado em versões mais recentes do Firefox e Chrome e no IE10. Este objeto permite a criação de um conjunto de pares chave/valor que representam os campos de formulários e seus valores, usa o mesmo formato que um formulário usaria se o *enconding type* fosse "multipart/form-data". Também aplicamos a propriedade "*processData*" e "*contentType*" para "false", desta forma o dado não será transformado para o formato *querystring* e não haverá alteração no cabeçalho da requisição respectivamente.

```
01 doUpload: function(){
02     var oMyForm = new FormData();
03     var id = $(window.target).parent().parent().next().attr("id");
04     var file = $('#'+id)[0].files[0];
05     oMyForm.append("file", file);
06
07     $(window.target).parent().next().show();
08     UploadFoto.doProgressUpload(25);
09
10     $.ajax({
11         url: 'uploadFile',
12         data: oMyForm,
13         dataType: 'text',
14         processData: false,
15         contentType: false,
16         type: 'POST',
17         success: function(data){
18           setTimeout(function(){
19             clearTimeout(ptimer);
20             $(window.target).parent().next().find('> .bar').width(100 +
'%').html(100 + '%');
21             $(window.target).parent().next().hide();
22
$(window.target).parent().addClass('done').empty().html(data);
23           }, 2500);
24         },
25         error: function(xhr, textStatus, errorThrown){
26             //Tratar erro inesperado
27         },
28         complete: function(xhr, textStatus){
29             //faz alguma coisa ao completar a requisiÃ§Ã£o
30         }
31     });
32 }
```

Código 61 - Método doUpload

Do lado servidor utiliza-se o objeto *MultipartHttpServletRequest* do pacote *org.springframework.web.multipart.MultipartHttpServletRequest* para obter a foto da requisição na action uploadFile. Ainda é necessário ter a configuração no contextoWeb.xml conforme abaixo:

```
01 <beans ...>
02     ...
03     <bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver"
/>
04 </beans>
```

Código 62 - Configuração para upload de arquivo

A *action uploadFile* armazena os bytes da foto em um objeto e retorna uma string que representa um componente html para imagem, cuja *url* é uma outra *action* que obterá a foto armazenada no servidor e a exibe na tela.

2.2.4.3 Configuração

2.2.4.3.1 web.xml

O arquivo web.xml deve ser configurado para disponibilizar a aplicação Java EE no servidor de aplicações.

2.2.4.3.1.1 DispatcherServlet

A DispatcherServlet deve ser configurada no arquivo web.xml para disponibilizar os serviços do Spring, de acordo com o exemplo abaixo:

```
01 <servlet>
02     <servlet-name>springmvc</servlet-name>
03     <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
04     <init-param>
05           <param-name>contextConfigLocation</param-name>
06           <param-value>/WEB-INF/contextoWeb.xml</param-value>
07     </init-param>
08     <load-on-startup>1</load-on-startup>
09 </servlet>
10 <servlet-mapping>
11     <servlet-name>springmvc</servlet-name>
12     <url-pattern>/</url-pattern>
13 </servlet-mapping>
```

Código 63 – Configuração da DispatcherServlet no web.xml

A *Linha 03* mostra a configuração da DispatcherServlet, do Spring.

A *Linha 10* mostra o mapeamento de todos os URLS para a Servlet do Spring.

2.2.4.3.2 contextoWeb.xml

O arquivo contextoWeb.xml deve ser configurado para definir as características da aplicação web através do Spring framework.

2.2.4.3.2.1 Handler Mapping

Esta seção descreve as configurações dos mapeamentos dos URLs para os controladores adequados de acordo com a requisição. Estão descritos aqui os seguintes mapeamentos:

2.2.4.3.2.1.1 Mapeamento Automático

O mapeamento automático permite definir um controlador específico e um método específico a ser invocado de acordo com o URL da requisição. Neste caso, o controlador representa um resource (recurso) e o método, uma action (ação). Abaixo, o padrão de URLs para requisições com mapeamento automático:

```
01 http://host/aplicação/{resource}/{action}
Ex:
02 http://ipiranga/WAPDSIBlueprintJava/zonaVenda/listar
```

Código 64 - Exemplo de URL

De acordo com o padrão de URL acima, o {resource} representa o nome do controlador que deve ser invocado e a {action} representa o nome do método a ser invocado. O método invocado deve retornar uma *String* contendo o caminho para a *view* a ser exibida. Este caminho deve ser mapeado para a página JSP correspondente no projeto.

Abaixo, a configuração necessária para a resolução automática de views no arquivo contextoWeb.xml.

```
01 <!-- (PDSI - COMENTÁRIO)
02 Habilita a funcionalidade de mapeamento padrão com o nome do controlador e nome da ação
03 através da url: /[resource]/[action].
04 Exemplo: Função "criar" no "PontoVendaController" será mapeado através da url:
05 "/pontoVenda/criar"
06  -->
07 <bean id="classnameControllerMappings"
class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"
08       p:order="2"
09       p:interceptors-ref="localeChangeInterceptor"
10       p:caseSensitive="true" >
11     <property name="defaultHandler">
12            <bean class="org.springframework.web.servlet.mvc.UrlFilenameViewController"
/>
13     </property>
14 </bean>
```

Código 65 – Configuração para resolução automática de views

A *Linha 07* declara o bean para mapeamento automático.

A *Linha 12* define o bean para resolução de views automaticamente.

No exemplo abaixo, o URL http://ipiranga.com.br/PDSIBluePrintJava/zonaVenda/buscar invocará o método buscar do controlador ZonaVendaController. Além disso, de acordo com a implementação do controlador abaixo, serão passados como parâmetro na requisição valores que correspondem aos atributos do objeto ZonaVenda.

```
01 @RequestMapping
02 public String buscar(@ModelAttribute("zonaVenda") ZonaVenda zonaVenda,
03           BindingResult result, Model model) {
04     if (zonaVenda != null) {
05           String cdGv = null;
06           if (zonaVenda.getGerenciaVenda() != null) {
07                 cdGv = zonaVenda.getGerenciaVenda().getCdGv();
08           }
09           Collection<ZonaVenda> zonas = servicoPontoVenda.obterZonasVendaPor(
10                       zonaVenda.getCdZonven(), cdGv);
11           model.addAttribute("resultados", zonas);
12     }
13     Collection<GerenciaVenda> gerencias = servicoGerenciaVenda
14                 .obterTodasAsGerenciasVenda();
15     model.addAttribute("gerencias", getGerenciasMap(gerencias));
16
17     return "/zonaVenda/listar";
18 }
```

Código 66 – Exemplo de requisição com resolução automática e passagem de parâmetros

2.2.4.3.2.1.2 Mapeamento para Endereços Especiais

Vários padrões de URL podem ser definidos para servir a endereços que não sigam o padrão definido pelo sistema. Esta abordagem pode ser utilizada, por exemplo, para página principal da aplicação, de acordo com a seguinte configuração:

```
01 <!-- (PDSI - COMENTÁRIO)
02 Habilita a funcionalidade de mapeamento de uma URL para um controlador
específico.
03 Neste caso, este mapeador associa possíveis URLs iniciais do site a um
controlador
04 para tratamento da página inicial.
05  -->
06 <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
07     <property name="order" value="1" />
08     <property name="mappings">
09           <value>
10                 /index=homeController
11                 /index.html=homeController
12                 /index.jsp=homeController
13                 /=homeController
14           </value>
15     </property>
16 </bean>
```

Código 67 – Configuração do mapeamento para a página principal

A *Linha 06* declara o bean para mapeamento direto de urls.

A *Linhas de 08 a 15* declaram os mapeamentos que devem ser resolvidos pelo bean.

2.2.4.3.2.1.3 Mapeamento de Arquivos Estáticos

Arquivos estáticos são servidos por Servlet responsável por prover tais arquivos e pode ser configurada para utilizar cache. O exemplo abaixo ilustra a configuração necessária para servir arquivos estáticos:

```
01 <!-- (PDSI - COMENTÁRIO)
02 Servlet exclusivo do Spring MVC para prover recursos do projeto. Em geral é
utilizado
03 para prover arquivos estáticos como Javascritp, CSS, Imagens, ...
04 -->
05 <mvc:resources mapping="/assets/**" location="/assets/" order="0"/>
```

Código 68 – Configuração de mapeamento para servir arquivos estáticos

A *Linha 05* define, por wildcards, os arquivos estáticos a serem servidos e sua localização no conteúdo web do projeto.

2.2.4.3.2.2 View Resolver

Para realizar a resolução de views através do Tiles, as configurações abaixo devem ser realizadas:

2.2.4.3.2.2.1 Tiles Configurer

O trecho abaixo realiza a configuração do Tiles para trabalhar com o Spring.

```
01 <bean id="tilesConfigurer"
class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
02   <property name="definitions">
03          <list>
04                <value>/layouts/tiles.xml</value>
05          </list>
06   </property>
07 </bean>
```

Código 69 – Configuração do Tiles no Spring

A *Linha 04* define o arquivo de configuração do Tiles a ser utilizado.

2.2.4.3.2.2.2 View Resolver

O trecho abaixo realiza a configuração do padrão para resolução dos nomes das views pelo tiles:

```
01    <!-- (PDSI - COMENTÁRIO)
02    View Resolver da aplicação. Essa classe é responsável por
identificar a visão que
03    deve ser retornada para o usuário. O Spring MVC recomenda a
colocação das páginas
04    dentro da pasta WEB-INF para proteger as páginas do acesso direto
via URLs
05    digitadas manualmente.
06     -->
07    <bean id="viewResolver"
08    class= "org.springframework.web.servlet.view.
InternalResourceViewResolver">
09          <property name="prefix" value="/WEB-INF/views/" />
10          <property name="suffix" value=".jsp" />
11          <property name="order" value="3" />
12    </bean>
```

Código 70 – Configuração do ViewResolver do Tiles no Spring

A *Linha 08* define a classe para resolução da view.

A *Linha 09* define o prefixo que será concatenado à String retornada pelos controladores.

A *Linha 10* define o sufixo que será concatenado à String retornada pelos controladores.

De acordo com a configuração no trecho de código acima, a resolução será a seguinte:

/WEB-INF/views/<STRING_RETORNADA_PELO_CONTROLADOR>.jsp

2.2.4.3.2.3 Internacionalização

As seguintes configurações são necessárias para o projeto:

2.2.4.3.2.3.1 Arquivos de Mensagens e Rótulos

Os arquivos de mensagens e rótulos contêm as mensagens e rótulos a serem exibidos para o usuário, sendo um arquivo por idioma.

O padrão de nomenclatura destes arquivos é propriedades é:

- **mensagens.properties:** Mensagens reutilizáveis.
- **mensagens_<SIGLA DO PAIS>.properties:** Mensagens específicas do idioma utilizado.

O exemplo abaixo ilustra como configurar os arquivos de mensagens e rótulos internacionalizáveis:

```
01 <!-- (PDSI - COMENTÁRIO)
02 Define o arquivo de resource bundle do projeto.
03 -->
04 <bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource"
05      p:basenames="resources/mensagens" />
```

Código 71 – Configuração dos arquivos de rótulos e mensagens internacionalizáveis

A *Linha 04* declara o *bean* para gerenciamento de mensagens e rótulos e sua localização no conteúdo java.

2.2.4.3.2.3.2 Interceptador de Locale

Classe que permite definir um local específico a ser utilizado através de um parâmetro de requisição (*request parameter*). Dessa forma, as formatações internacionalizáveis podem ser independentes do local do usuário. Por exemplo, o usuário poderia selecionar qual idioma prefere utilizar.

O exemplo abaixo ilustra como definir um parâmetro de requisição a ser utilizado para se definir o *locale* desejado.

```
01 <!-- (PDSI - COMENTÁRIO)
02 Habilita uso do interceptador necessário à identificação do locale
03 -->
04 <bean id="localeChangeInterceptor"
05     class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"
06           p:paramName="locale" />
```

Código 72 – Configuração do interceptador de locale

A *Linha 04* declara o interceptador que fará a leitura do *locale* a partir de parâmetro na requisição.

2.2.4.3.2.3.3 Armazenamento do Locale

O *locale* do usuário deve estar acessível para a camada de apresentação, para a exibição de mensagens no idioma correto e para a formatação adequada de datas e números.

O exemplo abaixo ilustra como definir o armazenamento do *locale* do usuário.

```
01 <!-- (PDSI - COMENTÁRIO)
02 Define onde é armazenado o locale do usuário.
03 -->
04 <bean id="localeResolver"
05     class="org.springframework.web.servlet.i18n.SessionLocaleResolver" />
```

Código 73 – Configuração do armazenamento do locale do usuário

A *Linha 04* configura o armazenamento do *locale* na sessão do usuário.

2.2.4.4 Tutoriais

Esta seção apresenta algumas funcionalidades aplicadas no sistema que são recorrentes no desenvolvimento de sistemas.

2.2.4.4.1 Download de Arquivos (Spring FileCopyUtils)

A classe *FileCopyUtils* contém métodos auxiliares utilizados para cópias de *streams*. No projeto a classe foi utilizada na funcionalidade de gerar o relatório de compras. Na *linha 24* é utilizado o método copy que faz a cópia de um *inputstream* para o *outputstream* do *HttpServletResponse*.

Uma vantagem da utilização desta classe é que todos os métodos da mesma fecham os *streams* automaticamente quando terminam a cópia do arquivo.

```
01 @RequestMapping(params = "arquivo")
02 public String download(
03            @ModelAttribute("gerenciaComercial") GerenciaComercial
gerenciaComercial,
04            BindingResult result, Model model, HttpServletResponse response) {
05
06     relatorioValidator.validate(gerenciaComercial, result);
07
08     model.addAttribute("gerenciaComercial", gerenciaComercial);
09
10     if (!result.hasErrors()) {
11            File arquivoDownload;
12            try {
13                   filePath = servico
14
       .gerarRelatorioDeComprasPorClientesDa(gerenciaComercial
15                                                .getCdGc());
16                   arquivoDownload = new File(filePath);
17
18                   response.setContentType("application/xls");
19                   response.setContentLength(new Long(arquivoDownload.length())
20                                .intValue());
21                   response.setHeader("Content-Disposition",
22                                "attachment; filename= Relatorios de
Gastos.xls");
23
24                   FileCopyUtils.copy(new FileInputStream(arquivoDownload),
25                                response.getOutputStream());
26            } catch (IOException e) {
27                   Log.error(this.getClass().toString(), "download",
e.getMessage(), e);
28            } catch (ExcecaoDeServico e) {
29                   Log.error(this.getClass().toString(), "download",
e.getMessage(), e);
30            }
31     }
32     return "/relatorios/relatorios-gastos-clientes";
33 }
```

Código 74 – Exemplo de controlador para download de arquivos

Maiores Informações:

http://docs.spring.io/spring/docs/1.2.9/api/org/springframework/util/FileCopyUtils.html

2.2.4.4.2 Tag Select

A tag select renderiza um elemento HTML do tipo select possibilitando que se faça a vinculação com objetos através da construção de um Map composto por uma chave e um valor. No exemplo abaixo podemos verificar a criação do map que será adicionado ao atributo do tipo select.

```
01 private Map<String, String> getGerenciasMap(
02           Collection<GerenciaComercial> gerencias) {
03     Map<String, String> gerenciasMap = new HashMap<String, String>();
04     if (gerencias == null)
05           return gerenciasMap;
06     for (GerenciaComercial g : gerencias) {
07           gerenciasMap.put(g.getCdGc(), g.getNmGc());
08     }
09     return gerenciasMap;
10 }
```

Código 75 – Exemplo de Map para utilização com a tag select

2.2.4.4.3 Upload de Arquivos xls e xlsx

Para a inclusão de um registro de compra a partir do upload de um arquivo excel(formatos xls ou xlsx), foi utilizada a biblioteca Apache POI (versão 3.9) responsável pela lógica de leitura dos arquivos enviados.

Com o intuito de ser possível validar arquivos tanto no formato xls quanto no formato xslx, foi utilizada a classe WorkbookFactory a qual é responsável por criar o tipo apropriado de workbook (*HSSFWorkbook* ou *XSSFWorkbook*), fazendo a detecção automática do formato inserido conforme podemos verificar no exemplo abaixo na *linha 9*. Da mesma forma, também é necessário utilizar a classe Sheet, responsável por instanciar a planilha na versão correta.

```
01 try {
02
03     File arquivoUpload = new File(caminhoArquivoUpload);
04     SimpleDateFormat format = new SimpleDateFormat("ddMMyy");
05     format.setLenient(false);
06
07     FileInputStream file = new FileInputStream(arquivoUpload);
08
09     Workbook workbook = WorkbookFactory.create(file);
10     Sheet sheet = workbook.getSheetAt(0);
11
12     List<List<String>> atributos = new ArrayList<List<String>>();
13
14     List<String> linha = new ArrayList<String>();
15     // Iterate through each rows from first sheet
16     Iterator<Row> rowIterator = sheet.iterator();
17
18     while (rowIterator.hasNext()) {
19           Row row = rowIterator.next();
20           // For each row, iterate through each columns
21           Iterator<Cell> cellIterator = row.cellIterator();
22
23           while (cellIterator.hasNext()) {
24
25                 Cell cell = cellIterator.next();
```

Código 76 – Exemplo de upload de arquivos xls e xlsx

2.2.4.4.4 Spring MultipartFile

O Spring possui suporte nativo para lidar com uploads de arquivo em aplicações web . Este suporte é feito através de objetos do tipo *MultipartFile*.

Por padrão, o Spring não faz o tratamento de arquivos *multpart*, sendo necessário configurar a utilização de um *MultipartResolver*. Após a configuração ter sido realizada, é necessário indicar nas páginas jsp se o *enctype* do *form* é do tipo *"multipart/form-data"*, conforme pode ser visto na *Linha 01* do Código 77.

O Spring inspeciona todas as requisições para saber se são do tipo *multipart* e, caso seja encontrado, a mesma poderá ser tratada no controlador de maneira que o mesmo receba como parâmetro um *ModelAttribute* de uma classe do tipo *MultipartFile*.

```
01 <form:form method="POST" enctype="multipart/form-data"
02     modelAttribute="uploadArquivo" cssClass="apps-form"
03     action="upload">
04     <fieldset>
05           <div class="row margin-bottom-10px">
06                 <div class="span12">
07                       <label for="arquivo"><fmt:message
key="registroCompra.incluir.abrirArquivo"/></label>
08                       <form:input path="arquivo" type="file" value="Escolher
Arquivo" />
09                 </div>
10           </div>
11           <input type="submit" value="Upload" class="btn btn-prime ico-pos-
right ico-upload" "/>
12     </fieldset>
```

Código 77 – Exemplo de uso de view para upload de arquivos

```
01 @RequestMapping
02 public String upload(
03           @ModelAttribute("uploadArquivo") UploadArquivo uploadArquivo,
04           HttpServletRequest request, HttpServletResponse response) {
05
06     InputStream inputStream = null;
07     OutputStream outputStream = null;
08     MultipartFile multipartFile = uploadArquivo.getArquivo();
09
10     String fileName = "";
11
12     if (multipartFile != null) {
13           fileName = multipartFile.getOriginalFilename();
14           try {
15                 inputStream = multipartFile.getInputStream();
```

Código 78 – Exemplo de uso do Spring MultipartFile

```
01 outputStream = new FileOutputStream(arquivo);
02
03 FileCopyUtils.copy(inputStream, outputStream);
04
05 servicosDeRegistroCompra
06           .incluirRegistrosCompraAPartirDeArquivo(caminhoArquivo);
```

Código 79 – Exemplo de implementação de gravação de arquivo

2.2.4.4.5 Exportação de Arquivo em Memória

A exportação do relatório de clientes pode ser feita de duas maneiras: Criando um arquivo Excel numa pasta do projeto e colocando-o na response do *HttpServletResponse*. A implementação representada no Código 80 realiza a exportação em memória como alternativa à implementação citada anteriormente.

Para realizar a exportação do arquivo em memória, é necessário criar um vetor de bytes que será populado com o retorno de um método da camada de serviço. Para utilizar o Apache POI para gerar o arquivo, é preciso utilizar um *ByteArrayOutputStream* que será posteriormente convertido em um vetor de bytes, como pode ser visto no Código 82. Após popular este vetor, basta utilizar o método *copy* da classe *FileCopyUtils* do Spring para transferir os bytes para a response do *HttpServletResponse*.

```
01 byte[] array = null;
02 relatorioValidator.validate(gerenciaComercial, result);
03
04 model.addAttribute("gerenciaComercial", gerenciaComercial);
05
06 if (!result.hasErrors()) {
07
08     try {
09           array = servico
10
       .gerarRelatorioEmMemoriaDeComprasPorClientesDa(gerenciaComercial
11                                      .getCdGc());
12
13           response.setContentType("application/xls");
14           response.setHeader("Content-Disposition",
15                       "attachment; filename= Relatorios de Gastos.xls");
16
17           FileCopyUtils.copy(array, response.getOutputStream());
```

Código 80 – Exemplo de implementação de exportação de arquivo em memória

```
01 public byte[] gerarRelatorioEmMemoriaDeComprasPorClientesDa(String cdGc)
02           throws ExcecaoDeServico {
03
04     ByteArrayOutputStream bos = null;
05
06     try {
07           GerenciaComercial gerenciaComercial = todasAsGerenciasComerciais
08                       .obterPorId(cdGc);
09
10           HSSFWorkbook workbook = new HSSFWorkbook();
11           HSSFSheet planilha = workbook.createSheet("Registro de Compras");
```

Código 81 – Exemplo de código para criação de planilhas

```
01 bos = new ByteArrayOutputStream();
02
03 workbook.write(bos);
04
05 return bos.toByteArray();
```

Código 82 – Exemplo de uso de vetor de bytes para exportação de planilhas

2.2.4.4.6 Busca por Múltiplos Campos

Numa das telas de busca foi implementado um exemplo com a busca por múltiplos campos. O campo Produto foi utilizado para realizar a consulta de um registro de compra através do código ou descrição do produto como pode ser visto na Código 83.

Para tratar este caso na aplicação é necessário criar um VO - explicado no item Value Objects – que é responsável por receber o valor inserido na camada de visão para a busca de um registro compra por produto associado. O exemplo da implementação deste VO é visto no código Código 83 abaixo.

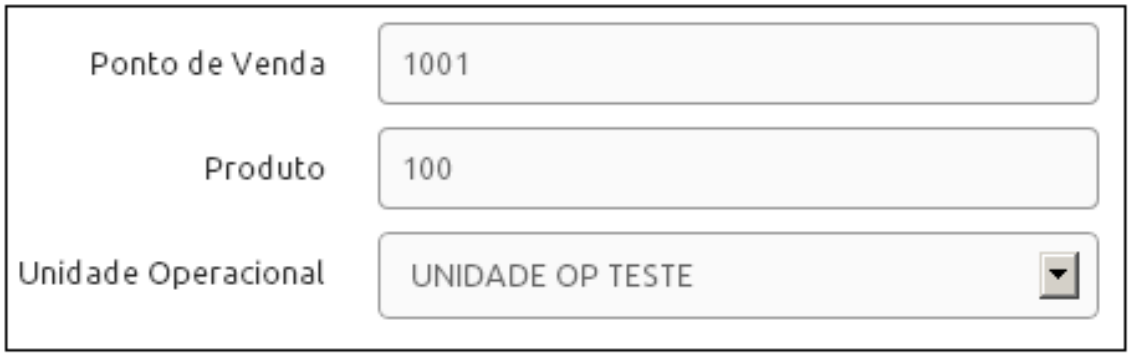


Figura 6 – Tela com campos para consulta por código ou por descrição

```
01 public class RegistroCompraVo {
02
03     private Integer cdPtov;
04
05     private String produto;
06
07     private String cdUnidOper;
08
09     public RegistroCompraVo() {
10            super();
11     }
12
13     public RegistroCompraVo(Integer cdPtov, String produto,
14                   String cdUnidOper) {
15            super();
16            this.cdPtov = cdPtov;
17            this.produto = produto;
18            this.cdUnidOper = cdUnidOper;
19     }
20
21     public String getProduto() {
22            return produto;
23     }
24
25     public void setProduto(String produto) {
26            this.produto = produto;
27     }
```

Código 83 – Exemplo de implementação de VO

Para o tratamento na camada de dados, se faz necessário o uso de *Criteria* que é uma interface provida pelo JPA que representa a query.

A utilização de *criteria* é vista no Código 84 abaixo. O predicado é utilizado para montar mais de uma restrição dentro da query, pois no uso de *criteria* só é permitido adicionar uma cláusula *WHERE*. No exemplo, foram adicionadas duas cláusulas que permitem a busca por mais de um campo.

```
01 if (produto != null && !produto.isEmpty()) {
02     if (isNumber(produto)) {
03           Predicate pred1 = cb.equal(
04                       registroCompra.get("produto").get("cdProd"), produto);
05           Predicate pred2 = cb.equal(
06                       registroCompra.get("produto").get("dsProd"), produto);
07           predicates.add(cb.or(pred1, pred2));
08     } else {
09           predicates.add(cb.equal(registroCompra.get("produto").get("dsProd"),
produto));
10     }
11 }
```

Código 84 – Exemplo de uso de criteria para filtrar por múltiplos campos

Maiores Informações sobre a implementação de Criteria:

http://docs.jboss.org/hibernate/orm/3.3/reference/pt-BR/html/querycriteria.html

2.2.4.4.7 Integração com o SAA (saa.properties)

Para tratar a integração da aplicação com o SAA (Sistema de Autorização e Autenticação), o framework necessita que a configuração do SAA seja definida no arquivo saa.properties. O objetivo desse arquivo é promover a leitura textual dos códigos de acesso das funções do SAA (leitura essa utilizada na consulta de acesso) e mapear URLs do sistema para suas permissões.

Para criar um "de para" dos códigos das funções do SAA basta criar o arquivo saa.properties no diretório src/resources e inserir como conteúdo a palavra reservada **funcao**.NOME_DA_FUNCAO=[código da função no SAA], conforme pode ser visto no exemplo abaixo:

```
01 funcao.zonaVenda.exibicao=2

02 funcao.zonaVenda.cadastro=3
```

*Essa leitura de funções será utilizada na construção de menu, liberação de botões, filtros de acesso e até mesmo consultas dentro do source da aplicação.

Mapeamento de URI

Como já foi dito, a outra característica dessa configuração é a possibilidade de mapear funções de acesso para determinadas URIs do sistema. Ou seja, caso o seu sistema possua acessos restritos de acordo com o perfil do usuário logado, além de inibir a exibição do item de menu usando o menu dinâmico você também pode configurar o framework para validar as requisições por acesso, basta inserir o interceptado do SAA no arquivo contextoWeb.xml (conforme pode ser visto abaixo).

*contextoWeb.xml

```
01 <mvc:interceptors>
02 <bean
lass="ipp.aci.pdsiframework.visao.interceptador.InterceptadorDoSaa" />
03 </mvc:interceptors>
```

Agora no arquivo saa.properties, utilizando a palavra reservada uri_funcao.URI[sem o contexto da aplicação]=FUNCAO[sem indicar a palavra reservada funcao.], o interceptador passa a mapear a URI somente disponibilizando o acesso das requisições de usuários devidamente autorizados.

```
01 uri_funcao.zonaVenda/exibir=zonaVenda.exibicao
02 uri_funcao.zonaVenda=funcao.zonaVenda.cadastro
```

Linha 1 - Indica que usuários detentores da função zonaVenda.exibicao podem acessar a requisição de exibição (consulta) das Zonas de Venda.

Linha 2 - Indica que usuários detentores da função zonaVenda.cadastro podem acessar qualquer a requisição dentro do contexto de Zona de Venda.

Toda a URI não indicada no mapeamento do saa.properties não terá o acesso controlado.

Toda a URI que possuir acesso restrito deve ser indicada na configuração do saa.properties.

A home da aplicação (uri /) nunca terá a restrição de acesso controlada, indicamos o uso dessa tela para login da aplicação ou para uma tela de descrição da solução.

2.2.4.4.8 Menu Dinâmico

Para facilitar a escrita e manutenção do componente de menu, agilizando a integração com o SAA, a biblioteca do PDSI possui uma estrutura JSTL constituída de quatro tags MENU, MENUITEM, SUBMENU E SUBMENUITEM.

```
1.  <pdsi:menu styleId="apps-menu" styleClass="tablet-menu3 mobile-menu1">
2.
3.      <pdsi:menuItem label="sistema.menu.inicio" link="/"
    functionName="inicio"/>
4.
5.      <pdsi:submenu label="sistema.menu.cadastros" functionName="cadastro
    cadastro.zonaVenda">
6.          <pdsi:submenuItem label="sistema.menu.zonaVenda"
    link="/zonaVenda/exibir" functionName="cadastro.zonaVenda" />
7.          <pdsi:submenuItem label="sistema.menu.zonaVendaModal"
    link="/zonaVendaModal/exibir"/>
8.      </pdsi:submenu>
9.
10.     <pdsi:submenu label="sistema.menu.propostaComercial"
    functionName="propostaComercial">
11.         <pdsi:submenuItem label="sistema.menu.proporCondicoes"
    link="/condicaoComercialProposta/exibir"/>
12.     </pdsi:submenu>
13.
14. </pdsi:menu>
```

Linha 01 - Indica o início da marcação do menu, nela devemos indicar a propriedade styleId="apps-menu" e styleClass="tablet-menu3 mobile-menu1" para o correto funcionamento do padrão visual. A marcação Menu deve possuir componentes do tipo menuItem e submenu.

Linha 03 - Indica um item de menu que não possui a característica drop-down, ou seja é um link direto do menu. A marcação menuItem não deve possuir componentes.

Linha 05 - Indica um item de menu que possui a característica drop-down, ou seja pode ter um link direto, mas também conseguirá abrigar outros links. A marcação de submenu deve possuir componentes do tipo submenuitem ou outro submenu

Linha 06 - Indica um item de submenu que possui a característica de link direto. A marcação de submenuitem deve possuir componentes do tipo submenuitem não deve possuir componentes.

Atributos das tags

functionName - Atributo que indica o nome da função mapeada no arquivo saa.properties para identificação do código da função cadastrada no SAA. É utilizado para verificar se o usuário possui permissão à função indicada, caso não tenha o item não é exibido no menu.

*O atributo functionName suporta mais de uma função indicada conforme pode ser observado na linha 05, onde o item de menu será exibido caso o usuário possua acesso à função cadastro ou cadastro.zonaVenda.

label - Indica a key mapeada no arquivo de internacionalização (mensagens.properties) referente ao texto do item de menu.

link - Como o nome já diz, indica link referente ao item de menu.

2.2.4.4.9 Utilizando o utilitário de usuário

Para a total integração entre o sistema e os artefatos do Framework (Menu Dinâmico, Tags de Controle de Acesso, Interceptador de Acesso do SAA), existe uma classe utilitária chamada UsuarioUtil. Esse utilitário tem a função de centralizar toda a inteligência necessária para o controle do usuário na sessão da aplicação.

Obtendo o mnemônico do usuário logado

```
01 String codigoUsuario = UsuarioUtil.getCodigoUsuarioLogado();
```

O exemplo acima exibe a maneira de utilizar o UsuarioUtil para obter o código do usuário (cd_usuario) inserido na sessão.

Obtendo a instância do usuário logado.

```
01 IUsuario usuario = UsuarioUtil.getUsuarioLogado();
```

O exemplo acima exibe a maneira de utilizar o UsuarioUtil para obter o usuário inserido na sessão. Repare que o retorno do método é a interface IUsuário (também contida no Framework).

Verificando se o usuário logado possui permissão à função especificada

```
01 if (UsuarioUtil.possuiPermissaoAcesso("cadastro.custo"))
```

O exemplo acima exibe a maneira de utilizar o UsuarioUtil para verificar se o usuário logado possui permissão de acesso a uma determinada função. Perceba que a String "cadastro.custo" é uma indicação do mapeamento do saa.properties

Removendo o usuário logado da sessão.

```
01 UsuarioUtil.removeUsuarioLogado();
```

O exemplo acima exibe a maneira de utilizar o UsuarioUtil para elimicar o registro do usuário na sessão.

Disponibilizando o usuário na sessão para acesso desta classe utilitária.

```
01 //A classe usuário é a implementação necessário para o Sistema da
interface IUsuario do Framework

02 Usuario usuario = new Usuario();

03 //Inserindo dados do usuário obtidos pelo SAA ou Base de Dados. De
acordo com a sua aplicação

04 UsuarioUtil.setUsuarioLogado(usuario);
```

O exemplo acima exibe a maneira de utilizar o UsuarioUtil para inserir um usuário na sessão.

*Observe que o Framework não possui uma classe default para a utilização dos componentes de acesso, assim os sistemas podem ter liberdade para construir entidades ou VOs de Usuário de acordo com a necessidade do sistema. No entanto**, é necessário que toda a classe referente a uma Entidade ou VO usuário implemente a interface IUsuario.**

3 Aplicação PDSI Blueprint Java

A aplicação *WAPDSIBlueprintJava* é um sistema que foi desenvolvido para exemplificar a Arquitetura Padrão Ipiranga. Esse projeto aborda questões importantes que ocorrem com frequência durante o desenvolvimento de sistemas. Esta seção tem o objetivo contextualizar o projeto, de modo a, facilitar o entendimento do mesmo. Além disso, os exemplos apresentados neste documento estão baseados nos artefatos produzidos na aplicação.

ATENÇÃO, esta é uma aplicação de exemplo. Não é recomendado o "copy and paste" de toda a aplicação para a geração de novas aplicações. O código da aplicação deve ser lido e entendido.

As partes críticas do sistema estão comentadas no código fonte para facilitar o entendimento do mesmo.

ATENÇÃO, os comentários explicativos do framework Java devem ser removidos dos novos projetos. Os comentários com (PDSI - COMENTÁRIO) não devem estar presentes em sistemas de ambientes de produção da Ipiranga.

3.1 Conteúdo da Camada de Apresentação

Esta seção fornece uma visão do conteúdo da camada de apresentação e descreve como este conteúdo está organizado. A camada de apresentação está dividida em dois tipos de conteúdo:

- Conteúdo Java
- Conteúdo Web

3.1.1 Conteúdo Java

O conteúdo Java é composto pelo pacote Java src/.../visao.

Este pacote contém os *beans* responsáveis pela lógica da camada de interface e pela comunicação com a camada de serviços.

Os *beans* podem ser:

- Modelos que representam conceitos de negócio.
- Validadores, utilizados para validar entradas do usuário.
- Controladores, responsáveis por gerenciar o fluxo da interação.
- Outros *beans*.

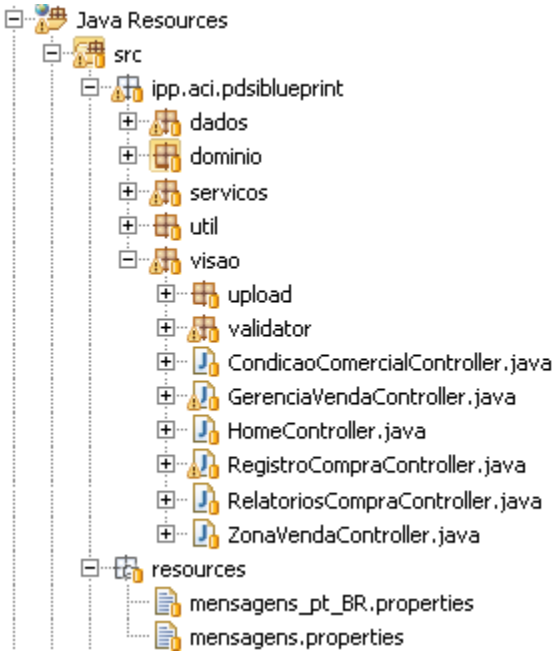


Figura 7 – Conteúdo Java da camada de apresentação

3.1.2 Conteúdo Web

O conteúdo Web é composto por arquivos diversos para montagem das páginas e configurações do sistema.

Tais arquivos devem ser organizados nas seguintes pastas, de acordo com a sua finalidade:

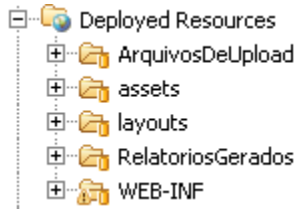


Figura 8 – Conteúdo web da camada de apresentação

Cada uma das pastas é explicada a seguir.

3.1.2.1 Arquivos de Upload

A pasta **ArquivosDeUpload** contém arquivos que foram carregados pelo usuário através da função de upload do sistema.

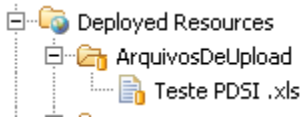


Figura 9 – Pasta ArquivosDeUpload

3.1.2.2 Arquivos Estáticos

A pasta **assets** contém arquivos JavaScript, arquivos de estilo, imagens e ícones, utilizados nos layouts e páginas criados.

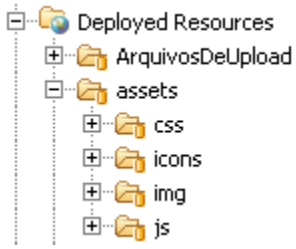


Figura 10 – Pasta assets

3.1.2.3 Layouts

A pasta **layouts** contém *templates* JSP reutilizáveis para a montagem de diferentes layouts de acordo com a necessidade, permitindo a criação de conjuntos consistentes de páginas.

Também nesta pasta se encontra o arquivo tiles.xml, que representa um arquivo de configuração obrigatório do framework.

A pasta **layouts/tiles** contém *templates* JSP reutilizáveis que representam os componentes utilizados pelos layouts, como cabeçalhos e menus.

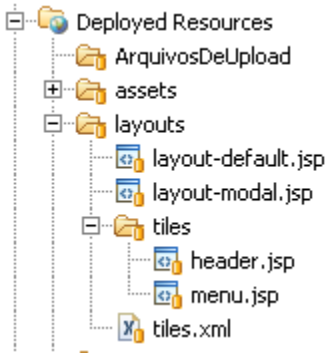


Figura 11 – Pasta Layouts

3.1.2.4 Relatórios Gerados

A pasta **RelatoriosGerados** contém arquivos temporários que correspondem aos relatórios gerados pelo usuário.

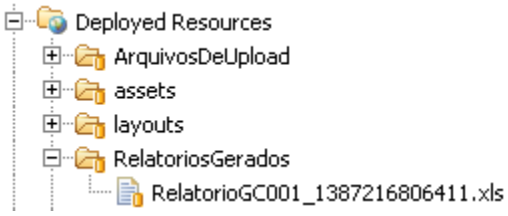


Figura 12 – Pasta RelatoriosGerados

3.1.2.5 Arquivos de Internacionalização

Os valores dos textos internacionalizáveis são armazenados em arquivos de propriedades localizados na pasta **src/resources**.

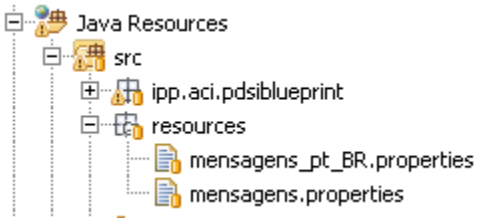


Figura 13 – Pasta src/resources

3.1.2.6 WEB-INF

Arquivos de configuração, bibliotecas e páginas web do sistema. As páginas são a interface de usuário do sistema e representam as funcionalidades disponibilizadas pelo mesmo.

3.1.2.7 Arquivos de Configuração

Os seguintes arquivos estão presentes na pasta **WEB-INF** e as respectivas configurações estão explicadas na seção Configuração, deste documento:

- contextoWeb.xml
- web.xml

Figura 14 – Arquivos de configuração da camada de apresentação

3.1.2.8 Bibliotecas

A pasta **lib** contém as bibliotecas utilizadas pelo sistema.

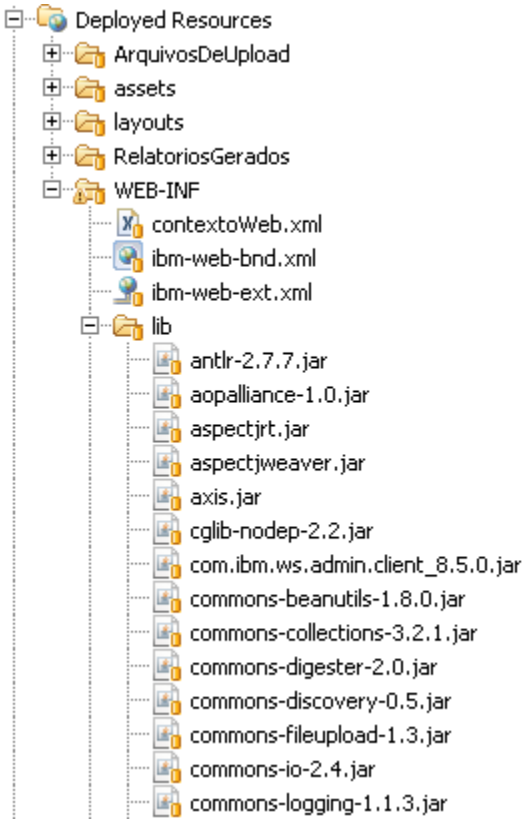


Figura 15 – Pasta lib

3.1.2.9 Páginas do Sistema

A pasta ***views*** contém as páginas do sistema.

As subpastas correspondem aos *beans* controladores no pacote java src/.../visao e cada página deve estar armazenada na subpasta correspondente ao respectivo controlador.

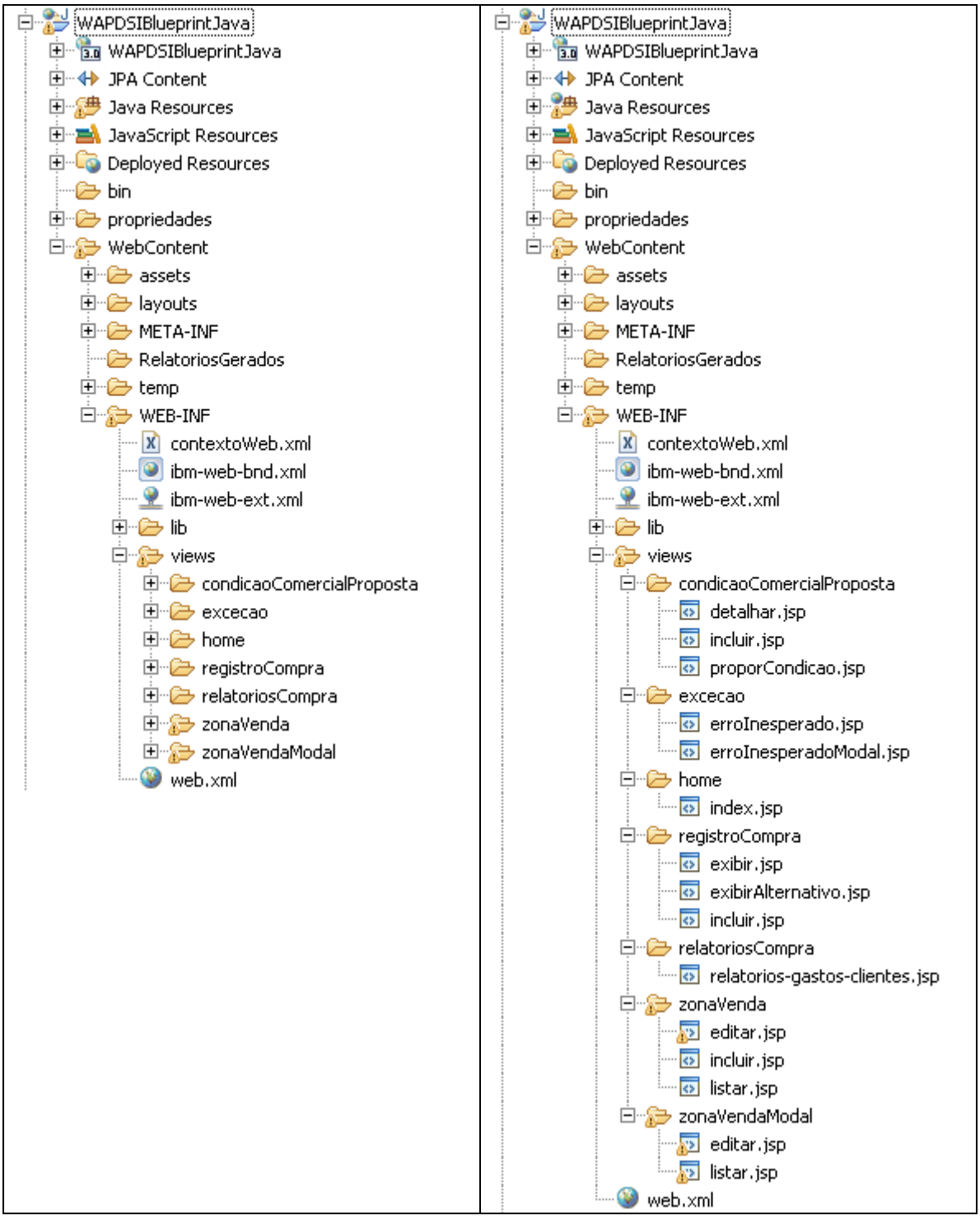


Figura 16 – Páginas do sistema

3.1.2.9.1 Tabela de Views

A tabela abaixo ilustra as estratégias de organização e nomenclatura utilizadas para as *views* do sistema:

Tabela 9 - Listagem de páginas do sistema

Controlador	Subpasta da View	Nome da View
CondicaoEspecialPropostaController	/views/condicaoComercialProposta/	detalhar.jsp
	/views/condicaoComercialProposta/	incluir.jsp
	/views/condicaoComercialProposta/	proporCondicao.jsp
RegistroCompraController	/views/registroCompra/	exibir.jsp
	/views/registroCompra/	incluir.jsp
	/views/relatoriosCompra/	relatoriosGastosClientes.jsp
ZonaVendaController	/views/zonaVenda/	editar.jsp
	/views/zonaVenda/	incluir.jsp
	/views/zonaVenda/	listar.jsp
HomeController	/views/home/	index.jsp

Esta tabela ilustra como as páginas estão organizadas por subpastas dentro da pasta de *views*. Estas subpastas estão relacionadas ao nome do controlador com a primeira letra minúscula e sem o sufixo *Controller*.

3.2 Infraestrutura do projeto

Esta seção descreve a infraestrutura padrão dos projetos, apresentando detalhes como organização de pacotes, arquivos de configuração e padrões.

3.2.1 Organização em pacotes

As classes devem ser organizadas em pacotes de acordo com a estrutura apresentada nesta seção. A devida organização de pacotes facilita a manutenção do código indicando o contexto e a finalidade de cada artefato que compõe o sistema.

Primeiramente, todo o sistema deve seguir o padrão base de estruturação de pacotes. Neste padrão, devem ser definidos o nome da empresa, seguido da sigla que representa a área responsável pela construção do sistema e por fim o nome do sistema.

```
01 ipp.<sigla da divisão responsável>.<nome do sistema>
Ex.:
02 ipp.aci.pdsiblueprint
```

Código 85 - Nomenclatura de pacotes base

Após o caminho base dos pacotes do sistema, deve ser definida a divisão lógica do sistema representada pelas camadas descritas no item 2 do documento. As camadas possíveis são Visão (visao), Serviço (servico), Domínio (dominio), Dados (dados).

```
01 <caminho base>.<camada do sistema>
Ex.:
02 ipp.aci.pdsiblueprint.visao
03 ipp.aci.pdsiblueprint.servico
04 ipp.aci.pdsiblueprint.dominio
05 ipp.aci.pdsiblueprint.dados
```

Código 86 - Nomenclatura de pacotes camadas

Após a visão lógica do sistema, subpacotes podem ser criados de acordo com a necessidade do sistema. Os pacotes devem ser utilizados para agrupar elementos que estão logicamente ou funcionalmente relacionados. Além dessas definições, alguns pacotes devem ser criados segundo as regras descritas abaixo.

As implementações das classes para acesso à persistência de dados devem estar localizadas em pacotes dentro da camada de dados. Esses pacotes devem ser nomeados com o nome da tecnologia ou ferramenta que implementa o acesso aos dados:

```
01 <caminho base>.dados.<nome implementação>
Ex:
02 ipp.aci.pdsiblueprint.dados.oracle
03 ipp.aci.pdsiblueprint.dados.servicos
```

Código 87 - Nomenclatura de pacotes implementação

Classes responsáveis por tratamento de dados em formato de web services, devem ser mapeadas em um pacote para serviços, informando também um nome para o serviço:

```
01 <caminho base>.dados.servicos.<identificador do servico>
Ex.:
02 ipp.aci.pdsiblueprint.dados.servicos.saa
```

Código 88 - Nomenclatura de pacotes serviços

As fábricas responsáveis por criar entidades complexas do domínio devem estar localizadas no pacote:

```
01 <caminho base>.dominio.fabrica
Ex.:
02 ipp.aci.pdsiblueprint.dominio.fabrica
```

Código 89 - Nomenclatura de pacotes fábricas

Os serviços de domínio devem estar localizados no pacote:

```
01 <caminho base>.dominio.servico
Ex.:
02 ipp.aci.pdsiblueprint.dominio.servico
```

Código 90 - Nomenclatura pacotes serviços de domínio

Os *value objects* (VO) devem estar localizados no pacote:

```
01 <caminho base>.dominio.vo
Ex.:
02 ipp.aci.pdsiblueprint.dominio.vo
```

Código 91 - Nomenclatura de pacotes VO

As classes utilitárias são classes que fornecem funcionalidades para toda a aplicação. Essas classes devem estar localizadas em um pacote comum chamado útil no mesmo nível das camadas, como exemplificado abaixo:

```
01 <caminho base>.util
Ex.:
02 ipp.aci.pdsiblueprint.util
```

Código 92 - Nomenclatura de pacotes util

As exceções de um sistema devem estar localizadas em um pacote comum chamado *excecao* contido no pacote *util*, para ser compartilhado por toda a aplicação, como exemplificado abaixo:

```
01 <caminho base>.util.excecao
Ex.:
02 ipp.aci.pdsiblueprint.util.excecao
```

Código 93 - Nomenclatura de pacotes exceção

A estruturação completa está exemplificada abaixo:

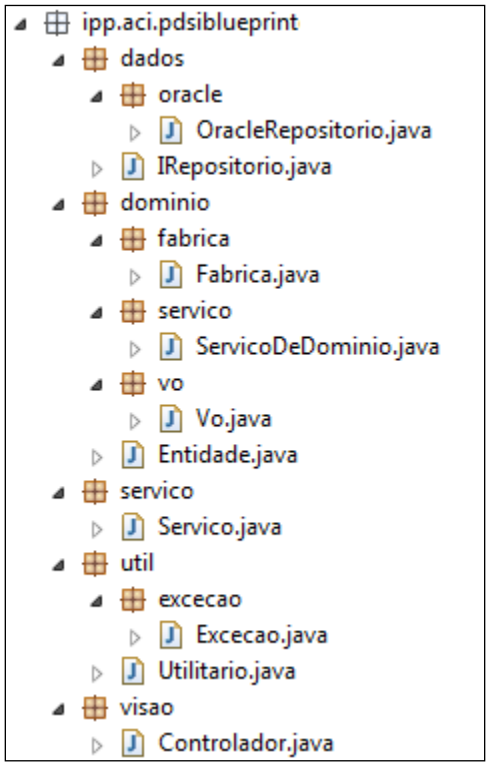


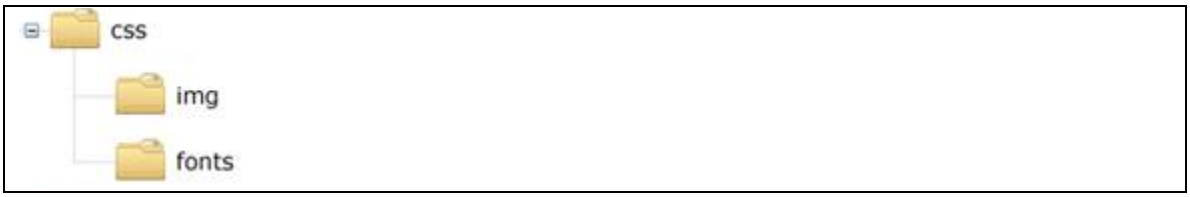
Figura 17 - Organização dos Artefatos do Sistema

3.2.2 Organização de Diretórios

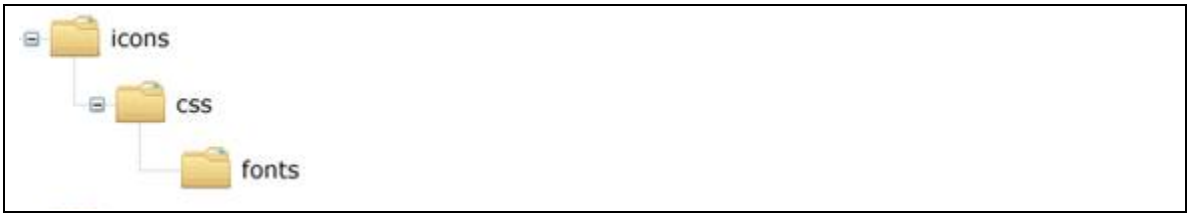
Além das classes, também é necessária uma organização para os demais artefatos necessários ao sistema, com o mesmo intuito de facilitar entendimento e manutenção da aplicação.

Artefatos como imagens, estilos, ícones e códigos javascript devem estar em uma pasta chamada *assets*, que significa recursos.

Os estilos (css) devem estar contidos nos recursos e estarão divididos entre imagens e fontes:

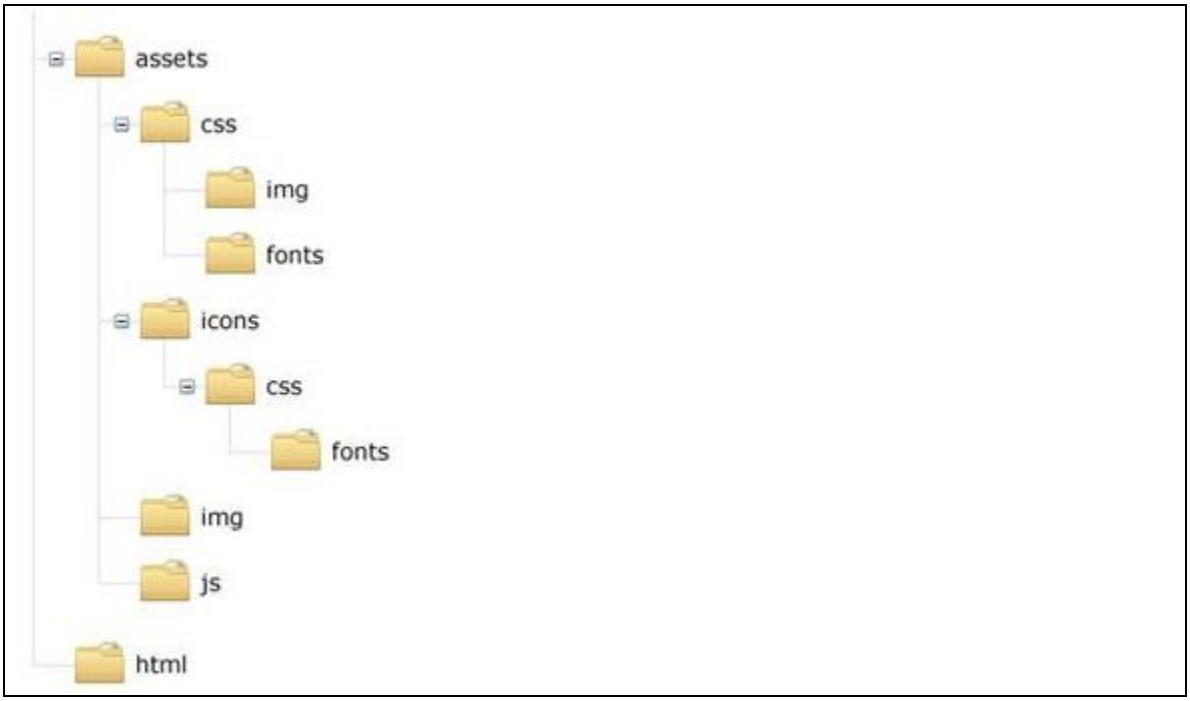


Os ícones (icons) devem estar contidos nos recursos e poderão conter estilos:



Além destes, deve conter uma pasta para imagens (img) e códigos javascript (js), inclusos nos recursos, e uma pasta no mesmo nível de recursos para códigos java server *pages* (jsp) ou páginas html (html).

A estruturação completa está exemplificada abaixo:



3.2.3 Arquivos de Configuração

Este tópico apresenta informações dos arquivos de configuração utilizados no projeto. Estas informações incluem um resumo sobre o objetivo do arquivo, localização dos arquivos no projeto e referências para mais detalhes na documentação.

3.2.3.1 infraestrutura.xml

Este arquivo é responsável pelas configurações de infraestrutura do Spring. Neste arquivo estão configurações de estratégias de implementação formas de criação de objetos e injeção de dependência de classes de infraestrutura da aplicação.

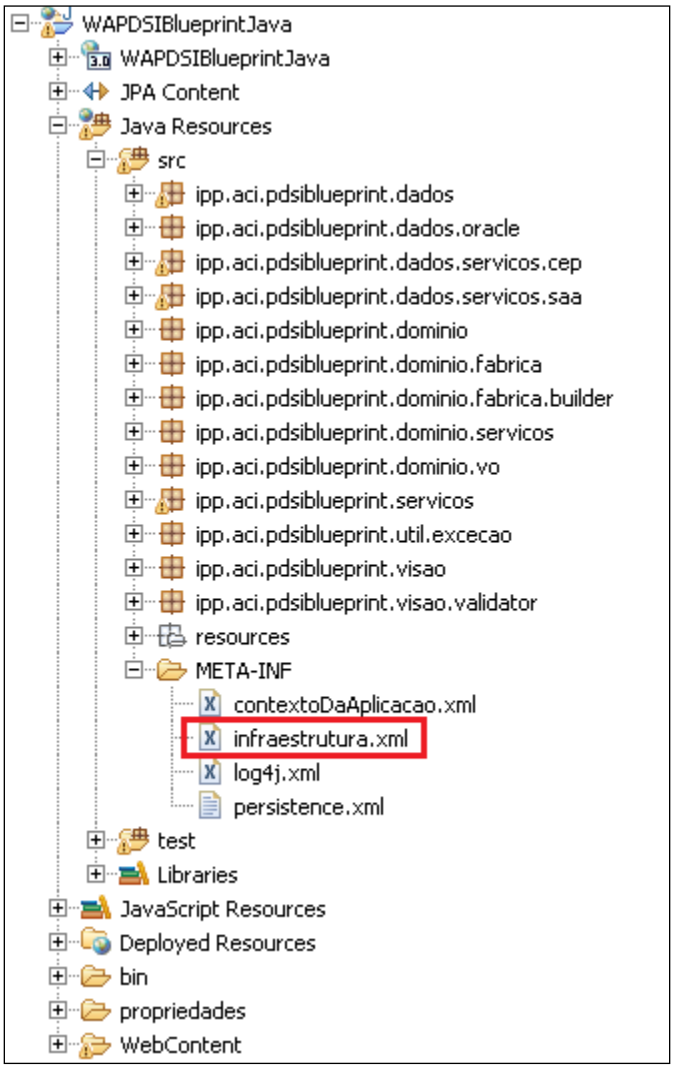


Figura 18 - Localização do Arquivo infraestrutura.xml

3.2.3.2 persistence.xml

Esse arquivo contém informações para a criação da unidade de persistência (*Persistence Unit*). A unidade de persistência contém uma série de informações que são utilizadas para configurar o gerenciador de entidades (*Entity Manager*) e o acesso à fonte de dados. Além disso, podem ser feitas configurações de frameworks relacionados ao acesso às fontes de dados como o Hibernate e o EhCache.

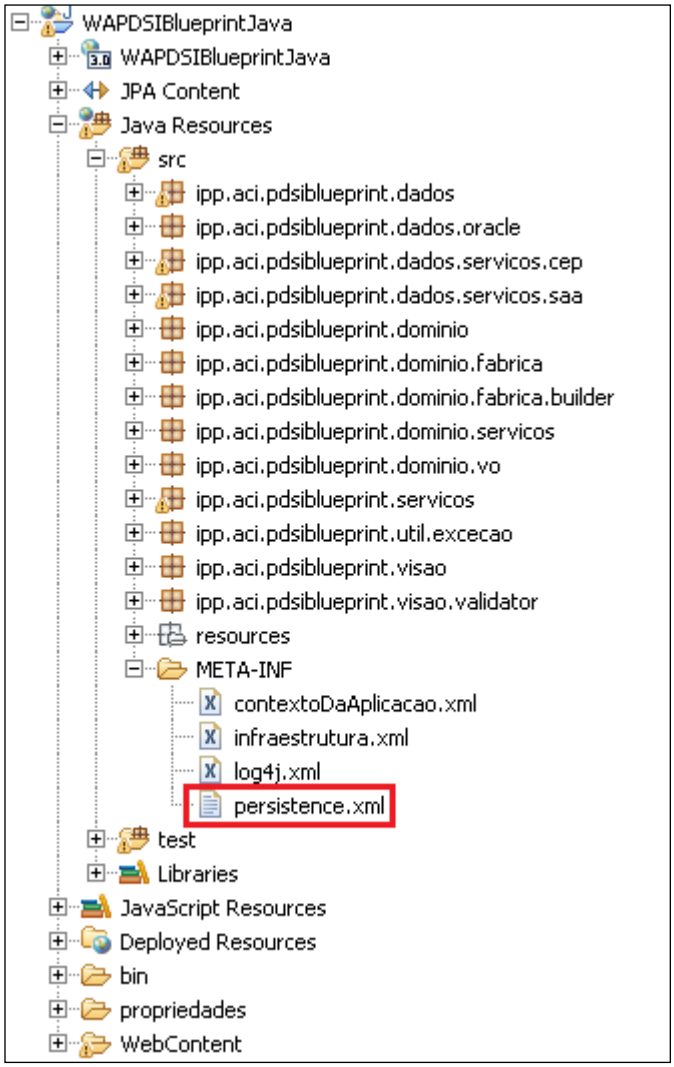


Figura 19 - Localização do Arquivo persistence.xml

3.2.3.3 web.xml

Arquivo de configuração serve como um descritor de implantação do sistema web. Este arquivo é obrigatório e possui configurações e mapeamento base do servlet do Spring MVC.

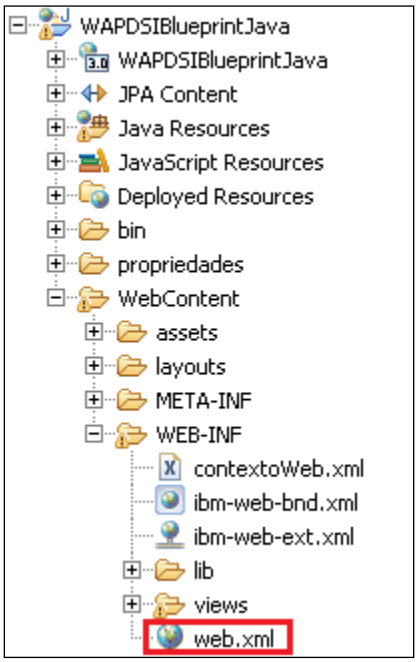


Figura 20 - Localização do Arquivo web.xml

3.2.3.4 applicationContext.xml

Este arquivo é utilizado pelo Spring para identificar e criar os *beans* da aplicação.

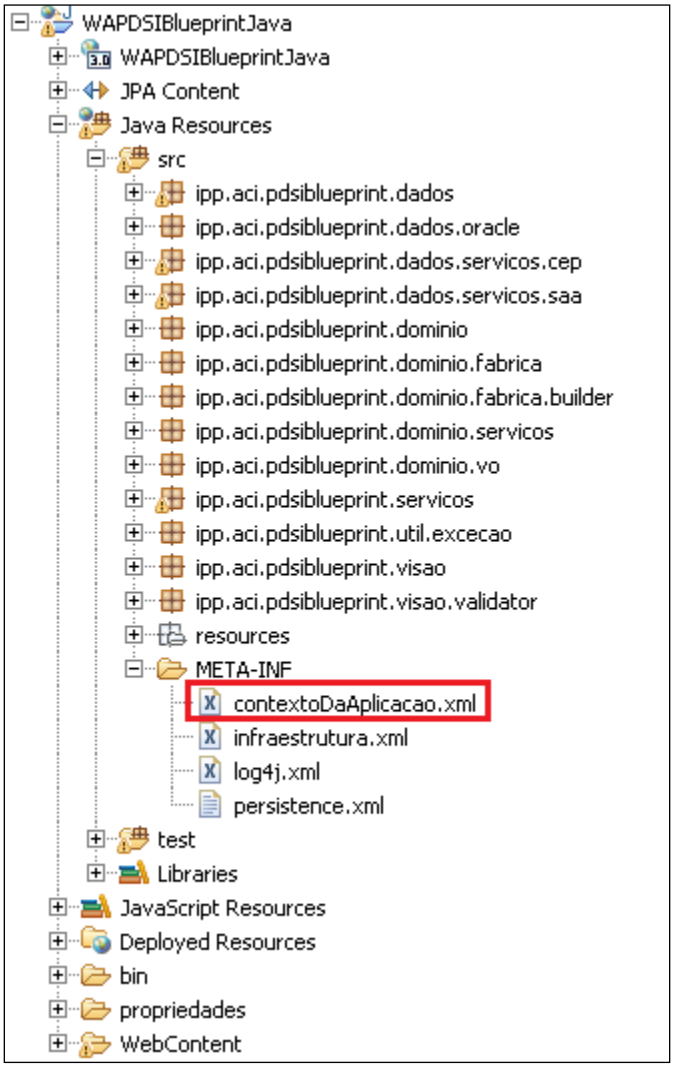


Figura 21 - Localização do Arquivo applicationContext.xml

3.2.3.5 log4j.xml

O Log4j fornece a infraestrutura para o log, como gravação de arquivo, formatação do texto do log, nível de depuração, entre outros. O arquivo log4j.xml concentra as configurações do Log4j.

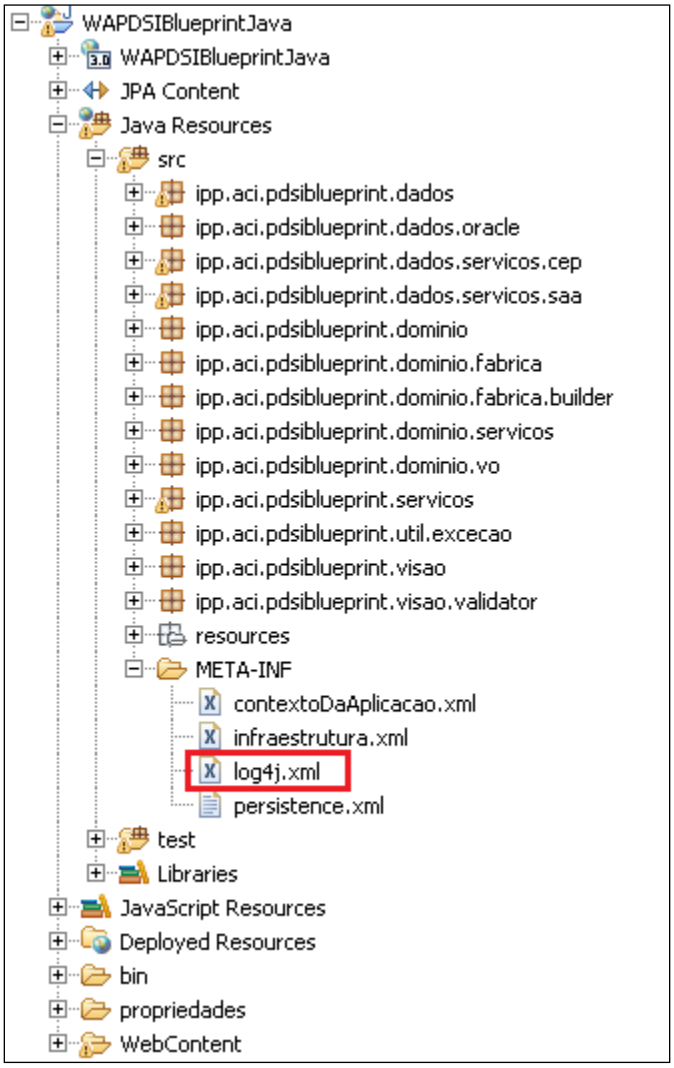


Figura 22 - Localização do Arquivo log4j.xml

3.2.3.6 contextoWeb.xml

Como dito anteriormente, este arquivo é utilizado para definir as características da aplicação web através do Spring framework. Essas características estão relacionadas à camada de visão do sistema. Dentre as configurações contidas no arquivo é possível citar a parte de internacionalização do sistema, o mapeamento entre as requisições do usuário e as classes responsáveis pelo tratamento, configurações de upload de arquivo, entre outros.

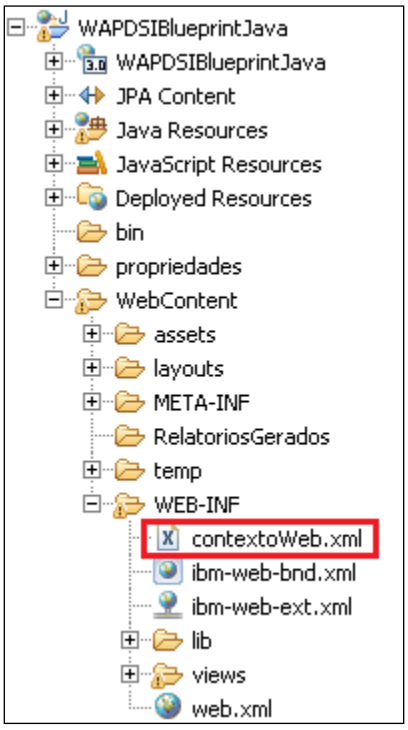


Figura 23- Localização do arquivo contextoWeb.xml

3.3 Bibliotecas Utilizadas

Esta seção apresenta uma descrição das bibliotecas utilizadas no projeto. Esta seção é importante para entender o objetivo de cada biblioteca e o porquê da utilização da mesma. Algumas bibliotecas somente são utilizadas em ambientes específicos como as bibliotecas para testes. Deste modo, ao migrar o projeto de ambiente é vital a verificação das bibliotecas que deverão acompanhar o projeto.

Abaixo é apresentada uma tabela com todas as bibliotecas utilizadas:

Tabela 10 - Listagem de bibliotecas utilizadas no projeto

Nome da Biblioteca
antlr-2.7.7.jar
aopalliance-1.0.jar
aspectjrt.jar
aspectjweaver.jar
axis.jar
cglib-nodep-2.2.jar
com.ibm.ws.admin.client_8.5.0.jar
commons-beanutils-1.8.0.jar
commons-collections-3.2.1.jar
commons-digester-2.0.jar
commons-discovery-0.5.jar
commons-fileupload-1.3.jar
commons-io-2.4.jar
commons-logging-1.1.3.jar
dom4j-1.6.1.jar
ehcache-2.7.5.jar
ehcache-core-2.6.5.jar
hibernate-commons-annotations-4.0.2.Final.jar
hibernate-core-4.2.4.Final.jar
hibernate-ehcache-4.0.0.CR6.jar
hibernate-entitymanager-4.2.4.Final.jar
hibernate-jpa-2.0-api-1.0.1.Final.jar
hibernate-validator-4.1.0.Final.jar
hibernate-validator-annotation-processor-4.1.0.Final.jar
hsqldb-1.8.0.7.jar
javassist-3.15.0-GA.jar
javax.persistence.jar
jboss-logging-3.1.0.GA.jar
jboss-transaction-api_1.1_spec-1.0.1.Final.jar
json-simple-1.1.1.jar
log4j-1.2.17.jar
pdsi-framework_1.0.1
poi-3.9-20121203.jar

poi-ooxml-3.9.jar
poi-ooxml-schemas-3.9-20121203.jar
poi-scratchpad-3.9-20121203.jar
slf4j-api-1.6.4.jar
slf4j-log4j12-1.7.5.jar
spring-aop-3.2.5.RELEASE.jar
spring-beans-3.2.5.RELEASE.jar
spring-context-3.2.5.RELEASE.jar
spring-context-support-3.2.5.RELEASE.jar
spring-core-3.2.5.RELEASE.jar
spring-expression-3.2.5.RELEASE.jar
spring-jdbc-3.2.5.RELEASE.jar
spring-orm-3.2.5.RELEASE.jar
spring-test-3.2.5.RELEASE.jar
spring-tx-3.2.5.RELEASE.jar
spring-web-3.2.5.RELEASE.jar
spring-webmvc-3.2.5.RELEASE.jar
standard-1.1.2.jar
stax-api-1.0.1.jar
tiles-api-3.0.3.jar
tiles-autotag-core-runtime-1.1.0.jar
tiles-core-3.0.3.jar
tiles-extras-3.0.3.jar
tiles-jsp-3.0.3.jar
tiles-request-api-1.0.3.jar
tiles-request-jsp-1.0.3.jar
tiles-request-servlet-1.0.3.jar
tiles-servlet-3.0.3.jar
tiles-template-3.0.3.jar
validation-api-1.0.0.GA.jar
xml-apis-1.0.b2.jar
xmlbeans-2.3.0.jar

3.3.1 PDSI Framework

A biblioteca PDSI Framework fornece os recursos necessários para a construção das aplicações, nela estão contidos os elementos descritos nesse documento, como por exemplo as classes OracleRepositorioGenerico, ControladorGenerico, LogInterceptor e Utilitários de Auxilio do desenvolvimento de sistema da Ipiranga.

- `pdsi-framework_1.0.1.jar`

3.3.2 Hibernate

O Hibernate fornece diversos recursos para o acesso a fontes de dados. Para utilização do Hibernate, é necessário adicionar ao projeto uma lista de bibliotecas obrigatórias. Essa lista conta com as bibliotecas principais e uma série de dependências. Segue abaixo a lista de bibliotecas obrigatórias:

- `antlr-2.7.7.jar`
- `dom4j-1.6.1.jar`
- `hibernate-commons-annotations-4.0.2.Final.jar`
- `hibernate-core-4.2.4.Final.jar`
- `hibernate-jpa-2.0-api-1.0.1.Final.jar`
- `javassist-3.15.0-GA.jar`
- `jboss-logging-3.1.0.GA.jar`
- `jboss-transaction-api_1.1_spec-1.0.1.Final.jar`

Opcionalmente, o Hibernate oferece funcionalidades que não são obrigatórias e que podem ser adicionadas no projeto. Para o frameworks estamos utilizando as seguintes:

- `O hibernete utiliza uma série de bibliotecas que fornecem funionalidades para a utilização de cache na aplicação`
  - `hibernate-ehcache-4.0.0.CR6.jar`
  - `ehcache-core-2.6.5.jar`
  - `slf4j-api-1.6.4.jar`
  - `ehcache-2.7.5.jar`
  - `slf4j-simple-1.6.4.jar`
- `hibernate-entitymanager-4.2.4.Final.jar – Biblioteca que fornece a opção de utilização do gerenciador de entidades (Entity Manger)`
- `hibernate-validator-4.1.0.Final.jar`
- `hibernate-validator-annotation-processor-4.1.0.Final.jar`

3.3.3 Spring

O Spring é um framework orientado a componentes que oferece diversas funcionalidades para as aplicações. Esses componentes estão subdivididos em diversas bibliotecas. Para a utilização de determinadas funcionalidade do Spring basta adicionar a biblioteca relacionada à funcionalidade. Abaixo segue a lista de bibliotecas do Spring e suas dependências.

- `spring-aop-3.1.1.RELEASE.jar`
- `spring-asm-3.1.1.RELEASE.jar`
- `spring-beans-3.1.1.RELEASE.jar`
- `spring-context-3.1.1.RELEASE.jar`
- `spring-context-support-3.1.1.RELEASE.jar`
- `spring-core-3.1.1.RELEASE.jar`
- `spring-expression-3.1.1.RELEASE.jar`
- `spring-jdbc-3.1.1.RELEASE.jar`
- `spring-orm-3.1.1.RELEASE.jar`
- `spring-test-3.1.1.RELEASE.jar`
- `spring-tx-3.1.1.RELEASE.jar`
- `spring-web-3.1.1.RELEASE.jar`
- `spring-webmvc-3.1.1.RELEASE.jar`

3.3.4 JUnit

Esta biblioteca oferece funcionalidades para a execução de testes unitários na aplicação. Ao contrário das outras bibliotecas, o JUnit foi adicionado no *claspath* da aplicação através do Rational Application Developer.

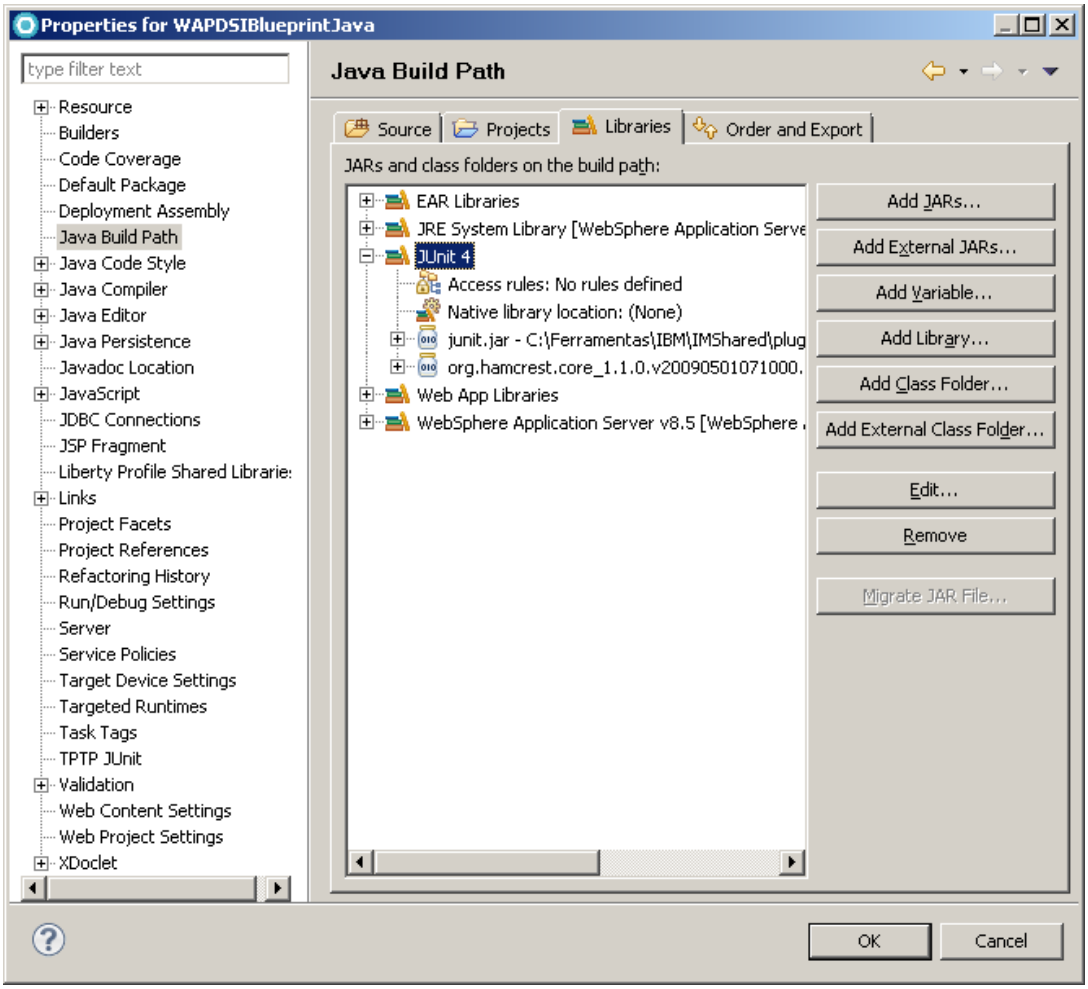


Figura 24 - Biblioteca do JUnit

ATENÇÃO, esta biblioteca não pode ser copiada para ambientes de produção.

3.3.5 HSQLDB

Essa biblioteca oferece funcionalidades para a utilização de um banco de dados em memória. Esta biblioteca é muito útil para a execução de testes, pois a execução em memória é mais rápida do que a criação de conexões para bancos de dados. No contexto deste framework somente será utilizada para a execução de testes. **ATENÇÃO, esta biblioteca não pode ser copiada para ambientes de produção**.

- `hsqldb-1.8.0.7.jar`

3.3.6 Tiles

O Tiles é um framework que permite a autoria de fragmentos de páginas que são montados em tempo de execução, reduzindo a redundância de código de interface, possibilitando o reuso, padronizando as páginas e simplificando a manutenção e evolução da camada de apresentação.

Para utilização do Tiles, é necessário adicionar ao projeto as seguintes bibliotecas do framework:

- `tiles-api-3.0.3.jar`
- `tiles-autotag-core-runtime-1.1.0.jar`

- `tiles-core-3.0.3.jar`
- `tiles-extras-3.0.3.jar`
- `tiles-jsp-3.0.3.jar`
- `tiles-request-api-1.0.3.jar`
- `tiles-request-jsp-1.0.3.jar`
- `tiles-request-servlet-1.0.3.jar`
- `tiles-servlet-3.0.3.jar`
- `tiles-template-3.0.3.jar`
  Também é necessário adicionar as seguintes bibliotecas ao classpath do projeto, para o correto funcionamento do Tiles:
- `commons-beanutils-1.8.0.jar`
- `commons-digester-2.0.jar`
- `commons-logging-1.1.3.jar`

3.4 Testes

A aplicação foi desenvolvida com um pacote contendo as classes de testes cujo objetivo era a execução de testes unitários. Essa medida tem o objetivo de garantir a qualidade dos artefatos gerados. As classes de testes foram desenvolvidas utilizando o banco em memória HSQL e o Framework JUnit.

Foram desenvolvidos testes unitários para duas camadas do sistema. As camadas testadas foram a Camada de Dados e a Camada de Serviços. Os testes estão utilizando uma configuração diferenciada da aplicação, que está localizada no pacote *resources*. Os arquivos de testes estão anotados contendo o caminho para os arquivos de configurações adequados.

Para a execução dos testes foi criado um script de inserção localizado dentro do pacote *resources* com o nome *insert-data.sql*. Para que o script seja rodado no banco em memoria é necessário criar um objeto do tipo *EmbeddedDatabase* que é criado aplicando o script de inserção.

```
01 @RunWith(SpringJUnit4ClassRunner.class)
02 @ContextConfiguration(locations = {
03         "classpath:/META-INF/applicationContext.xml",
04         "classpath:/resources/infraestrutura.xml" })
05 public class TestaServicoDeRegistroCompra {
06     ...
07     @Autowired
08     ServicosDeRegistroCompra servico;
09
10     EmbeddedDatabase db;
11
12     @Test
13     @Transactional
14     @Rollback(false)
15     public void popularBaseDeDados() throws SQLException {
16
17         EmbeddedDatabaseBuilder builder = new
EmbeddedDatabaseBuilder();
18         db =
builder.setType(EmbeddedDatabaseType.HSQL).setName("ipirangaDB")
19                 .addScript("classpath:/resources/inserts-
data.sql").build();
20     }
21
22     @Test
23     public void
quandoObtemRegistrosCompraAssociadosAUnidadeOperacionalComSucesso() {
24         Collection<RegistroCompra> registrosAssociados = servico
25
.obterRegistrosCompraAssociadosAUnidadeOperacional("UNIOP01");
26         Iterator<RegistroCompra> iterator =
registrosAssociados.iterator();
27         assertEquals(Long.valueOf(123456),
iterator.next().getNumeroRegCpra());
28     }
29
30     ...
31 }
```

Código 94 - Exemplo de classe de teste

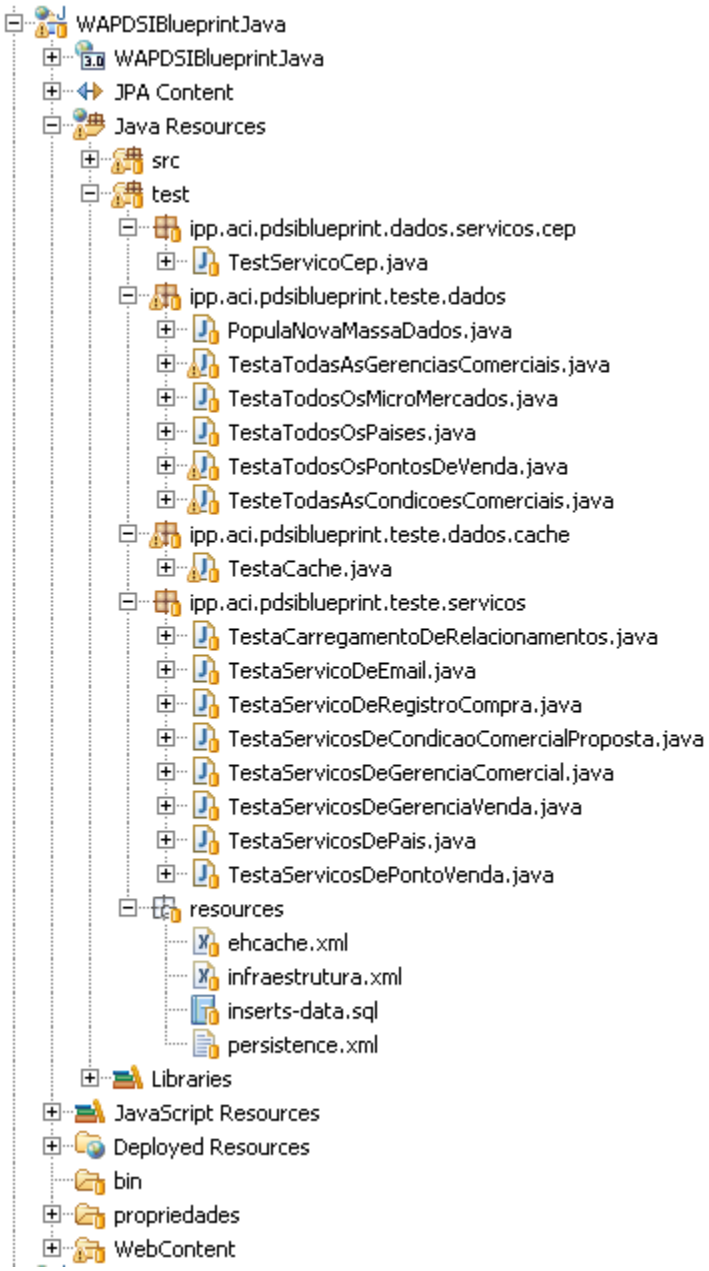
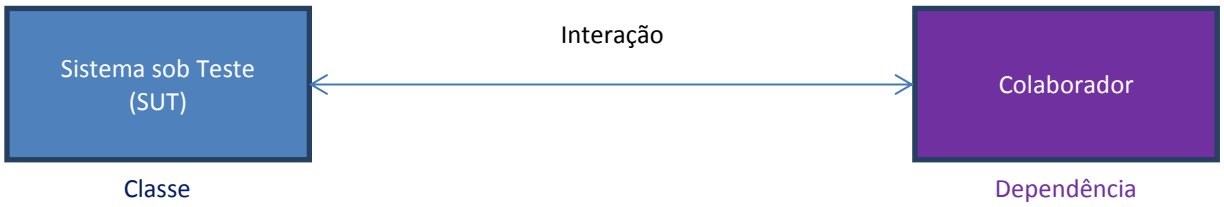


Figura 25 - Pacote de testes do sistema

3.5 Testes com Mockito

Um desafio enfrentado durante o desenvolvimento dos testes unitários é que eventualmente a classe testada pode depender de outros componentes. A configuração desses componentes pode se tornar uma tarefa tão complexa que em alguns casos inviabilize o teste.



Uma dependência é um **Colaborador**, é alguém que o **Sistema sob Teste (SUT)** necessita usar para algum propósito.

Figura 26 – Classe Testada e uma Dependência

Em vez de configurar as dependências, é possível usar componentes que emulem (tenha estrutura e comportamento semelhante) os componentes reais, mas, que na verdade, são versões simplificadas que reduzem a complexidade e facilitam o teste.

Meszaros (2007) usa o termo *Test Double* como o termo genérico (meta) para componentes que emulam componentes reais para fins de teste. Além disso, definiu alguns tipos de *Test Double*:

- Objetos ***Dummy*** são criados, mas nunca utilizados. Normalmente eles são usados apenas para preencher listas de parâmetros.
- Objetos ***Fake*** possuem implementações, mas geralmente são pouco adequados para serem usados em ambientes de produção (um banco de dados em memória é um bom exemplo).
- ***Stubs*** fornecem respostas pré-definidas para as chamadas feitas durante o teste, geralmente não respondem chamadas para as quais não foram programadas.
- ***Spies*** são *stubs* que também registram algumas informações baseadas na forma como foram chamadas. Um exemplo disto pode ser um serviço de e-mail que registra quantas mensagens foram enviadas.
- ***Mocks*** são objetos pré-programados com expectativas (expectations) que definem como queremos que o *mock* se comporte durante as chamadas que ele espera receber.

3.5.1 Objetos Mock

Mocks descrevem casos especiais de objetos que emulam objetos reais para teste. Eles são criados para testar o comportamento de algum outro objeto. *Mocks* têm a mesma interface que os objetos reais que emulam, permitindo que um objeto cliente interaja com ele como se estivesse usando um objeto real.

Em testes unitários, os *mocks* podem emular o comportamento de objetos complexos e reais e, portanto, são úteis quando um objeto real é impraticável ou impossível de incorporar em um teste unitário. Se um objeto tiver uma das seguintes características, pode ser útil usar um *mock* no seu lugar (Wikipedians, 2017):

- O objeto fornece resultados não deterministas (por exemplo, o tempo atual ou a temperatura atual);
- O objeto tem estados que são difíceis de criar ou reproduzir (por exemplo, um erro de rede);
- O objeto é lento (por exemplo, um banco de dados completo, que teria que ser inicializado antes do teste);
- O objeto ainda não existe ou pode mudar o comportamento;
- O objeto inclui informações e métodos exclusivamente para fins de teste (e não para sua tarefa real).

3.5.2 Mock Frameworks

É possível criar *mocks* manualmente (por meio do código) ou usar um *mock framework* para emular classes.

O uso de um *mock framework* simplifica significativamente o desenvolvimento de testes para classes com dependências externas. Quando um *mock framework* é usado em testes, é possível:

- Simular as dependências externas e inseri-las como *Mocks* na classe em teste;
- Executar a classe em teste;
- Verificar se o código foi executado corretamente.

Entre os frameworks que podemos citar temos: *jMock*, *EasyMock*, *JMockit* e *Mockito*.

O *Mockito* é um framework que facilita a criação de *Mocks* e pode ser usado junto com os testes unitários. *Mockito* (versão atual: 2) tem uma comunidade muito numerosa no *StackOverflow*; encontra - se entre os top 10 frameworks java e; é considerado pela comunidade de *Extreme Programming* como o futuro das abordagens *Test Driven Development – TDD*, *Behavior Driven Developmnet – BDD* e *Mocking* em *Java*.

Para seguir o padrão da Ipiranga de mock, é necessário declarar uma dependência na biblioteca "mockito-core" no *Maven*, como pode ser visto no exemplo abaixo:

```
01        <dependency>
02             <groupId>org.mockito</groupId>
03             <artifactId>mockito-core</artifactId>
04             <version>2.10.0</version>
05        </dependency>
```

Código 95 – Dependência “mockito-core”

Mockito usa:

- ***byte-buddy***: para criar uma subclasse da classe dada.
- ***Objenesis***: uma vez que a classe é gerada e carregada, sua instância é criada usando *Objenesis*.

3.5.3 Criando Mocks

O Mockito fornece vários métodos para criar *mocks*:

- Usando o método estático *Mockito.mock()*

```
01    import static org.mockito.Mockito.*;
02    ...
03    TerrenoVo terreno = mock(TerrenoVo.class);
04    ...
```

Código 96 – Criando objetos mock com “mock()”

- Usando a anotação *@Mock*

```
01    import static org.mockito.Mockito.*;
02    ...
03    @Mock
04    private TerrenoVo terreno;
05    ...
```

Código 97 – Criando objetos mock com "@Mock"

Quando é usada a anotação *@Mock*, deve-se ativar a criação de objetos anotados. O *@RunWith (MockitoJUnitRunner.class)* permite isso. Esta configuração invoca o método estático *MockitoAnnotations.initMocks(this)* para criar os campos anotados

```
01    @RunWith(MockitoJUnitRunner.class)
04    public class TerrenoSdTest {
03     @Mock
04     private TerrenoVo terreno;
05     ...
06    }
```

Código 98 – Ativando a criação de objetos mock com "@Mock"

Mockito pode emular interfaces, classes abstratas e classes concretas não finais. *Mockito* não pode emular classes finais e métodos finais ou estáticos.

3.5.4 *Criando Spies*

A notação *@Spy* ou o método *spy()* criam um *Spy* que pode ser usado para espionar um *objeto real*. Com um *Spy* (Mocking parcial), métodos reais são invocados, mas ainda podem ser verificados e emulados (*stubbed*). As chamadas a estes métodos são gravadas e os fatos dessas chamadas podem ser verificados (*verifiy*()). A seguir um exemplo de *Spy*:

```
01    @RunWith(MockitoJUnitRunner.class)
04    public class FinanciamentoSdTest{
03     @Spy
04     private FinanciamentoSd servico;
05     ...
06    }
```

Código 99 – Criando Spy com "@Spy"

Todos os métodos *Mockito* usados para configurar um *Mock* também são aplicáveis à configuração de um *Spy*.

Spies devem ser raramente e cuidadosamente usados em comparação com *mocks*, mas você pode achá-los úteis para testar código legado que não pode ser refatorado, onde o teste requer *mock parcial*. Nesses casos, você pode simplesmente criar um *Spy* e emular (*stub*) alguns dos seus métodos para obter o comportamento desejado. Antes da versão 1.8, *Spies* eram considerados *smell codes*, pois eles não implementavam corretamente o conceito de "partial mocking".

3.5.5 *Configurando Mocks*

Mockito permite configurar os *valores de retorno* dos métodos especificados no *mock* através de sua API. Chamadas a métodos não especificados no *mock* retornam valores padrão "vazios":

- *null* para objetos;
- *0* para números;
- *false* para booleano;
- *empty collections* para coleções;

Também é possível especificar *valores multiplos* que serão retornados como resultados de chamadas consecutivas a um método. O último valor será usado como resultado para futuras chamadas ao método:

```
01    @RunWith(MockitoJUnitRunner.class)
02    public class NegociacaoSdCalculaMixGAPropostaTest {
03          ...
04          doReturn(5.0,10.0).
            when(servico).totalProduto(ArgumentMatchers.<ProdutoVo>a
            nyList(), ArgumentMatchers.<EnumProduto>anyList(),
            anyString());

05     ...
06    }
```

Código 100 – Configurando valores de retorno

3.5.5.1 Stubbing de Métodos

É uma boa prática definir o que fazer (Do) e quando fazer (When) quando é feita a configuração do *Mock*. Isso é chamado de *stubbing*. Mockito oferece duas formas de fazer *stubbing*:

1. *"when thenReturn";*
2. *"doReturn when".*

A sequência de métodos *when(....).thenReturn (....)* é usada para especificar um *valor de retorno* para uma chamada de *método* com *parâmetros pré-definidos*.

Também é possível usar métodos como *anyString()* ou *anyInt()* para definir o *tipo* de entrada, e que um certo valor será retornado independente do *valor* de entrada.

```
01    @RunWith(MockitoJUnitRunner.class)
02    public class FinanciamentoSdTest {
03     ...
04     public void verificarFuncao() {
05      ...
06      when(servico.obterAliquotaImpostoRenda()).thenReturn(34.0);
07      when(servico.calcularInflacao(anyInt())).
            thenReturn(0.0043923222705009035);
08      ...
09     }
10    }
```

Código 101 – Configurando "when thenReturn"

A sequência *doReturn(...).when(...).methodCall* é usada para especificar um *valor de retorno* quando o *método* deste *mock* é chamado com *parâmetros pré-definidos*.

```
01    @RunWith(MockitoJUnitRunner.class)
02    public class FinanciamentoSdTest {
03     ...
04     public void verificarFuncao() {
05      ...
06      doReturn(34.0).when(servico).obterAliquotaImpostoRenda();
07      doReturn(0.00439232222705009035).
            when(servico).calcularInflacao(anyInt());
08      ...
09     }
10    }
```

Código 102 – Configurando "doReturn when"

3.5.5.2 Lançando exceções

Os métodos *thenThrow()* e *doThrow()* definem o comportamento de um método que lançará uma exceção:

```
01    when(servico.calcularInflacao("1"))
            .thenThrow(new IllegalArgumentException());
02    //ou
03    doThrow(new IllegalArgumentException()).
            when(serviço).calcularInflacao("1");
```

Código 103 – Lançando exceções

Mockito verifica se a exceção que está sendo lançada é válida para o método emulado (*stubbed*) e reclamará se a exceção não estiver na lista de exceções verificadas do método. Considere o seguinte:

```
01    when(servico.calcularInflacao("1")).
            thenThrow(new IOException());
```

Código 104 – Verificando exceções

Mockito detectará que o método `calcularInflacao()` não pode lançar uma *IOException*.

3.5.5.3 Retornando Respostas Customizadas – Answer

É possível definir um objeto *Answer* para resultados complexos, ou seja, enquanto *thenReturn* retorna um *valor pré-definido* para todas as chamadas, com *Answers* é possível calcular a resposta (valor de retorno) baseado nos argumentos passados ao método emulado (*stubbed*).

then(), um alias para *thenAnswer()*, e *doAnswer()* tem o mesmo comportamento, como a seguir:

```
01    when(servico.calcularInflacao(anyInt()))
            .thenAnswer( invocation ->
                  invocation.getArgument(0) + 1);
02    doAnswer(invocation ->
                  invocation.getArgument(0) + 1).when(servico)
            .calcularInflacao(anyInt());
```

Código 105 – Configurando Answers

Também é possível lançar uma exceção como resultado de uma chamada a um método:

```
when(servico.calcularInflacao("1")).
      thenAnswer(invocation ->
            { throw new IllegalArgumentException(); });
```

Código 106 – Lançando exceções com Answer

3.5.6 Verificando Comportamento

Uma vez que um *Mock* ou *Spy* foi usado, é possível usar o método *verify()* para verificar se os métodos foram chamados com seus respetivos argumentos. Este tipo de teste é chamado de *behavior testing*.

Behavior testing não verifica o resultado de uma chamada a um método, no entanto, verifica se o método foi chamado, e com os parâmetros corretos.

```
@RunWith(MockitoJUnitRunner.class)
public class FinanciamentoSdTest {
 @Spy
 private FinanciamentoSd servico;
      ...
      doReturn(0.0043923222705009035)
            .when(servico).calcularInflacao(anyInt());
      ...
      verify(servico).calcularInflacao(anyInt());
      ...
}
```

Código 107 – Verificando comportamento

Por padrão, *Mockito* verifica que o método foi chamado uma vez, mas é possível verificar qualquer número de invocações. O segundo argumento de *verify()* pode receber alguns outros métodos como *times*(2), *atLeastOnce*(), *atLeast*(5), *atMost*(5), *only*(), *never*()):

```
@RunWith(MockitoJUnitRunner.class)
public class FinanciamentoSdTest {
 @Spy
 private FinanciamentoSd servico;
      ...
      doReturn(0.0043923222705009035)
            .when(servico).calcularInflacao(anyInt());
      ...
      verify(servico, atLeastOnce())
            .calcularInflacao(anyInt());
      ...
}
```

Código 108 – Verificando comportamento e o número de invocações

Mockito também pode verificar a ordem das chamadas (***InOrder***) em um grupo de *Mocks*. Esta característica não é usada com frequência, mas pode ser útil quando a ordem das invocações é importante.

3.5.7 *Injetando Mocks*

A anotação *@InjectMocks* tentará injetar *mocks* via ***constructor*** *injection*, ***setter*** *injection*, ou ***property*** *injection*; nessa ordem. Se alguma destas estratégias falharem, *Mockito* **não reportará a falha**; ou seja, a dependência deve ser fornecida através de codificação.

Por exemplo, assumindo as seguintes classes:

```
01    //Classe sob teste
02    public class NegociacaoSd {
03          protected IInflacaoDados inflacaoDados;
04          ...
05    }

06    //Classe para injetar a dependência na classe de teste
07    public class InjetaMockNegociacaoSdUtil {
08          public void criaNegociacaoSd(NegociacaoSd negociacaoSd,
                    IInflacaoDados inflacaoDados, ...) {
09                negociacaoSd.inflacaoDados = inflacaoDados;
10                ...
11          }
12    }

13    //Classe de teste
14    public class NegociacaoSdTest {
15          private NegociacaoSd servico;
16          @Mock
17          private IInflacaoDados inflacaoDados;
18          private InjetaMockNegociacaoSdUtil injetaNegociacaoSd =
                              new InjetaMockNegociacaoSdUtil();
19          //inject
20          injetaNegociacaoSd.criaNegociacaoSd(servico,
            inflacaoDados, ...);
21          //adiciona um comportamneto em inflacaoDados
22          given(inflacaoDados.obterInflacaoMedia())
                  .willReturn(5.4);
23          //testa alguma funcionalidade
24          Assert.assertEquals(servico. ...,,);
25          ...
26    }
```

Código 109 – Injetando dependências através de uma classe utilitária

No exemplo anterior as dependências da classe "***NegociacaoSd***" (por exemplo: ***inflacaoDados***) são injetadas através de uma classe utilitária "***InjetaMockNegociacaoSdUtil***".

A classe "***NegociacaoSd***" pode ser construida via *Mockito* e suas dependências podem ser preenchidas com *objetos Mock* ou *Spy* (property injection) tal como demonstrado no código a seguir:

```
1     //Testing class
2     public class NegociacaoSdTest {
3           //@InjectMocks cria e injeta mocks
4           @InjectMocks
5           private NegociacaoSd servico;
6           //mock a ser injetado
7           @Mock
8           private IInflacaoDados inflacaoDados;
9           //adiciona um comportamneto em inflacaoDados
10          given(inflacaoDados.obterInflacaoMedia())
                  .willReturn(5.4);
11          //testa alguma funcionalidade
12          Assert.assertEquals(servico. ...,,);
13          ...
14    }
```

Código 110 – Injetando dependências através de *@InjectMocks*

3.5.8 Mantendo o Código de Teste Limpo

A regra *STRICT_STUBS* ajuda a manter o código de teste limpo e verifica por erros que não foram percebidos. Nós ativamos a regra *STRICT_STUBS* do *Mockito* através das seguintes linhas de código:

```
1     //Testing class
2     public class FinanciamentoSdTest {
3           @Spy
4           private FinanciamentoSd servico;
5           @Rule
6           public MockitoRule rule =
       MockitoJUnit.rule().strictness(Strictness.STRICT_STUBS);
13          ...
14    }
```

Código 111 – Ativando a regra STRICT STUBS

Quando ativamos esta regra:

- O teste falhará no começo quando o método emulado (*stubbed*) for chamado com argumentos diferentes dos quais foi configurado (*PotentialStubbingProblem exception*).

```
1     //Método invocado com argumentos diferentes
2     doReturn(0.0043923222705009035)
            .when(servico).calcularInflacao(5);
3     servico. calcularInflacao(6);
```

Código 112 – *PotentialStubbingProblem*

- O teste falhará no começo quando *forem chamados métodos de mocks sem comportamento configurado* (PotentialStubbingProblem exception).

```
1     //Método invocado não foi configurado
2     servico. calcularInflacaoInexistente(6);
```

Código 113 – *PotentialStubbingProblem*

- O teste falhará quando o método emulado (*stubbed*) não for chamado (UnnecessaryStubbingException exception).

```
1     //verifica que todos os métodos configurados foram chamados
2     doReturn(0.0043923222705009035)
            .when(servico).calcularInflacao(5);
3     //fim da execução
```

Código 114 – UnnecessaryStubbingException

- org.mockito.Mockito.verifyNoMoreInteractions(Object) verifica se todos os métodos emulados (*stubbed*) foram chamados durante o teste.

```
1     //verifica que todos os métodos configurados foram chamados
2     verifyNoMoreInteractions(servico);
```

Código 115 – verifyNoMoreInteractions

3.5.9 Behavior Driven Development – BDD

Mockito incentiva o *behavior testing*, em lugar do *state testing*. *Behavior Driven Development* define um estilo de escrita de testes que usa o formato ***given***, ***when*** e ***then*** nos métodos de teste. *Mockito* fornece métodos especiais para este estilo de escrita. No exemplo a seguir, nós usamos o método ***given*** da classe *BDDMockito* em lugar do método ***when***:

```
01    @RunWith(MockitoJUnitRunner.class)
02    public class FinanciamentoSdCalcularValorAtualTest {
03          @Mock
04          private FinanciamentoVo encaixe;
05          @Mock
06          private FluxoFinanciamentoVo fluxoFinanciamento;
07          private FinanciamentoSd financiamentoSd;
08          ...
09          @Test
10          public void verificaCalcularValor() {
11                //given
12                given(fluxoFinanciamento.getEncaixe())
                        .willReturn(encaixe);
13                given(encaixe.getValorFinanciado())
                        .willReturn(1000.0);
14                given(encaixe.getPercLiberacao())
                        .willReturn(new Double[]{1.0});
15                //when
16                Double resultado =
                        financiamentoSd.calcularValorAtual(
                                          fluxoFinanciamento, 0);
17                Double esperado = 10.0;
18                //then
19                Assert.assertEquals(
                  "FinanciamentoSd.calcularValorAtual: Deveria ser "
                        + esperado + ".", esperado, resultado,
                        Constantes.TOLERANCIA_PADRAO);
20          }
21    }
```

Código 116 – Behavior Driven Development com Mockito

3.5.10 Limitações

Mockito possui algumas limitações, por exemplo, não é possível criar *Mocks* de métodos estáticos e métodos privados. Para isto é possível configurar *Powermock* com *Mockito*.

4 Padrões de Nomenclatura Java

Padrões de nomenclatura estabelecem um vocabulário base para a estruturação dos sistemas. Esse vocabulário facilita a comunicação entre os integrantes do projeto e contextualiza elementos do sistema. A seguir são definidas algumas regras gerais para a nomenclatura.

Ao nomear uma classe ou interface escreva em letra maiúscula a primeira letra de cada palavra que aparece no nome:

```
01 public class AtualizaDetalhesAction implements SingleAction {
02    ...
03 }
```

Código 117 - Nomenclatura de Classe

Ao nomear um método use letra minúscula para a primeira palavra e letra maiúscula apenas para a primeira letra de cada palavra subsequente que aparece no nome

```
01 public class Cliente {
02    public Cliente() {                    // método construtor
03          ...
04    }
05    public void incluir() {
06          ...
07    }
08    public Image getNome() {
09          ...
10    }
11 }
```

Código 118 - Nomenclatura de métodos

Ao nomear uma variável use letra minúscula para a primeira palavra e letra maiúscula apenas para a primeira letra de cada palavra subsequente que aparece no nome

```
01 class Usuario {
02    ...
03    private Ramal ramal;
04   private Nome nomeCompleto;
05   ...
06   public Ramal setRamal(Ramal ramal) {
07      Ramal ramalAntigo = this. ramal;
08      this.ramal = ramal;
09      return ramalAntigo;
10    }
11    ...
12    public void setNomeCompleto (Nome nomeCompleto) {
13          ...
14    }
15    ...
16 }
```

Código 119 - Nomenclatura de variáveis

Ao nomear constantes use letras maiúsculas para cada palavra e separe cada par de palavras com um *underscore.*

```
01 class Byte {
02    public static final Byte VALOR_MAX = 255;
03    public static final Byte VALOR_MIN = 0;
04    public static final Class TIPO = Byte.class;
05 }
```

Código 120 - Nomenclatura de constantes

Quando um nome qualquer contiver um acrônimo escreva apenas a primeira letra do acrônimo em maiúsculo

```
01 setDSTOffset() > setDstOffset()
02 loadXMLDocument() >  loadXmlDocument()
```

Código 121 - Uso de acrônimos

5 Framework Single Page Application (SPA)

SPA – Single Page Application é um modelo de desenvolvimento de aplicações Web e mobile que vem ganhando destaque em grandes empresas como Google, Twitter, Facebook, Microsoft, etc.

SPA basicamente significa codificar menos no server-side e mais no client-side (ou seja no navegador). É praticamente uma aplicação Desktop rodando sob o navegador. A comunicação do front-end com o back-end se dá através de chamadas AJAX para end-points de uma API.

No caso da Ipiranga, o back-end será uma aplicação Java desenvolvida na arquitetura da Ipiranga, descrita neste documento.

Uma aplicação SPA é uma aplicação web ou é um website que cabe em uma única página com o propósito de fornecer ao usuário uma experiência similar a uma aplicação desktop.

Na prática, a página da aplicação é carregada apenas uma vez, e mesmo que o usuário recarregue a aplicação, ela possui a habilidade de se manter na última tarefa feita pelo usuário, em muitos momentos com a tela exatamente como o usuário a deixou. A mais conhecida dessas aplicações é o Gmail.

5.1 AngularJS

Quanto mais interatividade acontece no lado do cliente, mais código de JavaScript é necessário para que esses componentes interativos funcionem bem. E quanto mais código for escrito, mais importante é ter uma base de código limpa e bem organizada. E este é exatamente o problema que os frameworks de JavaScript ajudam a resolver, cada um com sua própria abordagem.

Para o desenvolvimento dos sistemas da Ipiranga uma tecnologia foi homologado, o AngularJS. O AngularJS foi definido em virtude da grande participação no mercado, além de sua estabilidade e quantidade de material disponível na internet. A utilização de outras versões do framework Angular (2 e 4) bem como outras tecnologias SPA não são o alvo deste documento. O uso de tais tecnologias somente será aceita após estabelecido o padrão formal de utilização junto a equipe de Arquitetura da Ipiranga.

Uma das principais características do AngularJS é o two-way Databinding, o qual possibilita uma ligação direta e bidirecional dos dados, permitindo sincronização automática dos modelos e das visões (models e views) como pode ser visto na Figura 27 seguir.

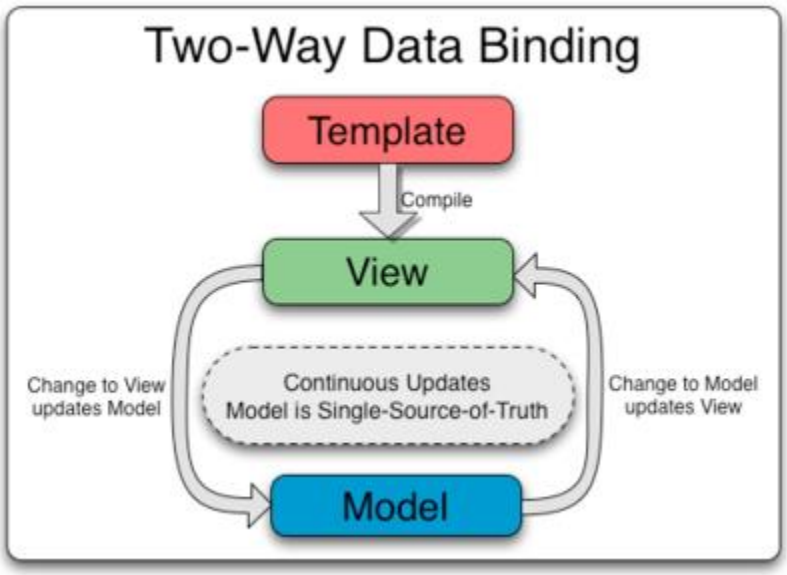


Figura 27 - Modelo de funcionamento Two-Way DataBinding

É possível fazer a separação das funcionalidades da aplicação em arquivos JavaScript, sendo necessário somente sua importação nos locais que irão utiliza-los. Esta separação permite a criação de controles, serviços, filtros, diretivas, templates e até mesmo de configurações, tornando cada componente altamente reutilizável. Abaixo é explicado cada um destes itens descritos.

Controladores (ou controllers) são responsáveis pelo controle do fluxo da informação que será apresentada na view, nele podem estar implementadas algumas regras de exibição bem como funcionalidades que serão chamadas pela view. Aplicações de grande porte devem conter diversos controladores, onde cada um apresentará o fluxo de informação de suas respectivas funcionalidades, tornando assim o código mais legível e estruturado reduzindo o número de possíveis falhas. Os controladores devem ser nomeados seguindo o padrão PascalCase, ou seja, a primeira letra de cada palavra maiúscula.

Em AngularJS, os templates podem ser definidos como views com o HTML acrescentado de elementos específicos do Angular, tais como diretivas e atributos, podendo, portanto, compreender documentos HTML inteiros, bem como apenas pequenos trechos de código HTML.

Os filtros transformam a aparência dos dados presentes em uma página web. Por exemplo, um filtro pode converter um número do escopo para uma string de tempo, ou de moeda corrente.

Serviços (ou services) são singleton, ou seja, são objetos únicos criados na inicialização da aplicação e devem ser injetados em outros componentes para execução de determinadas tarefas. A implementação de serviços facilita a reutilização de códigos, a execução de testes e a organização do projeto.

Diretivas representam novos nomes de elementos HTML ou nomes de atributos dentro de elementos HTML; diretivas adicionam e modificam o comportamento de elementos HTML para fornecimento de vários tipos de suporte a aplicações AngularJS.

5.1.1 Internacionalização

O AngularJS permite uma implementação bastante limitada de internacionalização, restringindo-se a formatação de datas, números, moedas entre outros mas não permitindo a implementação de trechos específicos de sua aplicação.

Existem diversas bibliotecas e frameworks que implementam soluções para este problema, sendo uma das mais utilizadas o Angular-translate. Mantido por seu criador, Pascal Precht, e pela comunidade do AngularJS o Angular-translate permite a implementação de internacionalização conforme descrita nos próximos tópicos.

5.1.1.1 Mapeamento de mensagens e estruturação

O mapeamento de mensagens pelo angular-translate deve seguir o padrão apresentado abaixo:

```
1  var translationsPT = {
2    'tela.pdsi.titulo': 'PDSI',
3    'comum.adicionar' : 'Adicionar',
4    'tela.adicionado': 'O contato {{username}} foi adicionado com sucesso.'
5  };
6
7  angular.module("zonaDeVenda").config(['$translateProvider', function ($translateProvider) {
8    $translateProvider.translations('pt-br', translationsPT);
9    $translateProvider.preferredLanguage('pt-br');
10   $translateProvider.useSanitizeValueStrategy('escape');
11 }]);
```

Figura 28 - Mapeamento de mensagens angular-translate

Vale ressaltar que o angular-translate permite o mapeamento das mensagens utilizando arquivos JSON e carregando este para o framework.

5.1.1.2 Injeção e carregamento

Deve-se fazer o carregamento do framework conforme mostrado na figura abaixo:

```
<script src="assets/lib/angular-translate.min.js"></script>
```

Figura 29 - Carregamento do framework

Deve-se também incluir ‘pascalprecht.translate’ como dependência na aplicação:

```
1  angular.module("zonaDeVenda", ["ngRoute", 'pascalprecht.translate']);
```

Figura 30 - Inclusão da dependência correspondente ao angular-translate

5.1.1.3 Utilização por diretivas e filtros

Para utilização do angular-translate basta utilizar um dos casos descritos abaixo:

- Filtro – {{‘chave’ | translate }}
- Diretiva - <tag_html translate=’chave’></tag_html>
- Filtro passando parâmetros - {{ 'chave' | translate:'{ parametro: ‘valor’ }' }}

5.1.2 Mover itens através de listas de seleção

Com objetivo de carregar o arquivo JavaScript que possui as funções que dão suporte aos eventos necessários para o funcionamento do componente capaz de mover itens selecionados através de duas listas de seleção (*Select Lists*), conforme definido no padrão visual da Ipiranga, é necessário que seja incluído nas páginas desenvolvidas em AngularJS o código. O trecho de código abaixo deve ser chamado no carregamento da página podendo ser incluído em no controlador do template que esta sendo carregado.

```
01 var loadScriptIpp = function() {
02     jQuery.ajax(
03         url: 'assets/lib/app.mim.js',
04         dataType: 'script',
05     });
06 };
```

Figura 31 - Trecho para funcionamento de listas de seleção

5.2 Estrutura de Diretórios

A estruturação das pastas é de suma importância para o crescimento e manutenção dos sistemas. Por isso, neste tópico é detalhada a forma de organização que as aplicações devem aplicar.

Baseado na premissa de que na Ipiranga serão desenvolvidas aplicações comerciais de tamanho médio, **foi definido** que a **estrutura de pasta** utilizada deve ser a **Specific Style**, conforme pode ser visto na Figura 32 e posteriormente explicado. Além disso, como um projeto grande sempre pode ser convertido em projetos menores, essa forma de estrutura de pastas pode ser utilizada sempre.

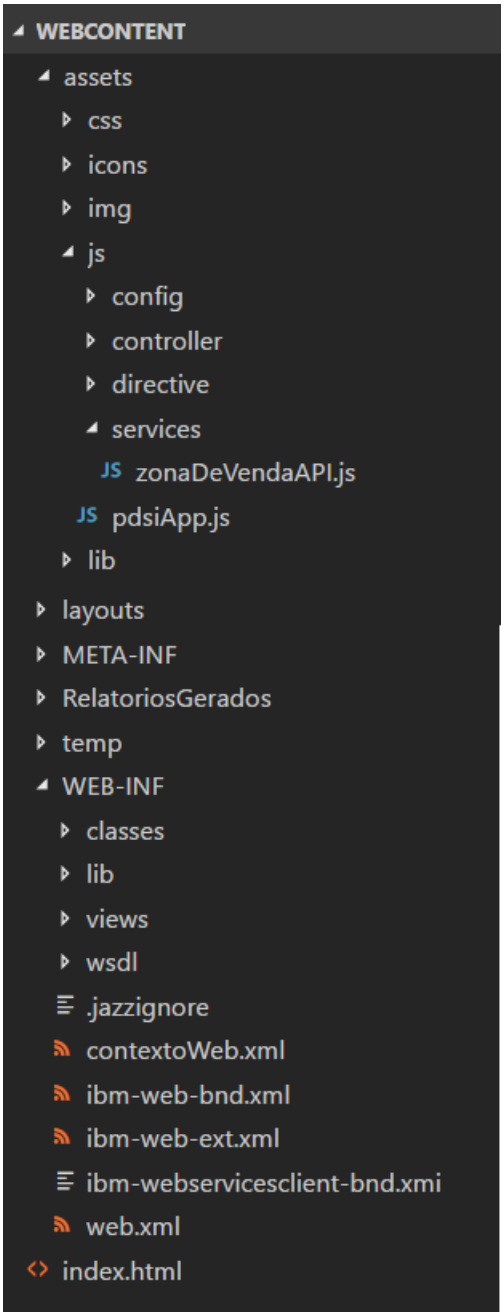


Figura 32 - Hierarquia de diretórios do projeto

A raiz WEBCONTENT contém todos os diretórios de front-end do projeto, sendo o foco desta secção na estrutura do SPA. Em editores/IDEs puramente de JavaScript esta pasta que é carregada para o desenvolvimento da aplicação.

A pasta assets deve conter arquivos necessários para a aplicação que não estão relacionados diretamente com o código AngularJS, dentre os arquivos estão:

- css – pasta para consolidar os arquivos Cascading Style Sheets que tem o objetivo de estilizar as páginas web.
- img e icons – pastas responsáveis por armazenar as imagens e icones da aplicação. Não é recomendado que a aplicação utilize muitas imagens, pois estes arquivos acabam deixando a aplicação mais pesada para download.
- js – A pasta js é a principal pasta da aplicação, nela devem ficar todos os arquivos JavaScript específicos da aplicação. Esta pasta deve ser organizada da seguinte forma:
    - config – pasta com arquivos JavaScript de configuração da aplicação. Nela devem conter arquivos de inicialização, configurações de rotas, arquivos com definição de constantes, entre outros.
    - controller – esta pasta deve conter os controladores angular da aplicação. Esta pasta poderá ser estruturada de duas formas. Se aplicação for pequena, todos os controllers podem ser armazenados nesta pasta sem subdivisões. Caso a aplicação seja grande, os controladores podem ser agrupados em pastas relativas as entidades. Ou seja, se existirem vários controladores para Zona de Venda, deverá ser criada uma pasta zonaVenda que conterá estes controladores.
    - directive – esta pasta deve agrupar as diretivas angular relativos a aplicação.
    - service – esta pasta deve agrupar os serviços angular relativos a aplicação.pdsi-app.js – este arquivo irá conter a criação dos módulos da aplicação, configuração de rotas e inicializações. Caso este arquivo fique grande, ele pode ser subdividido em outros arquivos que devem estar localizados na pasta config.
    - libs - esta pasta irá conter as bibliotecas utilizadas na aplicação, como por exemplo o próprio AngularJS. As bibliotecas podem ser divididas em subpastas.

- Caso seja necessário, outras pastas podem ser criadas para agrupamento de outros tipos de arquivos, como less,scss e fonts.

A pasta view irá conter os templates da aplicação. Esta pasta deve ser subdividida em pastas relativas aos objetos do sistema como descrito nos controladores. Desta forma, se existir um controlador para Zona de Venda, deve existir uma subpasta zonaVenda com todos os templates relativos a este objeto. A view poderá conter uma subpasta common que irá agrupar templates compartilhados por outros, por exemplo, um rodapé que deverá ser incluído em todas as páginas deverá ser refatorado em um arquivo e incluído nesta pasta.

A página index.html representa a página, template, da aplicação. Ela é o ponto de entrada da aplicação e importa os scripts necessários para execução.

5.3 Boas práticas e checklist de desenvolvimento

Diversos benefícios podem ser obtidos através de um código escrito com qualidade, como maior performance, melhoria na segurança, facilidade de manutenção, entre outros. Para isso é fundamental se atentar a boas práticas de programação. Neste item serão listadas algumas boas práticas e será apresentado um checklist para o desenvolvimento e inspeção do código.

Uma boa prática para desenvolvimento em AngularJS é a modularização de códigos referentes a artefatos que são utilizados em diversas partes do sistema, como, por exemplo, controles, diretivas, filtros, serviços e outros tipos. Esta prática não é obrigatória, mas é aconselhável. Esta abordagem provê como benefício a reutilização de código, o que facilita futuras manutenções, evitando que uma única atualização gere diversas modificações idênticas pelo código.

Uma boa prática é evitar ligar tudo ao $scope, porque muitas ligações aglutinam a lista de observação do loop $digest. Para evitar isso, deve ser utilizada a propriedade controllerAs, que permite que alias significativos sejam criados, facilitando o entendimento e manutenção do código da aplicação. Além disso, a utilização dessa propriedade evita erros na utilização de variáveis com o mesmo nome em controllers diferentes aninhados sob o MainController.

Outra boa prática é a redução de watches na aplicação, visto que o contexto dos watchers deve ser atualizado toda vez que o valor observado for modificado. Uma forma de reduzir watchers é utilizando o one-time-binding para dados que não mudam dinamicamente. Além disso, utilização da diretiva ng-repeat deve ser evitada sempre que possível, já que esta geralmente utiliza um vetor de objetos do $scope, que deteriora a performance da aplicação.

Com relação a segurança deve ser feita a validação dos dados em diversas camadas, isto é, devem ser implementadas validações tanto no AngularJS (lado cliente) como no serviço REST (lado servidor) ao receber os dados na chamada do serviço.

As implementações de segurança devem ser feitas em back-end, visto que os códigos JavaScripts rodam no browser dos usuários, o que permite alterações locais e criação de meios para "burlar" a camada de segurança implementada. Se tratando de aplicações WEB, deve-se verificar também as vulnerabilidades típicas deste tipo de aplicação, tais como: SQLInjections, Cross-Site Script (XSS), Cross Site Request Forgery (CSRF) entre outras.

Também é uma boa prática a criação de testes unitários para as funcionalidades criadas, a fim de realizar uma validação mais precisa e potencializar a qualidade da aplicação.

Desenvolvedores não devem utilizar o jQuery, para que não trabalhem mais pensando em padrões HTML, visando o entendimento do funcionamento do AngularJS. Isto é importante pois o AngularJS é um framework poderoso com diversos recursos, e estes recursos podem estar sendo subutilizados nesses casos provocando perda na qualidade do sistema.

É uma boa prática a criação de um documento padrão de inspeção de código, a fim de garantir que a validação seja realizada corretamente seja qual for a pessoa que esteja executando o documento. A seguir estão os critérios que são validados. Somente devem ser homologados os códigos que atendam a todos os critérios.

- A estrutura de pastas deve seguir o tipo Specific Style.
- O código-fonte deverá ser modularizado, focando em benefícios como reuso e performance.
- O uso de templates deve ser feita sempre utilizando partials.
- Utilizar a propriedade controllerAs.
- Quando o projeto for homologar, o mesmo deverá se minificado com objetivo de melhorar a performance.
- Os testes unitários devem cobrir 60% do código-fonte da aplicação.
- A aplicação deverá tratar os principais retornos do protocolo HTTP de um serviço REST.
- A utilização do one-data biding sempre que os dados não forem alterados, apenas exibidos na interface da aplicação.
- Evitar o uso do jQuery.

- Não utilizar o prefixo "$" no nome de varáveis, propriedades ou métodos.
- Não utilizar o prefixo "ng" para nomear diretivas.
- Utilizar camelCase para nomear os módulos , filtros e diretivas.
- Utilizar PascalCase + Controller para nomear os controllers.
- Utilizar PascalCase para serviços construtores e camelCase para outros tipos de serviços
- Os scripts devem estar no final do documento.
- A importação dos arquivos css deve estar dentro da tag <head>.
- Utilizar o SSL para comunicação entre os serviços REST e a aplicação.
- Verificar boas práticas de segurança (possíveis Cross-Site Script, Injections, alterações no JSON, entre outras)
- Verificar a quantidade/necessidade de watches por página
- Validação de dados no AngularJS e no back-end.

6 Integração Rest

Como pode ser visto na Figura 33, as aplicações SPA representam o front-end da aplicação que é executado no browser do usuário enquanto back-end se encarrega de enviar e receber informações do front-end, sendo hospedado em um servidor e constituído por outras tecnologias e ferramentas. No caso da Ipiranga o servidor é o Websphere e o back-end é desenvolvido em Java.

Como a Ipiranga já possui uma infraestrutura de sistemas bem definida e flexível, como pode ser visto neste documento, a estrutura do back-end irá se basear no framework da Ipiranga. Todas as regras e definições contidos neste documento devem ser seguidos para o back-end de aplicações SPA. A única diferença são os controladores, que serão detalhados abaixo.

```
1  @Controller
2  public class ZonaDeVendaRestController extends ControladorGenerico{
3
4      @Autowired
5      private ServicosDePontoVenda servicoPontoVenda;
6
7      @RequestMapping(method = RequestMethod.GET, produces = "application/json")
8      public @ResponseBody List<ZonaVenda> listar(
9              @RequestParam(value = "cdZonven", required = false) String cdZonven,
10             @RequestParam(value = "cdGv"    , required = false) String cdGv) {
11         List<ZonaVenda> lista = servicoPontoVenda.obterZonasVendaPor(cdGv,cdZonven);
12         return lista;
13     }
14 }
```

Figura 33 - Exemplo de Controller

Como pode ser visto na Figura 33 o Controller irá permanecer com anotações semelhantes às encontradas no Spring MVC para aplicações tradicionais. Entretanto, o controlador irá atender solicitações REST e responder JSON ao invés de renderizar páginas. Para isso, a anotação @RequestMapping deverá definir:

- O método HTTP que a action irá responder.

- GET: Está requisitando ao serviço um determinado recurso, por exemplo, uma listagem de dados;
- POST: Indica ao serviço que ele deve receber o recurso que está sendo enviado e adicionar em algum repositório;
- PUT: Indica ao serviço que o recurso que está sendo enviado deve ser alterado se ele já existir, ou ainda, pode ser adicionado caso ele ainda não exista;
- DELETE: Indica que o serviço deve excluir o recurso.
- Existem outros métodos, mas não são muito utilizados.

- O que é produzido pela action, no caso como as respostas serão em JSON, deve ser utilizado "application/json".

Além do @RequestMapping, deve ser utilizada a anotação @ResponseBody que indica que o retorno estará vinculado ao corpo da resposta, retornando assim o texto limpo. Como pode ser visto no método, o retorno é uma lista de objetos, para transformar o retorno em JSON deve ser configurado o framework Jackson que será mais detalhado abaixo.

Outra anotação utilizada é o @RequestParam que deverá indicar o tipo de informação suportada pela action. Alguns dados podem ser passados para esta anotação incluindo o alias, se é obrigatório e o valor default.

6.1.1 Jackson

O Jackson é uma biblioteca para processamento de JSON em Java. Além do JSON ela possui módulos para processamento de outros tipos de arquivo como CSV, Properties e XML. Para o framework da Ipiranga, ela deve ser utilizada para fazer a conversão das entidades para JSON e do JSON para as entidades.

Para a utilização do Jackson, algumas configurações precisam ser realizadas no projeto. Primeiramente, é preciso adicionar as bibliotecas Jackson Annotations, Core e Databind na versão previamente aprovada pela equipe de Arquitetura da Ipiranga (versão 2.6.7). Também é necessário fazer algumas configurações no contextoWeb.xml descritas abaixo:

É preciso fazer a inclusão do conversor de beans, que lê e escreve JSON, no objeto responsável por tratar requisições HTTP. A Figura 34 ilustra esta configuração.

```
<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
    <property name="order" value="2" />
    <property name="messageConverters">
        <list>
            <bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"/>
        </list>
    </property>
</bean>
```

Figura 34 - Configuração Conversor

Além desta inclusão, é preciso ajustar a ordenação dos beans para a entrada da configuração realizada acima. A Figura 35 ilustra a reordenação.

```
<beans ...>
    <mvc:resources mapping="/assets/**" location="/assets/" order="0"/>
    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="order" value="1" />
        ...
    </bean>
    <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
        <property name="order" value="2" />
        ...
    </bean>
    <bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping" p:order="3" ...>
        ...
    </bean>
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"...>
        <property name="order" value="4" />
        ...
    </bean>
    <bean class="org.springframework.web.servlet.view.UrlBasedViewResolver" ...>
        <property name="order" value="5"/>
        ...
    </bean>
    ...
</beans>
```

Figura 35 - Reordenação dos beans

Assim, quando existir alguma action no controlador que retorne uma entidade e produza um JSON, o conversor automaticamente irá converter a entidade para um texto contendo o JSON. A entidade será convertida com todas as propriedades contidas na mesma. **ATENÇÃO**, relacionamentos configurados como Lazy resultarão em erros caso a propriedade não esteja carregada previamente do banco ou não esteja configurada para ser ignorada. Para ignorar uma propriedade da entidade basta adicionar a anotação @JsonIgnore.

```
...
@Entity
@Table(...)
public class ZonaVenda implements Serializable {

    @Id
    @Column(...)
    private String cdZonven;

    @JoinColumn(...)
    @ManyToOne(fetch = FetchType.LAZY)
    @JsonIgnore
    private GerenciaVenda gerenciaVenda;

    ...
}
```

Figura 36 - Ignorando uma propriedade

7 Referências

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional.

Meszaros, G (2007). xUnit Test Patterns: Refactoring Test Code. Addison-Wesley. ISBN 978-0-13-149505-0.

Fowler, M. (2007). Mocks Aren't Stubs. https://martinfowler.com/articles/mocksArentStubs.html

Wikipedians (2017). Software Testing. PediaPress.
https://books.google.com.br/books?id=o2mFgGjktncC

Wikipedia (2017). https://en.wikipedia.org/wiki/Mock_object